All for One and One for All: Program Logics for Exploiting Internal Determinism in Parallel Programs

Extended Version

ALEXANDRE MOINE, New York University, USA
SAM WESTRICK, New York University, USA
JOSEPH TASSAROTTI, New York University, USA

Nondeterminism makes parallel programs challenging to write and reason about. To avoid these challenges, researchers have developed techniques for internally deterministic parallel programming, in which the steps of a parallel computation proceed in a deterministic way. Internal determinism is useful because it lets a programmer reason about a program as if it executed in a sequential order. However, no verification framework exists to exploit this property and simplify formal reasoning about internally deterministic programs.

To capture the essence of why internally deterministic programs should be easier to reason about, this paper defines a property called schedule-independent safety. A program satisfies schedule-independent safety, if, to show that the program is safe across all orderings, it suffices to show that one terminating execution of the program is safe. We then present a separation logic called Musketeer for proving that a program satisfies schedule-independent safety. Once a parallel program has been shown to satisfy schedule-independent safety, we can verify it with a new logic called Angelic, which allows one to dynamically select and verify just one sequential ordering of the program.

Using Musketeer, we prove the soundness of MiniDet, an affine type system for enforcing internal determinism. MiniDet supports several core algorithmic primitives for internally deterministic programming that have been identified in the research literature, including a deterministic version of a concurrent hash set. Because any syntactically well-typed MiniDet program satisfies schedule-independent safety, we can apply Angelic to verify such programs.

All results in this paper have been verified in Rocq using the Iris separation logic framework.

CCS Concepts: • **Theory of computation → Program verification**; **Separation logic**; • **Software and its engineering → Parallel programming languages**.

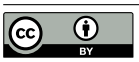
Additional Key Words and Phrases: program verification, separation logic, parallelism, determinism

ACM Reference Format:
Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2026. All for One and One for All: Program Logics for Exploiting Internal Determinism in Parallel Programs: Extended Version. *Proc. ACM Program. Lang.* 10, POPL, Article 26 (January 2026), 33 pages. https://doi.org/10.1145/3776668

1 Introduction

One of the most challenging aspects of concurrent and parallel programming is dealing with nondeterminism. Nondeterminism complicates almost every aspect of trying to make programs correct. Bugs often arise because programmers struggle to reason about the set of all possible nondeterministic outcomes and interleavings. Finding those bugs becomes more difficult, as testing can only cover a subset of possible outcomes. Even when bugs are found, nondeterminism makes them

Authors' Contact Information: Alexandre Moine, alexandre.moine@nyu.edu, New York University, New York, USA; Sam Westrick, shw8119@nyu.edu, New York University, New York, USA; Joseph Tassarotti, jt4767@nyu.edu, New York University, New York, USA.



ACM 2475-1421/2026/1-ART26
https://doi.org/10.1145/3776668

harder to reproduce and debug. These challenges also extend to formal methods for such programs, where nondeterminism makes various analyses and verification techniques more complex.

For these reasons, there has long been interest in methods for *deterministic* parallel programming. A range of algorithmic techniques [Blelloch et al. 2012], language designs [Blelloch et al. 1994; Kuper et al. 2014a], type systems [Bocchino Jr. et al. 2009], specialized operating systems and runtimes [Aviram et al. 2010], and various other approaches have been developed for making parallel programs deterministic. Researchers in this area have long noted that determinism is not simply a binary property, and in fact there is a spectrum of *degrees* of determinism. On one end of the spectrum is *external determinism*, which simply says that the input/output behavior of a program is deterministic. However, in an externally deterministic program, even if the final output is deterministic, the manner in which the computation takes place may be highly nondeterministic and vary across runs. As a result, external determinism does not eliminate all of the programming challenges associated with nondeterminism. For example, a programmer who attaches a debugger to an externally deterministic program may still see different internal behaviors across different runs, complicating efforts to understand the program's behavior.

A stronger property, called *internal determinism*, requires in addition that the structure and internal steps of a computation are deterministic. More formally, in an internally deterministic program, for a given input, every execution will generate the same *computation graph*, a trace that captures the dependencies of operations and their results. With this strong form of determinism, we can reason about the program's behavior by considering any *one* sequential traversal of operations in the computation graph. This is useful, because as Blelloch et al. [2012] put it:

> In addition to returning deterministic results, internal determinism has many advantages including ease of reasoning about the code, ease of verifying correctness, ease of debugging, ease of defining invariants, ease of defining good coverage for testing, and ease of formally, informally and experimentally reasoning about performance.

Although ensuring internal determinism might seem expensive, Blelloch et al. [2012] have shown that by using a core set of algorithmic techniques and building blocks, it is possible to develop fast and scalable internally deterministic algorithms for a range of benchmark problems.

In this paper, we explore the meaning and benefits of internal determinism from the perspective of program verification. If one of the advantages of internal determinism is that it simplifies reasoning about programs, then it should be possible to exploit this property in the form of new reasoning rules in a program logic. To do so, we first define a property we call *schedule-independent safety*, which holds for a parallel program e if, to verify that every execution of e is safe (*i.e.* never triggers undefined behavior or a failing assert), it suffices to prove that at least *one* interleaving of operations in e is terminating and safe. Internal determinism implies schedule-independent safety, and it is this property that makes reasoning about internally deterministic programs simpler. Schedule-independent safety recalls the motto of Dumas' Three Musketeers, "all for one and one for all": the safety of all interleavings amounts to the safety of one of them. Building on this observation, we develop Musketeer, a separation logic for proving that a program satisfies schedule-independent safety. Although Musketeer is formulated as a unary program logic, schedule-independent safety is a $\forall\forall$ *hyperproperty* [Clarkson and Schneider 2010], since it relates safety of any chosen execution of a program e to all other executions of e. Thus, to prove the soundness of Musketeer, we encode Musketeer triples into a new relational logic called ChainedLog. In contrast to most prior relational concurrent separation logics, which are restricted to $\forall\exists$ hyperproperties, ChainedLog supports $\forall\forall$ hyperproperties using a judgement we call a *chained triple*.

Intuitively, the high-level reasoning rules of Musketeer restrict the user to verify only internally-deterministic programs. However, while internally deterministic programs always satisfy schedule-independent safety, the converse is false: a program may be nondeterministic because it observes actions from concurrent tasks, but it may do so without jeopardizing safety. In order to verify such programs (§7.2, §7.3), we use the fact that Musketeer is defined in terms of the more flexible and more complex ChainedLog, and drop down to this low-level logic to conduct the proof.

We next explore how to exploit schedule-independent safety to simplify verification of programs. To that end, we present a logic called Angelic that allows one to *angelically* select and verify one sequential ordering of operations in a parallel program. Angelic is sound to apply to programs that satisfy schedule-independent safety because the safety and termination of the one ordering verified during the proof will imply safety for all other executions. This is in contrast to standard concurrent separation logics, in which one must consider *all* possible orderings during a proof.

Using these logics, we verify a number of examples from the literature on internal determinism and related properties. First, we show how to use Musketeer to prove properties about language-based approaches for enforcing internal determinism. In particular, because Musketeer is a higher-order impredicative logic, Musketeer can encode logical relations models for type systems that are designed to enforce internal determinism. We start by applying this to a simple ownership-based affine type system we call MiniDet. The resulting logical relations model for MiniDet shows that every well-typed program satisfies schedule-independent safety. Next we use Musketeer to prove specifications for *priority writes* and *deterministic concurrent hash sets*, two of the core primitives that Blelloch et al. [2012] use in several of their examples of internally deterministic algorithms. Using these specifications, we extend MiniDet and its logical relations model with typing rules for priority writes and hash sets, showing that schedule-independent safety is preserved.

Finally, putting these pieces together, we turn to parallel array deduplication, one of the example benchmark problems considered by Blelloch et al. [2012]. We first show that an implementation of the algorithm they propose for this problem can be syntactically-typed in MiniDet, thereby showing that it is schedule-independent safe. Next, we use Angelic to verify a correctness property for this algorithm. Although the algorithm is written using a parallel for-loop that does concurrent insertions into a hash set, by using Angelic, we can reason *as if* the parallel loop was a standard, sequential loop, thereby simplifying verification.

Contributions. The contributions of this paper are the following:

- We identify schedule-independent safety as a key property of deterministic parallel programs.
- We present Musketeer, a separation logic for proving that a program satisfies schedule-independent safety, meant to be used as a tool for proving automatic approaches correct.
- We present Angelic, a separation logic for proving that *one* interleaving safely terminates.
- We use Musketeer to verify properties of MiniDet, an affine type system guaranteeing schedule-independent safety.
- We verify that priority writes and a deterministic concurrent hash set satisfy schedule-independent safety using Musketeer, and then use this property to verify a deduplication algorithm using Angelic.
- We formally verify all the presented results [Moine et al. 2025], including the soundness of the logics and the examples, in the Rocq prover using the Iris framework [Jung et al. 2018].

2 Key Ideas

In this section, we first give a simple motivating example (§2.1), describe some of the core concepts behind how Musketeer guarantees schedule-independent safety (§2.2), and conclude by showing some of the rules of Angelic that allow for reasoning sequentially about a parallel program (§2.3).

M-Assert
$\{\top\}\ \mathsf{assert}\, v\ \{\lambda w\ \_.\ \ulcorner w = () \wedge v = \mathsf{true}\urcorner\}$

M-KSplit
$\mathsf{counter}\, v\, (q_1 + q_2)\, (i_1 + i_2)\ \dashv\vdash\ \mathsf{counter}\, v\, q_1\, i_1 * \mathsf{counter}\, v\, q_2\, i_2$

M-KRef
$\{\top\}\ \mathsf{ref}\, i\ \{\lambda v\ \_.\ \mathsf{counter}\, v\, 1\, i\}$

M-KAdd
$\{\mathsf{counter}\, v\, q\, i\}\ \mathsf{atomic\_add}\, v\, j\ \{\lambda \_\ \_.\ \mathsf{counter}\, v\, q\, (i + j)\}$

M-KGet
$\{\mathsf{counter}\, v\, 1\, i\}\ \mathsf{get}\, v\ \{\lambda w\ \_.\ \ulcorner w = i\urcorner * \mathsf{counter}\, v\, 1\, i\}$

M-Par
$$\frac{\{P_1\}\ e_1\ \{Q_1\} \qquad \{P_2\}\ e_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathsf{par}\, e_1\, e_2\ \{\lambda v\, x.\ \exists v_1\, v_2\, x_1\, x_2.\ \ulcorner v = (v_1, v_2) \wedge x = (x_1, x_2)\urcorner * Q_1\, v_1\, x_1 * Q_2\, v_2\, x_2\}}$$

Fig. 1. Reasoning Rules for a Concurrent Counter and Key Reasoning Rules of Musketeer

2.1 A Motivating Example

Our example program is named dumas and appears below:

$$\begin{aligned}\mathsf{dumas} \triangleq\ & \lambda n.\ \mathsf{let}\, r\ =\ \mathsf{ref}\, 0\ \mathsf{in}\\ & \quad \mathsf{par}\ (\lambda\_.\ \mathsf{atomic\_add}\, r\, 1802)\ (\lambda\_.\ \mathsf{atomic\_add}\, r\, 42);\\ & \quad \mathsf{assert}\ (\mathsf{get}\, r == n)\end{aligned}$$

The dumas program takes an argument n. It first allocates a reference r initialized to 0, and then calls in parallel two closures, one that atomically adds 1802 to r, and the other that atomically adds 42. After the parallel phase, the function asserts that the content of r is equal to n.

Imagine we wish to prove that (dumas 1844) is safe—that is, for every interleaving, the program will never get stuck, and in particular the assertion will succeed. Of course, many existing concurrent separation logics can easily prove this. In such logics, one can use an *invariant* assertion to reason about the shared access to r by the two parallel threads. This invariant would ensure that, no matter which order the threads perform their additions, after both have finished r will contain 1844.

We propose an alternate approach that simplifies reasoning by exploiting the internal determinism in programs like dumas. In our approach, we first prove in a light-weight way that, for any given value of n, the order of the parallel additions in (dumas n) does not affect the outcome of the assert. Then, to prove safety of (dumas n) for the specific value of $n = 1844$, we can just pick *one* possible ordering and verify safety of that ordering.

2.2 Verifying Schedule-Independent Safety with Musketeer

Our first contribution is Musketeer, a logic for proving that a program satisfies schedule-independent safety, *i.e.* that safety of any one complete execution implies safety of all possible executions. Although Musketeer is itself a program logic, we stress that Musketeer is not meant to be used directly. Rather, Musketeer is a kind of intermediate logic designed for proving the soundness of other light-weight, automatic approaches of ensuring schedule-independent safety such as type systems. For instance, our main case study focuses on using Musketeer to show the soundness of an affine type system guaranteeing schedule-independent safety (§7). Nevertheless, for the sake of explaining the ideas behind Musketeer, here we explain the reasoning rules that would allow one to verify manually the schedule-independent safety of (dumas n) for all n.

Key reasoning rules. Musketeer takes the form of a unary separation logic with triples written $\{P\}\ e\ \{Q\}$, where P is a precondition, e the program being verified and Q the postcondition. The postcondition Q is of the form $\lambda v\, x.\, R$, where v is the value being returned by the execution of e and x is a *ghost return value*. We explain ghost return values in detail later, but for now, they can be thought of as a special way to existentially quantify variables in the postcondition. This

Musketeer triple guarantees the following hyper-property: "if one execution of e is safe starting from a heap satisfying P and terminates, then every execution of e is safe starting from a heap satisfying P and all terminating executions will end in a heap satisfying Q". The upper part of Figure 1 shows the main reasoning rules we use for our example (in these rules, the horizontal bar is an implication in the meta-logic). While the assertions and rules of Musketeer are similar to standard separation logic rules, there are two key differences. First, Musketeer *does not* provide the usual disjunction or existential elimination rules from separation logic. That is, to prove a triple of the form $\{P_1 \lor P_2\}\ e\ \{Q\}$, we *cannot* in general do case analysis on the precondition and reduce this to proving $\{P_1\}\ e\ \{Q\}$ and $\{P_2\}\ e\ \{Q\}$. As we will see later, this restriction is necessary because the imprecision in disjunctions and existentials can encode nondeterministic behavior, where different executions pick different witnesses.

Second, unlike traditional separation logic rules, rules in Musketeer *do not guarantee safety*. Rather, they guarantee that *safety is independent of scheduling*. Thus, these rules often have weaker preconditions than standard separation logic rules. The rule M-Assert illustrates this unusual aspect of Musketeer. This rule applies to an expression $\mathsf{assert}\, v$, for an arbitrary value v, and has a *trivial* precondition. The postcondition has the pure facts that the return value w is $()$ and that v equals true, *i.e.* that the assert did not fail. In contrast, the standard separation logic rule for $\mathsf{assert}\, v$ requires the user to *prove* that $v = \mathsf{true}$! This is because the expression $\mathsf{assert}\, v$ is safe only if the value $v = \mathsf{true}$ (§3.2). So in conventional separation logic, where a triple implies safety, the obligation is to show that the assert will be safe. However, in Musketeer, the rule M-Assert corresponds exactly to the "motto" of Musketeer triples: if one execution of $\mathsf{assert}\, v$ is safe and terminates with value w such that $w = ()$ and $v = \mathsf{true}$, then every execution of $\mathsf{assert}\, v$ is safe and terminates with value $w = ()$, and $v = \mathsf{true}$ in those executions too. This property is true in a trivial way: since the argument v in $\mathsf{assert}\, v$ is already a value, there is only one possible safe execution for $\mathsf{assert}\, v$, and such an execution is possible only if $v = \mathsf{true}$.[1]

On the contrary, M-Par has a standard shape. This rule allows for verifying the parallel primitive $\mathsf{par}\, e_1\, e_2$. It requires the user to split the precondition into two parts P_1 and P_2, and to establish the two triples $\{P_1\}\ e_1\ \{Q_1\}$ and $\{P_2\}\ e_2\ \{Q_2\}$. The postcondition of the rule asserts that the value v being returned is an immutable pair (v_1, v_2) and the ghost return value x is itself a pair of two ghost return values x_1 and x_2, such that $Q\, v_1\, x_1$ and $Q\, v_2\, x_2$ hold.

Verifying dumas. The other rules in Figure 1 are the reasoning rules for the concurrent counter we use in dumas. They make use of a predicate $\mathsf{counter}\, v\, q\, i$, asserting that v is a concurrent counter with fractional ownership $q \in (0; 1]$. When $q = 1$ the assertion represents exclusive ownership of the counter, in which case i is the value stored in the counter. Otherwise, it asserts ownership of a partial *share* of the counter, and i is the contribution added to the counter with this share. M-KSplit shows that counter can be split into several shares. M-KRef verifies $\mathsf{ref}\, i$, has a trivial precondition and returns a counter initialized to i with fraction 1. M-KAdd verifies $\mathsf{atomic\_add}\, v\, j$, where the share may have an arbitrary fraction. M-KGet verifies $\mathsf{get}\, v$, requiring that $\mathsf{counter}\, v\, 1\, i$ holds. The fraction is 1, preventing a concurrent add to v. Such a concurrent add would introduce nondeterminism based on the relative ordering of the add and get, thereby breaking schedule-independent safety.

Using the above rules, we can show that for any n, $\{\top\}\ (\mathsf{dumas}\, n)\ \{\lambda\_\_.\ \top\}$, that is, without precondition, the safety of $(\mathsf{dumas}\, n)$ is scheduling independent. To do so, we use M-KRef to

[1]Note that Musketeer supports a bind rule (M-Bind, Figure 7) that allows the user to reason under an evaluation context. Hence, Musketeer supports reasoning on an expression $(\mathsf{assert}\, e)$, for an arbitrary expression e. To conduct such a proof, the user should first apply M-Bind and focus on e, show that e itself satisfies schedule-independent safety, and then for any value v to which e may reduce, apply M-Assert on the remaining expression $(\mathsf{assert}\, v)$.

$$
\begin{array}{rcll}
\top & \vdash & \mathsf{run}\,(\mathsf{assert}\,\mathsf{true})\,\{\lambda v.\ulcorner v = ()\urcorner\} & \textsc{A-Assert} \\
\mathsf{run}\,e_1\,\{\lambda v_1.\,\mathsf{run}\,e_2\,\{\lambda v_2.\,\psi\,(v_1,v_2)\}\} & \vdash & \mathsf{run}\,(\mathsf{par}\,e_1\,e_2)\,\{\psi\} & \textsc{A-ParSeqL} \\
\mathsf{run}\,e_2\,\{\lambda v_2.\,\mathsf{run}\,e_1\,\{\lambda v_1.\,\psi\,(v_1,v_2)\}\} & \vdash & \mathsf{run}\,(\mathsf{par}\,e_1\,e_2)\,\{\psi\} & \textsc{A-ParSeqR} \\
\top & \vdash & \mathsf{run}\,(\mathsf{ref}\,i)\,\{\lambda v.\,\exists \ell.\ulcorner v = \ell\urcorner * \ell \mapsto i\} & \textsc{A-Ref} \\
\ell \mapsto i & \vdash & \mathsf{run}\,(\mathsf{atomic\_add}\,v\,j)\,\{\lambda\_.\,\ell \mapsto (i+j)\} & \textsc{A-Add} \\
\ell \mapsto i & \vdash & \mathsf{run}\,(\mathsf{get}\,\ell)\,\{\lambda v.\ulcorner v = i\urcorner * \ell \mapsto i\} & \textsc{A-Get}
\end{array}
$$

Fig. 2. Reasoning Rules for a Concurrent Counter and Key Reasoning Rules of Angelic

initialize the counter, getting counter r 1 0, which we split into counter r $(1/2)$ 0 $*$ counter r $(1/2)$ 0, and then use M-Par. The counter r $(1/2)$ 0 given to each thread is sufficient to reason about the add they each perform, and when we combine the shares they give back, we get counter r 1 1844. Using M-KRef, we know that the get r returns 1844, leaving us to show $\{\top\}$ assert $(1844 == n)$ $\{\lambda\_\_.\ \top\}$.

At this point, we would get stuck in a standard separation logic proof, because the standard rule for assert would require us to prove that $(1844 == n)$ evaluates to true. However, that would only be the case if n was in fact 1844. Instead, in Musketeer, we can use a rule showing that $(1844 == n)$ will evaluate to some Boolean b, regardless of what value n is. At that point, we can use M-Assert to conclude, even though we don't know which value b will take.

2.3 Verifying That One Interleaving is Safe and Terminates with Angelic

Now that we know that for all n, (dumas n) satisfies schedule-independent safety, we can prove that (dumas 1844) is safe just by showing that *one* interleaving is safe and terminates. For such a simple example, it would suffice at this point to simply execute (dumas 1844) once and observe one safe, terminating execution. We would then be able to conclude that all possible executions are safe. However, for more complex examples (for example, programs that are parameterized by an argument from an infinite type), we propose Angelic, a program logic for verifying that one interleaving is safe and terminates.

Angelic uses a form of weakest-precondition reasoning, with specifications taking the form $\varphi \vdash \mathsf{run}\,e\,\{\psi\}$, where φ is the precondition, e the program being verified, and ψ the postcondition, of the form $\lambda v.\,\varphi'$, where v is the value being returned. In order to guarantee termination, Angelic's WP is defined as a *total weakest precondition*, that is, the WP is defined as a least fixpoint and does not mention the so-called later modality. Such a construction is standard, Krebbers et al. [2025, §4] describes the differences between a WP for partial and total correctness. Hence, $\mathsf{run}\,e\,\{\lambda\_.\ \top\}$ guarantees that one execution of e is safe and terminates.

Figure 2 presents a few reasoning rules for Angelic. It is helpful to read these rules backwards, applying the rule to a goal that matches the right side of the turnstile $\vdash$ and ending up with a goal of proving the left side. A-Assert verifies an assertion, for which the argument must be the Boolean true.[2] Indeed, since Angelic guarantees safety, the proof burden is now to show that the assert will succeed. A-ParSeqL says that to verify par e_1 e_2, it suffices to verify sequentially e_1 and then e_2. A-ParSeqR lets us verify the reverse order instead, reasoning first about e_2 and then e_1. As we will explain later on (§6.2), Angelic more generally allows for selecting *any* interleaving of steps within e_1 and e_2 by "jumping" between the two expressions during a proof. Finally, A-Ref, A-Add and A-Get shows how to reason on a concurrent counter. First, these rules do not involve any new predicate, and manipulate the plain points-to assertion linked with the counter. Second, no fractions or invariants are involved. Indeed, in Angelic, there is no need to split and join assertions, as the parallel primitive can be verified sequentially in any order.

[2]Angelic supports a bind rule (A-Bind, Figure 11). As in Musketeer, A-Bind allows for reasoning under an evaluation context. In combination with A-Assert, the user may reason about an expression (assert e) for an arbitrary expression e.

Values $\mathcal{V}$	$v ::= ()$	$\mid b \in \{\mathsf{true}, \mathsf{false}\}$	$\mid i \in \mathbb{Z}$	$\mid \ell \in \mathcal{L}$	$\mid (v, v) \mid \hat{\mu} f\, x.\, e$
Primitives	$\bowtie ::= +$	$\mid -$	$\mid \times$	$\mid \div$	$\mid \mathsf{mod} \mid == \mid < \mid \leq \mid > \mid \geq \mid \vee \mid \wedge$

Expressions $e ::=$	v, w	*value*	$\mathsf{assert}\, e$	*assertion*
	x	*variable*	$\mathsf{alloc}\, e$	*array allocation*
	$\mathsf{let}\, x = e\, \mathsf{in}\, e$	*sequencing*	$e[e]$	*array load*
	$\mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e$	*conditional*	$e[e] \leftarrow e$	*array store*
	$\mu f\, x.\, e$	*abstraction*	$\mathsf{length}\, e$	*array length*
	$e\, e$	*call*	$\mathsf{par}\, e\, e$	*parallelism*
	$e \bowtie e$	*primitive operation*	$e \,\|\|\, e$	*active parallel tuple*
	$\mathsf{prod}\, e\, e$	*product*	$\mathsf{CAS}\, e\, e\, e$	*compare-and-swap*
	$\mathsf{proj}_{k \in \{1,2\}}\, e$	*projections*		

Contexts $K ::=$	$\mathsf{let}\, x = \square\, \mathsf{in}\, e$	$\mid \mathsf{if}\, \square\, \mathsf{then}\, e\, \mathsf{else}\, e$	$\mid \mathsf{alloc}\, \square$	$\mid \mathsf{length}\, \square$	$\mid \mathsf{assert}\, \square$
	$\mid e[\square]$	$\mid \square[v]$	$\mid e[e] \leftarrow \square$	$\mid e[\square] \leftarrow v$	$\mid \square[v] \leftarrow v$
	$\mid e \bowtie \square$	$\mid \square \bowtie v$	$\mid e\, \square$	$\mid \square\, v$	
	$\mid \mathsf{CAS}\, e\, e\, e\, \square$	$\mid \mathsf{CAS}\, e\, e\, \square\, v$	$\mid \mathsf{CAS}\, e\, \square\, v\, v$	$\mid \mathsf{CAS}\, \square\, v\, v\, v$	
	$\mid \mathsf{prod}\, e\, \square$	$\mid \mathsf{prod}\, \square\, v$	$\mid \mathsf{proj}_k\, \square$		

Fig. 3. Syntax of MusketLang

Using these rules, we can verify that $\vdash \mathsf{run}\,(\mathsf{dumas}\,1844)\,\{\lambda\_.\, \top\}$ holds, which implies that there exists one interleaving that is safe and terminates. Combined with the fact that this program has schedule-independent safety, we conclude that $(\mathsf{dumas}\,1844)$ is always safe.

3 Syntax and Semantics

MusketLang is a call-by-value lambda calculus with mutable state and parallelism. We first present its syntax (§3.1) and then its semantics (§3.2). MusketLang is similar to HeapLang, the language that ships with Iris, except that it implements structured parallelism instead of fork-based concurrency.

3.1 Syntax

Figure 3 presents the syntax of MusketLang. A value $v \in \mathcal{V}$ is either the unit value $()$, a Boolean $b \in \{\mathsf{true}, \mathsf{false}\}$, an idealized integer $i \in \mathbb{Z}$, a location ℓ from an infinite set of locations $\mathcal{L}$, an immutable product (v_1, v_2) of two values, or a recursive function $\hat{\mu} f\, x.\, e$.

An expression e describe a computation in MusketLang. Recursive functions are written $\mu f\, x.\, e$. For non-recursive functions, we write $\lambda x.\, e \triangleq \mu\_\, x.\, e$. We define functions with multiple arguments as a chain of function constructors. Mutable state is available through arrays. Parallelism is available through a primitive $\mathsf{par}\, e_1\, e_2$, which evaluates to an *active parallel tuple* $e_1 \,||\, e_2$. Such a tuple evaluates the two expressions in parallel and returns their result as an immutable product. MusketLang also has a primitive compare-and-swap instruction $\mathsf{CAS}\, e_1\, e_2\, e_3\, e_4$, which targets an array entry and has 4 parameters: the array location, the offset into the array, the old value and the new value. References are defined as arrays of size 1 with the following operations:

$$\mathsf{ref} \triangleq \lambda x.\, \mathsf{let}\, r = \mathsf{alloc}\, 1\, \mathsf{in}\, r[0] \leftarrow x;\, r \qquad \mathsf{get} \triangleq \lambda r.\, r[0] \qquad \mathsf{set} \triangleq \lambda r\, v.\, r[0] \leftarrow v$$

An evaluation context K describes an expression with a hole $\square$ and dictates the right-to-left evaluation order of MusketLang.

3.2 Semantics

Figure 4 presents the head reduction relation $e \setminus \sigma \xrightarrow{\mathsf{head}} e' \setminus \sigma'$, describing a single step of expression e with store σ into expression e' and store σ'. A store is a map from location to arrays, modeled as a list of values. We write $\emptyset$ for the empty store and $\sigma(\ell)$ for the list of values at location ℓ in σ. To insert or update a location ℓ with array $\vec{v}$ in store σ, we write $[\ell := \vec{v}]\sigma$, and similarly write

$$\textsc{HeadIfTrue}\quad \text{if true then } e_1 \text{ else } e_2 \setminus \sigma \xrightarrow{\text{head}} e_1 \setminus \sigma$$

$$\textsc{HeadIfFalse}\quad \text{if false then } e_1 \text{ else } e_2 \setminus \sigma \xrightarrow{\text{head}} e_2 \setminus \sigma$$

$$\textsc{HeadCallPrim}\quad \frac{v_1 \bowtie v_2 \xrightarrow{\text{pure}} v}{v_1 \bowtie v_2 \setminus \sigma \xrightarrow{\text{head}} v \setminus \sigma}$$

$$\textsc{HeadAbs}\quad \mu f\, x.\, e \setminus \sigma \xrightarrow{\text{head}} \hat{\mu} f\, x.\, e \setminus \sigma$$

$$\textsc{HeadLetVal}\quad \text{let } x = v \text{ in } e \setminus \sigma \xrightarrow{\text{head}} [v/x]e \setminus \sigma$$

$$\textsc{HeadAlloc}\quad \frac{0 \le i \qquad \ell \notin \mathrm{dom}(\sigma)}{\text{alloc } i \setminus \sigma \xrightarrow{\text{head}} \ell \setminus [\ell := ()^i]\sigma}$$

$$\textsc{HeadLoad}\quad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\ell[i] \setminus \sigma \xrightarrow{\text{head}} v \setminus \sigma}$$

$$\textsc{HeadStore}\quad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}|}{\ell[i] \leftarrow v \setminus \sigma \xrightarrow{\text{head}} () \setminus [\ell := [i := v]\vec{w}]\sigma}$$

$$\textsc{HeadAssert}\quad \text{assert true} \setminus \sigma \xrightarrow{\text{head}} () \setminus \sigma$$

$$\textsc{HeadProduct}\quad \text{prod } v_1\, v_2 \setminus \sigma \xrightarrow{\text{head}} (v_1, v_2) \setminus \sigma$$

$$\textsc{HeadProj}\quad \frac{k \in \{1; 2\}}{\text{proj}_k\, (v_1, v_2) \setminus \sigma \xrightarrow{\text{head}} v_k \setminus \sigma}$$

$$\textsc{HeadLength}\quad \frac{\sigma(\ell) = \vec{w} \qquad i = |\vec{w}|}{\text{length } \ell \setminus \sigma \xrightarrow{\text{head}} i \setminus \sigma}$$

$$\textsc{HeadCASSucc}\quad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\text{CAS } \ell\, i\, v\, v' \setminus \sigma \xrightarrow{\text{head}} \text{true} \setminus [\ell := [i := v']\vec{w}]\sigma}$$

$$\textsc{HeadCASFail}\quad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v_0 \qquad v_0 \ne v}{\text{CAS } \ell\, i\, v\, v' \setminus \sigma \xrightarrow{\text{head}} \text{false} \setminus \sigma}$$

$$\textsc{HeadCall}\quad (\hat{\mu} f\, x.\, e)\, v \setminus \sigma \xrightarrow{\text{head}} [(\hat{\mu} f\, x.\, e)/f][x/v]e \setminus \sigma$$

$$\textsc{HeadFork}\quad \text{par } e_1\, e_2 \setminus \sigma \xrightarrow{\text{head}} e_1 \,||\, e_2 \setminus \sigma$$

$$\textsc{HeadJoin}\quad v_1 \,||\, v_2 \setminus \sigma \xrightarrow{\text{head}} (v_1, v_2) \setminus \sigma$$

Fig. 4. Head Reduction Relation

$$\textsc{StepHead}\quad \frac{e \setminus \sigma \xrightarrow{\text{head}} e' \setminus \sigma'}{e \setminus \sigma \longrightarrow e' \setminus \sigma'}$$

$$\textsc{StepCtx}\quad \frac{e \setminus \sigma \longrightarrow e' \setminus \sigma'}{K\langle e \rangle \setminus \sigma \longrightarrow K\langle e' \rangle \setminus \sigma'}$$

$$\textsc{StepParL}\quad \frac{e_1 \setminus \sigma \longrightarrow e_1' \setminus \sigma'}{e_1 \,||\, e_2 \setminus \sigma \longrightarrow e_1' \,||\, e_2 \setminus \sigma'}$$

$$\textsc{StepParR}\quad \frac{e_2 \setminus \sigma \longrightarrow e_2' \setminus \sigma'}{e_1 \,||\, e_2 \setminus \sigma \longrightarrow e_1 \,||\, e_2' \setminus \sigma'}$$

Fig. 5. Main Reduction Relation

$[i := w]\vec{v}$ to update offset i with value w in array $\vec{v}$. The length of an array $\vec{v}$ is written as $|\vec{v}|$, and v^i represents an array of size i initialized with value v.

Most of the reduction rules are standard. For example, HeadAlloc allocates an array initialized with the unit value and returns its location, which is selected nondeterministically. HeadLoad and HeadStore perform loads and stores, respectively. HeadCASSucc and HeadCASFail performs an atomic compare-and-swap at an offset in an array. HeadAssert reduces an assert statement to a unit if the asserted value is true; asserts of false are stuck expressions. HeadFork performs a fork, converting a primitive par operation into an active parallel tuple. HeadJoin takes an active parallel tuple where both sides have reached a value and converts it into an immutable product.

Figure 5 presents the main reduction relation $e \setminus \sigma \longrightarrow e' \setminus \sigma'$, describing a parallel step of computation, potentially under an evaluation context. StepHead performs a head step. StepCtx performs a computation step under an evaluation context. StepParL and StepParR implement parallelism: these two rules allow for the main reduction relation to perform nondeterministically a step to the left or right side of an active parallel tuple, respectively.

We write the reflexive-transitive closure of the reduction relation as $e \setminus \sigma \longrightarrow^* e' \setminus \sigma'$.

$$\textsc{RedHead}\ \frac{e \setminus \sigma \xrightarrow{\text{head}} e' \setminus \sigma'}{\mathsf{Red}\, e\, \sigma} \qquad \textsc{RedCtx}\ \frac{\mathsf{Red}\, e\, \sigma}{\mathsf{Red}\, (K\langle e\rangle)\, \sigma} \qquad \textsc{RedPar}\ \frac{e_1 \notin \mathcal{V} \implies \mathsf{Red}\, e_1\, \sigma \qquad e_2 \notin \mathcal{V} \implies \mathsf{Red}\, e_2\, \sigma}{\mathsf{Red}\, (e_1 \,||\, e_2)\, \sigma}\ e_1 \notin \mathcal{V} \vee e_2 \notin \mathcal{V}$$

$$\begin{aligned} \mathsf{Notstuck}\, e\, \sigma &\triangleq e \in \mathcal{V} \vee \mathsf{Red}\, e\, \sigma \\ \mathsf{Safe}\, e &\triangleq \forall e'\, \sigma'.\ (e \setminus \emptyset \longrightarrow^* e' \setminus \sigma') \implies \mathsf{Notstuck}\, e'\, \sigma' \\ \mathsf{SISafety}\ e &\triangleq \forall v\, \sigma.\ (e \setminus \emptyset \longrightarrow^* v \setminus \sigma) \implies \mathsf{Safe}\, e \end{aligned}$$

Fig. 6. Definition of the Red, Notstuck, Safe, and SISafety Predicates

4 A Separation Logic for Proving Schedule-Independent Safety

In this section, we present Musketeer in more detail. First, we define schedule-independent safety (§4.1). Next, we introduce our notations for triples and assertions (§4.2) and then present the reasoning rules of Musketeer (§4.3). We conclude with one of the main technical challenges in working with Musketeer, the absence of a rule for eliminating existentials, and explain how we overcame this with the novel concept of *ghost return values* (§4.4).

4.1 Definition of Schedule-Independent Safety

Let us make formal the definition of *schedule-independent safety*, that is, the property guaranteeing our motto "if one execution of e is safe and terminates, then every execution of e is safe".

What does it mean for a parallel program to be safe? We say that the configuration $e \setminus \sigma$ is *not stuck* if either e is a value, or every parallel task in e that has not reached a value can take a step—in the latter case, we call the configuration *reducible*. A program is defined to be safe if every configuration it can reach is not stuck. In particular, if a program e is safe, then no assertion in e can fail, since an assert of a false value is not reducible.

Figure 6 gives the formal definitions. The upper part of Figure 6 defines the property $\mathsf{Red}\, e\, \sigma$, asserting that the configuration $e \setminus \sigma$ is reducible. RedHead asserts that if e can take a head step, then it is reducible. RedCtx asserts that the reducibility of an expression $K\langle e\rangle$ follows from reducibility of e. RedPar asserts that an active parallel tuple $e_1 \,||\, e_2$ is reducible if at least one sub-expression is not a value (otherwise, a join is possible), and each sub-expression that is not a value is reducible. The lower part of Figure 6 asserts that the property $\mathsf{Notstuck}\, e\, \sigma$ holds if and only if either e is a value or $\mathsf{Red}\, e\, \sigma$ holds. Then, $\mathsf{Safe}\, e$ says that if $e \setminus \emptyset$ can reach $e' \setminus \sigma'$ in zero or more steps, then $\mathsf{Notstuck}\, e'\, \sigma'$. Finally, the main property $\mathsf{SISafety}\ e$, asserting that the safety of e is schedule-independent, is defined. The property says that if some execution of e reaches a value v, then e is safe. The soundness Theorem 4.1 of Musketeer guarantees that, for a verified program e, the property $\mathsf{SISafety}\ e$ holds.

4.2 Triples and Assertions

As we saw, Musketeer is a separation logic whose main judgement takes the form of a triple $\{P\}\ e\ \{Q\}$. In this triple, P is the *precondition*, e the program being verified, and Q the *postcondition*. The postcondition is of the form $\lambda v\, x.\, P'$, where v is the value being returned by the execution of e and x is a *ghost return value* returned by the verification of e. Both P and P' are separation logic assertions, and can be understood as *heap predicates*: they describe the content of a heap. We write $P * P'$ for the separating conjunction, $P \mathrel{-\!\!*} P'$ for the separating implication and $\ulcorner P \urcorner$ when the property P holds in the meta-logic (*i.e.* Rocq). Musketeer offers fractional [Bornat et al. 2005; Boyland 2003] points-to assertions $\ell \mapsto_q \vec{v}$. This assertion says that the location ℓ points to the array $\vec{v}$ with fraction $q \in (0; 1]$. When $q = 1$ we simply write $\ell \mapsto \vec{v}$. We use the term *vProp* for the type of assertions that can be used in Musketeer pre/post-conditions.

$$\textsc{M-If}\ \frac{(v = \text{true} \implies \{P\}\ e_1\ \{Q\}) \quad (v = \text{false} \implies \{P\}\ e_2\ \{Q\})}{\{P\}\ \text{if } v \text{ then } e_1 \text{ else } e_2\ \{Q\}} \qquad \textsc{M-Conseq}\ \frac{P \twoheadrightarrow P' \quad \{P'\}\ e\ \{Q'\} \quad \forall v\, x.\ Q\, v\, x \twoheadrightarrow Q'\, v\, x}{\{P\}\ e\ \{Q\}} \qquad \textsc{M-Val}\ \frac{P \twoheadrightarrow Q\, v\, x}{\{P\}\ v\ \{Q\}}$$

$$\textsc{M-Alloc}\ \{\top\}\ \text{alloc}\ w\ \{\lambda v\, (\ell, i).\ \ulcorner v = \ell \wedge w = i \wedge 0 \le i \urcorner * \ell \mapsto ()^i\}$$

$$\textsc{M-Load}\ \{\ell \mapsto_q \vec{v}\}\ \ell[w]\ \{\lambda v'\, i.\ \ulcorner w = i \wedge 0 \le i < |\vec{v}| \wedge \vec{v}(i) = v' \urcorner * \ell \mapsto_q \vec{v}\}$$

$$\textsc{M-Store}\ \{\ell \mapsto \vec{v}\}\ \ell[w] \leftarrow v'\ \{\lambda v''\, i.\ \ulcorner v'' = () \wedge w = i \wedge 0 \le i < |\vec{v}| \urcorner * \ell \mapsto [i := v']\vec{v}\}$$

$$\textsc{M-Bind}\ \frac{\{P\}\ e\ \{\lambda v\, x.\ Q'\, v\, x\} \quad \forall v\, x.\ \{Q'\, v\, x\}\ K\langle v\rangle\ \{Q\}}{\{P\}\ K\langle e\rangle\ \{Q\}} \qquad \textsc{M-Frame}\ \frac{\{P\}\ e\ \{Q\}}{\{P * P'\}\ e\ \{\lambda v\, x.\ Q\, v\, x * P'\}}$$

Fig. 7. Selected Reasoning Rules of Musketeer (extends Figure 1)

As described before, the Musketeer triple $\{P\}\ e\ \{Q\}$ can be intuitively read as implying the following hyper-property: "if one execution of e is safe starting from a heap satisfying P and terminates, then every execution of e is safe starting from a heap satisfying P and all terminating executions will end in a heap satisfying Q". If P and Q are trivial, then this implies the SISafety property. This is captured formally in the soundness theorem of the logic.

Theorem 4.1 (Soundness of Musketeer). *If* $\{\top\}\ e\ \{\lambda\_\_.\ \top\}$ *holds, then SISafety e holds.*

At first, this soundness theorem might seem weak, since it focuses on *safety* of all executions. What if we instead want to show that every terminating execution satisfies a stronger postcondition Q? In general, Theorem 4.1 does not directly imply such a stronger property, but recall that in MusketLang, safety implies that no assert fails. Thus, by annotating a program with appropriate assert statements, we can encode various specifications in terms of safety. We illustrated this aspect in our dumas example (§2.1), where safety implied that the return value across all executions would equal a particular number.

Although Musketeer is a unary logic with judgements referring to a single program e, the above statement reveals that the judgements are relating together multiple executions of that program. To make this work, under the hood, Musketeer's *vProp* assertions describe not one but *two* heaps, corresponding to two executions of the program. This has ramifications for some proof rules (§4.4). Later, we will see how *vProp* assertions can be encoded into assertions in a relational logic that makes these two different heaps more explicit.

4.3 Reasoning Rules for Musketeer

Figure 7 presents selected reasoning rules of Musketeer. Recall that because Musketeer triples do not imply safety, these rules differ from familiar separation logic rules. We have previously seen this in the rule M-Assert. A similar phenomenon happens in M-If, which targets the expression if v then e_1 else e_2. In standard separation logic, one must prove that v is a Boolean, since otherwise the if-statement would get stuck. However, in M-If, the user does not have to prove that v is a Boolean. Instead, the rule requires the user to verify the two sides of the if-statement under the hypothesis that v was the Boolean associated with the branch.

M-Alloc, M-Load and M-Store are similar to their standard separation logic counterparts, except that they do not require the user to show that the allocation size or the loaded or stored offset are valid integers. M-Alloc targets the expression alloc w and has a trivial pre-condition. The postcondition asserts that the value being returned is a location ℓ and that w is a non-negative integer–recall that we can think of the ghost return value (ℓ, i) as if it were just a special way of

existentially quantifying the variables ℓ and i in the postcondition. The postcondition additionally contains the points-to assertion $\ell \mapsto ()^i$ asserting that ℓ points to the array of size i initialized with the unit value. M-Load and M-Store follow the same pattern.

M-Alloc might surprise the reader, since based on the interpretation of triples we described above, the postcondition seems to imply that *every* execution of the allocation will return the same location ℓ. Yet allocation in MusketLang is *not* deterministic. The resolution of this seeming contradiction, is that because MusketLang does not allow for "constructing" a location (*e.g.* transforming an integer into a location), there is no way for the program to observe the nondeterminism of allocations. Hence, from the reasoning point-of-view we can conduct the proof *as if* allocations were made deterministically. This subtlety will appear in the model of Musketeer (§5.2).

M-Bind allows for reasoning under a context, and is very similar to the standard separation logic Bind rule, except that in the second premise, we quantify over not just the possible return values v, but also the ghost return value x. M-Val allows for concluding a proof about a value, allowing the user of the rule to pick an arbitrary ghost return value x. M-Frame shows that Musketeer supports framing. M-Conseq is the consequence rule of Musketeer: it allows for weakening the precondition and strengthening the postcondition.

4.4 Existential Reasoning with Ghost Return Values

In separation logic, existential quantification is essential for modularity. Among other things, it allows for concealing intermediate pointers behind an abstraction barrier. To see how this is typically done, let us consider an example making use of the following indirection function that creates a reference to a reference:

$$\mathsf{indirection} \triangleq \lambda v.\, \mathsf{ref}\,(\mathsf{ref}\, v)$$

Without using ghost return value, a possible specification for $\mathsf{indirection}\, v$ would be:

$$\{\top\}\ \mathsf{indirection}\, v\ \{\lambda w\ \_.\ \exists \ell.\ulcorner w = \ell\urcorner * \exists \ell'.\, \ell \mapsto [\ell'] * \ell' \mapsto [v]\}$$

In the above specification, the first existential quantification on ℓ does not hide or abstract over anything, since the returned value w uniquely characterizes ℓ. However, the existential quantification on ℓ' is more interesting, as it forms an abstraction barrier: it hides this intermediate location. Let us now focus on a client of indirection, and try to verify the following triple:

$$\{\top\}\ \mathsf{get}\,(\mathsf{indirection}\, v)\ \{\lambda\_\_.\ \top\}$$

Making use of M-Bind and then applying the above specification for indirection, we obtain:

$$\{\exists \ell.\ulcorner w = \ell\urcorner * \exists \ell'.\, \ell \mapsto [\ell'] * \ell' \mapsto [v]\}\ \mathsf{get}\, w\ \{\lambda\_\_.\ \top\} \qquad (\textsc{intermediate})$$

We now need to *eliminate* the existentials on ℓ and ℓ' in the precondition by introducing universally-quantified variables in the meta-logic. More precisely, we would like to apply the following standard separation logic rule:

$$\frac{\forall x.\ \{P\, x\}\ e\ \{Q\}}{\{\exists x.\, P\, x\}\ e\ \{Q\}}$$

However, Musketeer *does not support this rule*. Indeed, although Musketeer is formulated as a unary logic, it relates two executions of the same program. As we previously alluded to (§4.2), Musketeer's *vProp* assertions are, under the hood, tracking not one, but two heaps: one for each execution of the same program. The fact that preconditions describe two heaps implies that the precondition $\exists x.\, P\, x$ has two interpretations—one for each heap of the two executions of e being tracked by the triple. Although the precondition holds in both heaps, the witness x might differ between the two.

Whereas, in the premise of the above rule, quantifying over x at the meta-level means that x is treated as the same in both executions. We present a detailed example of such a case in Appendix A.

As a result, Musketeer only supports the weaker rule M-ElimExist, allowing an existential to be eliminated when the precondition guarantees that the witness is unique.

$$\text{M-ElimExist}\ \frac{(\forall x.\, P\, x \twoheadrightarrow \ulcorner U\, x \urcorner) \qquad (\forall x\, y.\, U\, x \land U\, y \implies x = y) \qquad (\forall x.\, \{P\, x\}\ e\ \{Q\})}{\{\exists x.\, P\, x\}\ e\ \{Q\}}$$

For example, in the above INTERMEDIATE triple, M-ElimExist would allow to eliminate the quantification on ℓ, since it is uniquely characterized by w. However, M-ElimExist is tedious to use in practice. Moreover, sometimes objects are *not* uniquely characterized by the precondition, and yet are chosen deterministically, so that the witnesses ought to be the same in both executions. For example, in the above INTERMEDIATE triple, M-ElimExist cannot be used to eliminate the quantification on ℓ'.

To solve this issue, we introduce ghost return values. In a Musketeer triple $\{P\}\ e\ \{\lambda v\, x.\, Q\, v\, x\}$, the ghost return value x is an object (of an arbitrary type, which is formally a parameter of the triple) that will eventually be chosen by the user when they apply M-Val. We think of the bound variable x as if it were existentially quantified, but the key is that the eventual "witness" selected when using M-Val will be the same across the two executions under consideration. As a result, instead of having to use the weak M-ElimExist to eliminate x, the ghost return value is *automatically* eliminated in a strong way by M-Bind.

Let us go back to our indirection example. We prove a specification for indirection in which ℓ' is bound in a ghost return value, instead of as an existential:

$$\{\top\}\ \mathsf{indirection}\, v\ \{\lambda v\, \ell'.\ \exists \ell.\, \ulcorner v = \ell \urcorner\ *\ \ell \mapsto [\ell']\ *\ \ell' \mapsto [v]\}$$

As we use this specification to reason about $(\mathsf{get}\ \ (\mathsf{indirection}\, v))$, M-Bind will eliminate ℓ' automatically, and we can use M-ElimExist to eliminate ℓ, reducing the proof to:

$$\{\ell \mapsto [\ell']\ *\ \ell' \mapsto [v]\}\ \mathsf{get}\, \ell\ \{\lambda \_\, \_.\ \top\}$$

allowing us to proceed and conclude, since there is no longer an existential to eliminate.

We extensively use ghost return values for the verification of MiniDet, our case study (§7). For instance, we use a ghost return value to record the content of references in the typing environment.

5 Unchaining the Reasoning with Chained Triples

For an expression e, a Musketeer triple guarantees the property "if one execution of e is safe and terminates, then every execution of e is safe". In order to justify the validity of the reasoning rules for Musketeer triples, we generalize the above property and define an intermediate logic called ChainedLog which targets two expressions e_l and e_r and guarantees the property "if one execution of e_l is safe and terminates, then every execution of e_r is safe". We first present chained triples (§5.1) and present some associated reasoning rules (§5.2). Finally, we explain how we encode Musketeer triples using chained triples (§5.3).

5.1 Chained Triples as a Generalization of Musketeer Triples

In ChainedLog, a chained triple takes the form:

$$\{\varphi_l\}\ e_l\ \{\psi_l \mid \varphi_r\}\ e_r\ \{\psi_r\}$$

The assertions φ_l and φ_r are the preconditions of e_l and e_r, respectively. The assertions ψ_l and ψ_r are both of the form $\lambda v.\, \varphi$, where v is a return value, and are the postconditions of e_l and e_r, respectively. Intuitively, the above chained triple says that, if there exists a reduction of e_l starting from a heap

satisfying φ_l, that is safe and terminates on a final heap with a value v_l satisfying $\psi_l\, v_l$, then every reduction of e_r starting from a heap satisfying φ_r is safe and if it terminates, it does so on a final heap with a value v_r satisfying $\psi_r\, v_r$. Moreover, chained triples guarantee determinism (for simplicity, see our commentary of C-Par), that is, $v_l = v_r$. Formally, we have the following soundness theorem:

Theorem 5.1 (Soundness of Chained Triples). *If* $\{\top\}\ e_l\ \{\_.\ \top \mid \top\}\ e_r\ \{\lambda\_.\ \top\}$ *holds, and if there exists a value* v *and a store* σ *such that* $e_l \setminus \emptyset \longrightarrow^* v \setminus \sigma$*, then the property Safe* e_r *holds.*

In particular, chained triples do not guarantee safety for e_l, but they do guarantee safety for e_r. We call the triples "chained" because enjoy the following rule that allows us to chain facts from one execution to the other:

$$\textsc{C-Chain}\ \frac{\{\varphi_l\}\ e_l\ \{\lambda v_l.\, \psi_l\, v_l * \varphi \mid \varphi_r\}\ e_r\ \{\lambda v_r.\, \psi_r\}}{\{\varphi_l\}\ e_l\ \{\lambda v_l.\, \psi_l\, v_l \mid \varphi \mathrel{-\!\!*} \varphi_r\}\ e_r\ \{\lambda v_r.\, \psi_r\}}$$

It is best to read this rule from the bottom up. Below the line, using the precondition for e_r requires showing φ. Above the line, the rule allows us to discharge this assumption by showing that φ holds in the postcondition of e_l. That is, if some knowledge φ is needed in order to verify the safety of e_r, then this knowledge can be gained from an execution of e_l.

Assertions φ of ChainedLog are ground Iris assertions of type *iProp*. As previously intuited (§4.2), they include two forms of points-to assertions, one for each side of the triple. We write $\ell \mapsto^{\boldsymbol{l}}_q \vec{v}$ the points-to assertion for the left expression, and $\ell \mapsto^{\boldsymbol{r}}_q \vec{v}$ for the right expression. Moreover, ChainedLog makes use of a *left-allocation token*, written leftalloc ℓ. This (non-persistent) assertion witnesses that ℓ has been allocated by the left expression and plays a key role for allocations.

5.2 Reasoning Rules for Chained Triples

Figure 8 presents selected reasoning rules for chained triples. Before commenting on these rules, let us underline a caveat of chained triples, explaining in part why we only use them as a model for Musketeer: chained triples do *not* support a Bind rule.[3] Hence, non-structural rules for chained triples explicitly mentions a stack of contexts, written $\vec{K}$.

Let us again start with the rules for reasoning about an assertion. C-AssertL allows for reasoning about assert v on the left-hand side. Because this rule targets the left hand-side, there is no safety-related proof obligation, hence the premise of the rule allows the user to suppose that v = true. C-AssertR is, on the contrary, similar to a standard separation logic rule for assertions: the assertion must target a Boolean, and this Boolean must be true.

C-AllocL allows for reasoning about an allocation of an array on the left-hand side. Again, there is no safety proof obligation, so the user gets to suppose that the argument of the allocation is a non-negative integer. The precondition is then augmented with a points-to assertion to a universally quantified location ℓ as well as the allocation token leftalloc ℓ. This latter assertion plays a role in C-AllocR, which allows for reasoning about an allocation on the right-hand side. Indeed, the assertion leftalloc ℓ appears in the precondition of the right-hand side. This assertion allows for *predicting* the location allocated on the right-hand side. As a result, the premise of C-AllocR does not universally quantify over the location allocated–the name ℓ is reused. The user can transmit a leftalloc ℓ assertion from the left-hand side to the right-hand side using C-FrameL and C-Chain.

This rule may seem surprising, since allocation is nondeterministic in MusketLang, yet this rule appears to ensure that the right-hand side allocation returns the *same* location as the left-hand side. The key is that a right-hand points-to assertion of the form $\ell \mapsto^{\boldsymbol{r}}_q \vec{v}$ does *not* mean that the

[3]The absence of a Bind rule comes from the chaining intention of these triples: the user needs to terminate the reasoning on the whole left-hand side expression before reasoning on the right-hand side.

$$\textsc{C-AssertL}\ \frac{v = \mathsf{true} \implies \{\varphi_l\}\ \vec{K}_l\langle()\rangle\ \{\psi_l \mid \varphi_r\}\ \vec{K}_r\langle e_r\rangle\ \{\psi_r\}}{\{\varphi_l\}\ \vec{K}_l\langle \mathsf{assert}\, v\rangle\ \{\psi_l \mid \varphi_r\}\ \vec{K}_r\langle e_r\rangle\ \{\psi_r\}} \qquad \textsc{C-AssertR}\ \frac{\{\varphi_l\}\ \vec{K}_l\langle e_l\rangle\ \{\psi_l \mid \varphi_r\}\ \vec{K}_r\langle()\rangle\ \{\psi_r\}}{\{\varphi_l\}\ \vec{K}_l\langle e_l\rangle\ \{\psi_l \mid \varphi_r\}\ \vec{K}_r\langle \mathsf{assert\ true}\rangle\ \{\psi_r\}}$$

$$\textsc{C-AllocL}\ \frac{\forall \ell\ i.\ v = i \wedge 0 \le i \implies \{\varphi_l * \ell \mapsto^{\boldsymbol{l}} ()^i * \mathsf{leftalloc}\,\ell\}\ \vec{K}_l\langle \ell\rangle\ \{\psi_r \mid \varphi_r\}\ \vec{K}_r\langle e_r\rangle\ \{\psi_r\}}{\{\varphi_l\}\ \vec{K}_l\langle \mathsf{alloc}\, v\rangle\ \{\psi_r \mid \varphi_r\}\ \vec{K}_r\langle e_r\rangle\ \{\psi_r\}}$$

$$\textsc{C-AllocR}\ \frac{0 \le i \qquad \{\varphi_l\}\ \vec{K}_l\langle e_l\rangle\ \{\psi_l \mid \varphi_r * \ell \mapsto^{\boldsymbol{r}} ()^i\}\ \vec{K}_r\langle \ell\rangle\ \{\psi_r\}}{\{\varphi_l\}\ \vec{K}_l\langle e_l\rangle\ \{\psi_l \mid \varphi_r * \mathsf{leftalloc}\,\ell\}\ \vec{K}_r\langle \mathsf{alloc}\, i\rangle\ \{\psi_r\}}$$

$$\textsc{C-LoadL}\ \frac{\forall i\ w.\ v = i \wedge 0 < i \le |\vec{v}| \wedge w = \vec{v}(i) \implies \{\varphi_l * \ell \mapsto^{\boldsymbol{l}}_q \vec{v}\}\ \vec{K}_l\langle \vec{w}\rangle\ \{\psi_r \mid \varphi_r\}\ \vec{K}_r\langle e_r\rangle\ \{\psi_r\}}{\{\varphi_l * \ell \mapsto^{\boldsymbol{l}}_q \vec{v}\}\ \vec{K}_l\langle \ell[v]\rangle\ \{\psi_r \mid \varphi_r\}\ \vec{K}_r\langle e_r\rangle\ \{\psi_r\}}$$

$$\textsc{C-LoadR}\ \frac{0 < i \le |\vec{v}| \wedge w = \vec{v}(i) \qquad \{\varphi_l\}\ \vec{K}_l\langle e_l\rangle\ \{\psi_l \mid \varphi_r * \ell \mapsto^{\boldsymbol{r}}_q \vec{v}\}\ \vec{K}_r\langle w\rangle\ \{\psi_r\}}{\{\varphi_l\}\ \vec{K}_l\langle e_l\rangle\ \{\psi_l \mid \varphi_r * \ell \mapsto^{\boldsymbol{r}}_q \vec{v}\}\ \vec{K}_r\langle \ell[i]\rangle\ \{\psi_r\}}$$

$$\textsc{C-Par}\ \frac{\{\varphi_{l1}\}\ e_{l1}\ \{\psi_{l1} \mid \varphi_{r1}\}\ e_{r1}\ \{\psi_{r1}\} \qquad \{\varphi_{l2}\}\ e_{l2}\ \{\psi_{l2} \mid \varphi_{r2}\}\ e_{r2}\ \{\psi_{r2}\} \qquad \forall v_1\ x_1\ v_2\ x_2.\ \{\psi_{l1} v_1\ x_1 * \psi_{l2}\, v_2\ x_2\}\ \vec{K}_l\langle (v_1, v_2)\rangle\ \{\psi_l \mid \psi_{r1} v_1\ x_1 * \psi_{r2}\, v_2\ x_2\}\ \vec{K}_r\langle (v_1, v_2)\rangle\ \{\psi_r\}}{\{\varphi_{l1} * \varphi_{l2}\}\ \vec{K}_l\langle \mathsf{par}\, e_{l1}\, e_{l2}\rangle\ \{\psi_l \mid \varphi_{r1} * \varphi_{r2}\}\ \vec{K}_r\langle \mathsf{par}\, e_{r1}\, e_{r2}\rangle\ \{\psi_r\}}$$

$$\textsc{C-FrameL}\ \frac{\{\varphi_l\}\ e_l\ \{\psi_l \mid \varphi_r\}\ e_r\ \{\psi_r\}}{\{\varphi_l * \varphi_0\}\ e_l\ \{\psi_l * \varphi_0 \mid \varphi_r\}\ e_r\ \{\psi_r\}} \qquad \textsc{C-FrameR}\ \frac{\{\varphi_l\}\ e_l\ \{\psi_l \mid \varphi_r\}\ e_r\ \{\psi_r\}}{\{\varphi_l\}\ e_l\ \{\psi_l \mid \varphi_r * \varphi_0\}\ e_r\ \{\lambda v_r.\, \psi_r\, v_r * \varphi_0\}}$$

Fig. 8. Selected Reasoning Rules for Chained Triples

specific location ℓ has that value in the right-hand side execution. Rather, it means that there exists some location which points to $\vec{v}$ on the right-hand side, and we can reason *as if* that location were equivalent to ℓ, under some implicit permutation renaming of locations. In other words, as we alluded to earlier in Section 4.3 when discussing the nondeterminism of allocation in Musketeer, the logic ensures that the specific location of an allocation does not matter, since we do not support casting integers to pointers.

Our approach of using the $\mathsf{leftalloc}\,\ell$ assertion has two consequences. First, as we will see (§5.3), it will allow us to define Musketeer triples in terms of chained triples where both the left- and right-hand side coincide; such a definition would be impossible if the allocation on the left and on the right-hand side could return different names. Second, it bounds the number of allocations on the right-hand side by the number of allocations on the left-hand side. We posit that this limitation can be lifted by distinguishing between synchronized locations, whose name come from the left-hand side, and unsynchronized one. We were able to conduct our case studies without such a feature.

C-LoadL and C-LoadR follow the same spirit as the previous rules: the rule for the left-hand side has no safety proof obligation, but the right-hand size has a standard separation logic shape.

C-Par targets a parallel primitive and is a *synchronization point*: both the left- and right-hand side must face a parallel primitive. The rule mimics a standard Par rule on both sides at once. In particular, it requires the user to split the preconditions of the left- and right-hand sides, which will be given to the corresponding side of the active parallel pair. The bottom premise of C-Par requires the user to verify the continuation, after the execution of the parallel primitive ended. This premise also show the (external) determinism guaranteed by chained triple: the execution is resumed on both sides with the same result of the parallel execution: the immutable pair (v_1, v_2). Note also that both sides agree on the ghost return values.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{vProp} & \triangleq & \mathbb{B} \to \mathit{iProp} & \forall x.\, P\, x & \triangleq & \lambda b.\, \forall x.\, P\, x\, b \\
P_1 * P_2 & \triangleq & \lambda b.\, P_1\, b * P_2\, b & \exists x.\, P\, x & \triangleq & \lambda b.\, \exists x.\, P\, x\, b \\
P_1 \mathrel{-\!\!*} P_2 & \triangleq & \lambda b.\, P_1\, b \mathrel{-\!\!*} P_2\, b & \ell \mapsto_q \vec{v} & \triangleq & \lambda b.\, \mathsf{if}\ b\ \mathsf{then}\ \ell \mapsto^{\boldsymbol{l}}_q \vec{v}\ \mathsf{else}\ \ell \mapsto^{\boldsymbol{r}}_q \vec{v}
\end{array}
$$

Fig. 9. Definition of *vProp* assertions

$$
\begin{array}{l}
\{P\}\ e\ \{Q\} \triangleq \forall \vec{K}\, \varphi_l\, \varphi_r\, \psi_l\, \psi_r. \\
\quad \left(\forall v\, x.\, \{Q\, v\, x\, \mathsf{true} * \varphi_l\}\ \vec{K}\langle v\rangle\ \{\psi_l \mid Q\, v\, x\, \mathsf{false} * \varphi_r\}\ \vec{K}\langle v\rangle\ \{\psi_r\}\right) \mathrel{-\!\!*} \\
\quad \{P\, \mathsf{true} * \varphi_l\}\ \vec{K}\langle e\rangle\ \{\psi_l \mid P\, \mathsf{false} * \varphi_r\}\ \vec{K}\langle e\rangle\ \{\psi_r\}
\end{array}
$$

Fig. 10. Definition of Musketeer Triples

5.3 Encoding Musketeer in ChainedLog

We now discuss how to encode Musketeer into ChainedLog. Recall that Musketeer's assertions have the type *vProp*. We encode these as functions from Booleans to *iProp*, the ground type of ChainedLog assertions. The idea is that the *vProp* tracks two heaps, and we use the Boolean parameter of the function to indicate which side of the ChainedLog the assertion is being interpreted to: true indicates the left side, and false the right side. The formal definition of *vProp* assertions appears in Figure 9. The Boolean parameter is threaded through the separating star and implication, and similarly for the $\forall$ and $\exists$ quantifier. The points-to assertion simply cases over the Boolean and returns the left or right version of the points-to. Entailment is defined as $P_1 \vdash P_2 \triangleq \forall b.\, P_1\, b \vdash P_2\, b$.

Next, we can encode Musketeer triples as shown in Figure 10. A Musketeer triple $\{P\}\ e\ \{Q\}$ is mapped to a chained triple where both sides refer to the expression e use the precondition P instantiated with Booleans corresponding to the appropriate side. Because chained triples do not support a bind rule, the encoding is written in a continuation passing style: rather than having Q in the post-condition of the chained triple, we instead quantify over an evaluation context $\vec{K}$ that represents an arbitrary continuation to run after e. This continuation is assumed to satisfy a chained tripled in which Q occurs in the preconditions. We additionally quantify over several assertions φ_l, φ_r, ψ_l, and ψ_r that are used to represent additional resources used by the continuation.

6 A Separation Logic for Verifying One Interleaving

We now return to Angelic, our program logic verifying that *one* interleaving of a MusketLang program is safe and terminates. We first present the assertions of Angelic (§6.1) and then present selected reasoning rules (§6.2).

6.1 Assertions of Angelic

Assertions of Angelic are Iris assertions of type *iProp*, written φ. The fractional points-to assertion of Angelic takes the form $\ell \mapsto_q \vec{v}$ (while we reuse the syntax of the points-to assertion from Musketeer, the two assertions are different—recall that Angelic and Musketeer are totally disjoint).

A key aspect of Angelic is that this logic has two reasoning modes. First, the *running mode* takes the form $\mathsf{run}\, e\, \{\psi\}$, where e is the expression being logically "run" and ψ is a postcondition, The assertion $\mathsf{run}\, e\, \{\psi\}$ is close to a weakest-precondition (WP). In fact, it enjoys all the rules of a standard separation logic WP. However, the running mode enjoys additional rules that allow one to dynamically "select" and verify just one interleaving. This selection is made possible thanks to a second mode, that we call the *scheduler mode*. The scheduler mode involves two key assertions. First, $\mathfrak{goal}$ is an opaque assertion, intuitively representing the proof obligation to verify the whole program. Second, the assertion $\mathsf{yielded}\, \pi\, e$ asserts the ownership of the task π, and that this task yielded facing expression e.

$$\textsc{A-Alloc}\ \frac{\ulcorner 0 \le i \urcorner}{\mathsf{run}\ (\mathsf{alloc}\ i)\ \{\lambda v.\ \exists \ell.\ \ell \mapsto ()^i\}} \qquad \textsc{A-Load}\ \frac{\ulcorner 0 \le i < |\vec{v}| \urcorner \qquad \ell \mapsto_q \vec{v}}{\mathsf{run}\ (\ell[i])\ \{w.\ \ulcorner w = \vec{v}(i) \urcorner * \ell \mapsto_q \vec{v}\}} \qquad \textsc{A-Bind}\ \frac{\mathsf{run}\ e\ \{\lambda v.\ \mathsf{run}\ (K\langle v\rangle)\ \{\psi\}\}}{\mathsf{run}\ (K\langle e\rangle)\ \{\psi\}}$$

$$\textsc{A-Store}\ \frac{\ulcorner 0 \le i < |\vec{v}| \urcorner \qquad \ell \mapsto_{\vec{v}}}{\mathsf{run}\ (\ell[i] \leftarrow v')\ \{w.\ \ulcorner w = () \urcorner * \ell \mapsto [i := v']\vec{v}\}} \qquad \textsc{A-Call}\ \frac{\mathsf{run}\ ([\hat{\mu} f\ x.\ e/f][v/x]e)\ \{\psi\}}{\mathsf{run}\ (\hat{\mu} f\ x.\ e)\ v\ \{\psi\}} \qquad \textsc{A-Conseq}\ \frac{\mathsf{run}\ e\ \{\psi'\} \qquad (\forall v.\ \psi'\ v \mathrel{-\!\!*} \psi\ v)}{\mathsf{run}\ e\ \{\psi\}}$$

$$\textsc{Yield}\ \frac{\forall \pi.\ \mathsf{yielded}\ \pi\ e \mathrel{-\!\!*} (\forall v.\ \mathsf{yielded}\ \pi\ v \mathrel{-\!\!*} \psi\ v \mathrel{-\!\!*} \mathfrak{goal}) \mathrel{-\!\!*} \mathfrak{goal}}{\mathsf{run}\ e\ \{\psi\}} \qquad \textsc{Resume}\ \frac{\mathsf{yielded}\ \pi\ e \qquad \mathsf{run}\ e\ \{\lambda v.\ \mathsf{yielded}\ \pi\ v \mathrel{-\!\!*} \mathfrak{goal}\}}{\mathfrak{goal}}$$

$$\textsc{Fork}\ \frac{\forall \pi_1\ \pi_2.\ \mathsf{yielded}\ \pi_1\ e_1 \mathrel{-\!\!*} \mathsf{yielded}\ \pi_2\ e_2 \mathrel{-\!\!*} (\forall v_1\ v_2.\ \mathsf{yielded}\ \pi_1\ v_1 \mathrel{-\!\!*} \mathsf{yielded}\ \pi_2\ v_2 \mathrel{-\!\!*} \psi\ (v_1, v_2) \mathrel{-\!\!*} \mathfrak{goal}) \mathrel{-\!\!*} \mathfrak{goal}}{\mathsf{run}\ (\mathsf{par}\ e_1\ e_2)\ \{\psi\}}$$

Fig. 11. Selected Reasoning Rules of Angelic (extends Figure 2)

The logic satisfies the following soundness theorem:

Theorem 6.1 (Soundness of Angelic). *If* $\mathsf{run}\ e\ \{\lambda\_.\ \top\}$ *holds, then there exists a value* v *and a store* σ *such that* $e \setminus \emptyset \longrightarrow^* v \setminus \sigma$.

6.2 Reasoning Rules of Angelic

Figure 11 presents the key reasoning rules allowing the user to select and verify an interleaving. These inference rules are at the *iProp* level: their premises are implicitly separated by $*$, and the implication between the premise and the conclusion is stated as the entailement $\vdash$.

The upper part of Figure 11 showcases that the run mode of Angelic is, for its sequential part, similar to a standard separation logic. A-Alloc performs an allocation, A-Load a load and A-Store a store—here, the allocation size and various offsets must be valid. A-Call verifies a function call. A-Conseq shows that the user can make the postcondition stronger. A-Bind allows for reasoning under an evaluation context.

The lower part of Figure 11 focuses on the scheduler mode of Angelic. Yield asserts (reading the rule from bottom to top) that the proof of $\mathsf{run}\ e\ \{\psi\}$ can pause, and switch to the scheduler mode—that is, a proof where the target is $\mathfrak{goal}$. To prove this target, the user gets to assume that some (universally quantified) task π yielded with expression e, and that when this expression will have reduced to a value v satisfying ψ, then $\mathfrak{goal}$ will hold.

Resume is the companion rule of Yield: it asserts that in order to prove $\mathfrak{goal}$, the user has to give up the ownership of a task π facing an expression e and switch back to the running mode to verify that $\mathsf{run}\ e\ \{\lambda v.\ \mathsf{yielded}\ \pi\ v \mathrel{-\!\!*} \mathfrak{goal}\}$.

Fork shows the real benefit of the scheduler mode. This rule asserts that, for verifying the parallel primitive $\mathsf{par}\ e_1\ e_2$, the user can switch to the scheduler mode. In this mode, the user gets to suppose that two tasks π_1 and π_2 yielded at e_1 and e_2, respectively. Moreover, the user can suppose that, when these two tasks would have completed their execution and reached values v_1 and v_2 such that $\psi\ (v_1, v_2)$ hold, the $\mathfrak{goal}$ will hold. At this point, the user can choose which of e_1 and e_2 to begin verifying using Resume.

Recall in Section 2.3 we saw rules A-ParSeqL and A-ParSeqR allowing one to verify a parallel composition by picking either a left-then-right or right-then-left sequential ordering. These two rules can be derived from the more general constructs of Angelic that we have now seen. For

$$\tau \triangleq \bot \mid \mathsf{empty} \mid \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \tau \to \tau \mid (\tau \times \tau) \mid \mathsf{ref}\,\tau$$
$$\Gamma \in \mathsf{Var} \rightharpoonup \tau$$

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{empty} \cdot \mathsf{empty} & \triangleq & \mathsf{empty} & \mathsf{int} \cdot \mathsf{int} & \triangleq & \mathsf{int} \\ \mathsf{unit} \cdot \mathsf{unit} & \triangleq & \mathsf{unit} & (\tau_1 \times \tau_2) \cdot (\tau_1' \times \tau_2') & \triangleq & ((\tau_1 \cdot \tau_1') \times (\tau_2 \cdot \tau_2')) \\ \mathsf{bool} \cdot \mathsf{bool} & \triangleq & \mathsf{bool} & (\tau_1 \to \tau_2) \cdot (\tau_1' \to \tau_2') & \triangleq & \text{if } (\tau_1 = \tau_1' \wedge \tau_2 = \tau_2') \text{ then } \tau_1 \to \tau_2 \text{ else } \bot \end{array}$$

Fig. 12. Syntax of MiniDet Type System

example, in order to show that A-ParSeqL holds, we first apply Fork, then use Resume for the expression e_1. We then use A-Conseq with Resume for expression e_2 and conclude.

7 Case Studies

To showcase Musketeer, we start by using it to prove the soundness of a simple affine type system that ensures schedule-independent safety (§7.1). We then extend this type system with two core algorithmic primitives proposed by Blelloch et al. [2012] for ensuring internal determinacy: priority writes (§7.2) and deterministic hash sets (§7.3). Interestingly, while these primitives appear to be internally deterministic to their client, their implementation is not: they observe internal state of shared data structures that may be concurrently modified. Yet, we show that they satisfy schedule-independent safety. Because all well-typed programs in this system have schedule-independent safety, we can use Angelic to reason about them, as we demonstrate by verifying a parallel list deduplication example (§7.4).

7.1 MiniDet: An Affine Type System for Determinism

This section presents MiniDet, an affine type system for MusketLang that ensures determinism. Like many other substructural type systems, the types in MiniDet can be thought of as tracking *ownership* of resources such as array references, thereby preventing threads from accessing shared resources in a way that would introduce nondeterministic behaviors.

Syntax. The syntax of types in MiniDet appears in Figure 12. A type τ is either the invalid type $\bot$ (used only internally), the empty type, describing an unknown value without ownership, the unit type unit, the Boolean type bool, the integer type int, the arrow type $\tau_1 \to \tau_2$, the immutable product $(\tau_1 \times \tau_2)$ or the reference type $\mathsf{ref}\,\tau$. A typing environment Γ is a finite map from variables to types. We write $\mathsf{dom}(\Gamma)$ for its domain.

The type system is *affine* meaning that, when splitting a typing context in two, a variable can only appear in one sub-context at a time. However, variables with types whose inhabitants have no associated notion of ownership, or variables with types with fractional reasoning, can be split and joined. In order to capture this notion, we equip types with a monoid operation $\_ \cdot \_$ taking two types as arguments and producing a new type. In particular, when $\tau \cdot \tau = \tau$, it means that a variable of type τ can be duplicated. The definition of the monoid operation appears in the lower part of Figure 12. The missing cases are all sent to $\bot$. In particular these definitions prevent a reference from being duplicated. We extend the monoid operation to typing environments by defining $\Gamma_1 \cdot \Gamma_2$ as the function that maps the variable x to τ_1 if $\Gamma_1(x) = \tau_1$ and x is not in the domain of Γ_2, τ_2 if $\Gamma_2(x) = \tau_2$ and x is not in the domain of Γ_1, and $\tau_1 \cdot \tau_2$ if $\Gamma_1(x) = \tau_1$ and $\Gamma_2(x) = \tau_2$.

The typing judgement of MiniDet takes the form $\Gamma \vdash e : \tau \dashv \Gamma'$, and asserts that e has type τ and transforms the typing environment Γ into Γ'.

Typing rules. Selected typing rules appear in Figure 13. T-Var types variable x at type τ if x has type τ in the typing environment. The returned environment is empty. T-Unit, T-Bool and rule T-Int type unboxed values. T-Assert types an assert primitive. T-Let types a let-binding

$$\textsc{T-Var}\ \{x := \tau\} \vdash x : \tau \dashv \emptyset \qquad \textsc{T-Unit}\ \emptyset \vdash () : \mathsf{unit} \dashv \emptyset \qquad \textsc{T-Bool}\ \emptyset \vdash b : \mathsf{bool} \dashv \emptyset \qquad \textsc{T-Int}\ \emptyset \vdash i : \mathsf{int} \dashv \emptyset$$

$$\textsc{T-Assert}\ \frac{\Gamma \vdash e : \mathsf{bool} \dashv \Gamma'}{\Gamma \vdash \mathsf{assert}\, e : \mathsf{unit} \dashv \Gamma'} \qquad \textsc{T-Let}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \dashv \Gamma_1' \qquad [x := \tau_1]\Gamma_1' \vdash e_2 : \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2 \dashv \mathsf{del}\, x\, \Gamma_2} \qquad \textsc{T-Weak}\ \frac{\Gamma_1 \subseteq \Gamma_1' \qquad \Gamma_2 \subseteq \Gamma_2' \qquad \Gamma_1' \vdash e : \tau \dashv \Gamma_2'}{\Gamma_1 \vdash e : \tau \dashv \Gamma_2}$$

$$\textsc{T-Abs}\ \frac{\Gamma = \Gamma \cdot \Gamma \qquad [f := \tau \to \tau'][x := \tau]\Gamma \vdash e : \tau' \dashv \emptyset}{\Gamma \vdash \mu f\, x.\, e : \tau \to \tau' \dashv \emptyset} \qquad \textsc{T-App}\ \frac{\Gamma_1 \vdash e_1 : \tau \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : \tau \to \tau' \dashv \Gamma_3}{\Gamma_1 \vdash e_2\, e_1 : \tau' \dashv \Gamma_3} \qquad \textsc{T-Ref}\ \frac{\Gamma \vdash e : \tau \dashv \Gamma'}{\Gamma \vdash \mathsf{ref}\, e : \mathsf{ref}\, \tau \dashv \Gamma'}$$

$$\textsc{T-Get}\ \{x := \mathsf{ref}\, \tau\} \vdash \mathsf{get}\, x : \tau \dashv \{x := \mathsf{ref}\, \mathsf{empty}\} \qquad \textsc{T-Set}\ \frac{\Gamma \vdash e : \tau \dashv \{x := \mathsf{ref}\, \mathsf{empty}\} \cdot \Gamma'}{\Gamma \vdash \mathsf{set}\, x\, e : \mathsf{unit} \dashv \{x := \mathsf{ref}\, \tau\} \cdot \Gamma'}$$

$$\textsc{T-Par}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \dashv \Gamma_1' \qquad \Gamma_2 \vdash e_2 : \tau_2 \dashv \Gamma_2'}{\Gamma_1 \cdot \Gamma_2 \vdash \mathsf{par}\, e_1\, e_2 : (\tau_1 \times \tau_2) \dashv \Gamma_1' \cdot \Gamma_2'} \qquad \textsc{T-Frame}\ \frac{\Gamma \vdash e : \tau \dashv \Gamma'}{\Gamma_0 \cdot \Gamma \vdash e : \tau \dashv \Gamma_0 \cdot \Gamma'}$$

Fig. 13. Selected Typing Rules of MiniDet

$\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ at type τ_2 with initial context Γ_1 if e_1 has type τ_1 under the same context and produces context Γ_2, and if e_2 has type τ_2 under the context Γ_1 in which x has type τ_1. The produced context of the let-binding is Γ_2 from which x has been deleted. T-Abs types a function with recursive name f, argument x and body e, of type $\tau \to \tau'$ and with typing environment Γ. This environment must be duplicable, that is $\Gamma = \Gamma \cdot \Gamma$. This duplicability implies that Γ contains no types with ownership, that is, for now, no references. The precondition requires that e has type τ' in Γ, augmented with f of type $\tau \to \tau'$ and x of type τ. T-App types a function call and is straightforward. T-Ref types a reference allocation. T-Get types a get operation on a variable x. This rule requires that x is of some type $\mathsf{ref}\, \tau$, returns a type τ and updates the binding of x to $\mathsf{ref}\, \mathsf{empty}$. This is because get returns the ownership of the content of the cell—meaning that the cell does not hold recursive ownership of its contents anymore.[4] T-Set is the dual, and types the expression $\mathsf{set}\, x\, e$. This rule requires that e is of some type τ and that, in the resulting environment, x is of type $\mathsf{ref}\, \mathsf{empty}$. The set operation returns unit and updates the type of x to $\mathsf{ref}\, \tau$, "filling" the cell. T-Par types a parallel primitive, and is similar to the related separation logic rules. Indeed, T-Par requires splitting the context in two parts, that will be used to type separately the two sub-tasks, whose result typing context will be merged in the result typing context of the rule. Finally, T-Frame allows for framing a part of the context for local reasoning, and T-Weak allows for removing bindings from the input and output typing environments.

Soundness of MiniDet. The above system prevents data-races, and hence guarantees that well-typed programs have schedule-indepedent safety, as formalized by the following lemma.

Lemma 7.1 (Soundness of MiniDet). *If $\emptyset \vdash e : \tau \dashv \emptyset$ holds, then SISafety e holds.*

To prove this theorem, we use program logic-based *semantic typing* [Timany et al. 2024b]. With this technique, we associate a triple (in our case, a Musketeer triple) to a typing judgement, and

[4]We could have derived another rule for get on a reference whose content is not tied to any ownership. We follow this approach when extending the type system with priority writes (§7.2).

$$s \triangleq \mathsf{sinvalid} \mid \mathsf{snone} \mid \mathsf{sprod}\, s\, s \mid \mathsf{sref}\, v\, s \mid \mathsf{sarrow}\, \gamma \qquad M \in \mathsf{Var} \rightharpoonup \tau \qquad V \in \mathsf{Var} \rightharpoonup \mathcal{V}$$

$$\begin{aligned}
[\![\mathsf{empty} \mid \mathsf{snone} \mid v]\!] &\triangleq \top & [\![\mathsf{bool} \mid \mathsf{snone} \mid v]\!] &\triangleq \ulcorner \exists b.\, v = b \urcorner \\
[\![\mathsf{unit} \mid \mathsf{snone} \mid v]\!] &\triangleq \ulcorner v = () \urcorner & [\![\mathsf{int} \mid \mathsf{snone} \mid v]\!] &\triangleq \ulcorner \exists i.\, v = i \urcorner \\
[\![(\tau_1 \times \tau_2) \mid \mathsf{sprod}\, s_1\, s_2 \mid v]\!] &\triangleq \exists v_1\, v_2.\, \ulcorner v = (v_1, v_2) \urcorner * [\![\tau_1 \mid s_1 \mid v_1]\!] * [\![\tau_2 \mid s_2 \mid v_2]\!] \\
[\![\mathsf{ref}\, \tau \mid \mathsf{sref}\, w\, s \mid v]\!] &\triangleq \exists \ell.\, \ulcorner v = \ell \urcorner * \ell \mapsto [w] * [\![\tau \mid s \mid w]\!] \\
[\![\tau \to \tau' \mid \mathsf{sarrow}\, \gamma \mid v]\!] &\triangleq \exists P.\, \gamma \Rightarrow P * \square P * \\
&\qquad \mathsf{onlyleft}\,(\square \forall w\, s.\, \{\triangleright P * [\![\tau \mid s \mid w]\!]\}\ (v\, w)\ \{\lambda v'\, s'.\, [\![\tau' \mid s' \mid v']\!]\}) \\
[\![\Gamma \mid M \mid V]\!] &\triangleq \ulcorner \mathrm{dom}(\Gamma) = \mathrm{dom}(M) = \mathrm{dom}(V) \urcorner * \mathop{\Large *}_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x) \mid M(x) \mid V(x)]\!]
\end{aligned}$$

$$\text{where} \quad \mathsf{onlyleft}\,(P) \triangleq \lambda b.\, \text{if } b \text{ then } (P \text{ true}) \text{ else } \top$$

$$\begin{aligned}
&[\![\Gamma \vdash e : \tau \dashv \Gamma']\!] \triangleq \forall M\, V. \\
&\quad \{[\![\Gamma \mid M \mid V]\!]\}\ ([V/]e)\ \{\lambda v\, (s, M').\, \ulcorner \Gamma \approx \Gamma' \wedge M \approx M' \urcorner * [\![\tau \mid s \mid v]\!] * [\![\Gamma' \mid M' \mid V|_{\mathrm{dom}(\Gamma')}]\!]\}
\end{aligned}$$

Fig. 14. Semantic Interpretation of MiniDet

show that whenever the typing judgement holds, the corresponding triple is valid. The soundness theorem of the type system is then derived from the soundness of the underlying logic.

The Musketeer triple associated to a typing judgement makes use of a *logical relation*. Typically, when using program logic-based semantic typing, a logical relation is a relation expressed in the assertions of the underlying logic that relates a type to a value it inhabits. In our case, however, the logical relation involves three parameters: a type, a value, and a *shape*. The shape captures the "determinism" of each type and will be used in connection with ghost return values. For example, the shape of a reference is the actual value stored in this reference, and the shape of a function records that the function's environment is deterministic.

Figure 14 defines the format of shapes. A shape s as either an invalid shape (whose purpose is similar to the invalid type, as we equip shapes with a monoid operation), the none shape, storing no information, the product shape $\mathsf{sprod}\, s_1\, s_2$, the reference shape $\mathsf{sref}\, v\, s$, where v represents the content of the reference and s the shape associated with v and finally the arrow shape $\mathsf{sarrow}\, \gamma$, where γ is the name of an Iris ghost cell [Jung et al. 2018].

The logical relation $[\![\tau \mid s \mid v]\!]$, shown in Figure 14 then relates a type τ, a shape s, and a value v. This relation is itself of type *vProp*. Unboxed types are interpreted as expected, associated the with snone shape. Products must be associated to the product shape and a product value, and the interpretation must recursively hold. For the reference type $\mathsf{ref}\, \tau$, the shape must be a reference shape $\mathsf{sref}\, w\, s$, v must be a location ℓ such that ℓ points to w and that recursively $[\![\tau \mid s \mid w]\!]$ holds. Note that the interpretation of a reference expresses the ownership of the associated points-to. Besides, the content of the reference w is *not* existentially quantified, but rather given by the shape.

The case of an arrow $\tau \to \tau'$ is subtle and differs from the approach used in other program logic based logical relations. In the usual approach, the interpretation of $\tau \to \tau'$ says that v is in the relation if for any w in the interpretation of τ, a Hoare triple of a certain form holds for the application $v\, w$. Unfortunately, this approach cannot be used directly with Musketeer. The reason is that the usual approach exploits the fact that the underlying logic is higher-order and impredicative, so that a Hoare triple is itself an assertion that can appear in the pre/post-condition of another triple. In contrast, in Musketeer, the assertions appearing in pre/post-conditions are *vProp*s, but the triple itself is not a *vProp*, it is an *iProp* in the underlying chained logic, as we saw in §5.

To work around this, we define an operation onlyleft that takes an *iProp* and coerces it into a *vProp* by requiring the proposition to only hold for the left-hand side. Using this, the logical relation asserts that, only in the left case, for any value w and shape s, a Hoare triple holds for $v\, w$. In this triple, the precondition requires $[\![\tau \mid s \mid w]\!]$, and the postcondition says that the result will satisfy

the interpretation of τ'. The precondition additionally requires P to hold for some existentially quantified *vProp* predicate P. (Technically, P is assumed to hold under a *later modality* $\triangleright$, but this detail can be ignored.) This P will correspond to the resources associated with whatever variables from a typing environment the function closes over. Thus, P is required to hold under the Iris *persistent* modality $\square$, ensuring that the proposition is duplicable—recall that the typing rule T-Abs requires functions to close over only duplicable environments. Finally, there is one last trick: to ensure that this existential quantification over P can later be eliminated using M-ElimExist, the witness is made unique by using an Iris *saved predicate* assertion, lifted at the *vProp* level and written $\gamma \Rightarrow P$, which states that γ is the name of a ghost variable that stores the assertion P. The γ here is bound as part of the shape sarrow γ. Since a ghost variable can only store one proposition, only one P can satisfy this assertion.

Figure 14 then defines the interpretation of a typing environment Γ, a shape environment M and a value environment V, written $[\![\Gamma \mid M \mid V]\!]$ as the lifting per-variable x of the logical relation. Using this, we obtain the interpretation of the typing judgement $\Gamma \vdash e : \tau \dashv \Gamma'$. This interpretation universally quantifies over a shape environment M and a variable environment V, and asserts a Musketeer triple. The precondition is the interpretation of the environments, and targets an expression $[V/]e$, that is, the expression e with variables replaced by values as specified by V. The postcondition binds a return value v as well as a ghost return value consisting of a shape s and a shape environment M'. The postcondition asserts that the two typing environment Γ and Γ' are *similar*, written $\Gamma \approx \Gamma'$ and that the shape environments M and M' are also similar, with (overloaded) notation $M \approx M'$. Intuitively these relations guarantee that variables did not change in nature in environments (*e.g.* a reference stayed a reference, and a reference shape stayed a reference shape, even if the content may have changed). We formally define these statements in Appendix B. The postcondition finally asserts that the return value is related to τ and s and that the returned environment Γ' is correct with M' and the same variables V, dropping unneeded bindings.

With these definitions, we state the fundamental lemma of the logical relation.

Lemma 7.2 (Fundamental). *If* $\Gamma \vdash e : \tau \dashv \Gamma'$ *holds then* $[\![\Gamma \vdash e : \tau \dashv \Gamma']\!]$ *holds too.*

From this lemma, it is easy to prove the soundness of MiniDet (Lemma 7.1). Let us suppose that $\emptyset \vdash e : \tau \dashv \emptyset$ holds. We apply Lemma 7.2 and learn that $[\![\emptyset \vdash e : \tau \dashv \emptyset]\!]$ holds too. Unfolding definitions and applying M-Conseq, this fact implies that $\{\top\}\ e\ \{\lambda\_\_.\ \top\}$ holds. We conclude by applying the soundness of Musketeer (Theorem 4.1).

7.2 Priority Writes

In this section, we extend MiniDet with rules for *priority writes* [Blelloch et al. 2012]. A priority write targets a reference r on an integer x and atomically updates the content y of r to $x \max y$. As long as there are no concurrent reads, priority writes can happen in parallel: because max is associative and commutative, the order in which the parallel write operations happen does not matter. Conversely, so long as there are no ongoing concurrent writes, reads from the reference will be safe and deterministic—and such reads can also happen in parallel. Thus, priority writes are deterministic so long as they are used in a *phased* manner, alternating between concurrent writes in one phase, and concurrent reads in the next. For simplicity, we consider priority writes on integers equipped with the max function.

Implementation of priority writes. Figure 15 shows the implementation of priority references. Allocating a priority reference with palloc just allocates a reference. The priority read pread is just a plain get operation. A priority write pwrite is a function with recursive name f taking two arguments: r, the reference to update, and x, the integer to update the reference with. The function

$$\text{palloc} \triangleq \lambda n.\,\text{ref}\ n \qquad \text{pread} \triangleq \lambda r.\,\text{get}\ r$$
$$\text{pwrite} \triangleq \mu f\, r\, x.\,\text{let}\ y = \text{get}\ r\ \text{in if}\ x < y\ \text{then}\ () \ \text{else if}\ \text{CAS}\ r\ 0\ x\ y\ \text{then}\ () \ \text{else}\ f\ r\ x$$

Fig. 15. Implementation of Priority Writes

$$\tau \triangleq \cdots \mid \text{pwrite}\, q \mid \text{pread}\, q$$
$$\text{pwrite}\, q_1 \cdot \text{pwrite}\, q_2 \triangleq \text{pwrite}\,(q_1 + q_2) \qquad \text{pread}\, q_1 \cdot \text{pread}\, q_2 \triangleq \text{pread}\,(q_1 + q_2)$$

$$\text{T-PAlloc}\ \frac{\Gamma \vdash e : \text{int} \dashv \Gamma'}{\Gamma \vdash \text{palloc}\, e : \text{pwrite}\ 1 \dashv \Gamma'} \qquad \text{T-PWrite}\ \frac{\Gamma \vdash e : \text{int} \dashv \Gamma' \qquad \Gamma'(x) = \text{pwrite}\, q}{\Gamma \vdash e : \text{pwrite}\ x\, e \dashv \Gamma'}$$

$$\text{T-PRead}\quad \{x := \text{pread}\, q\} \vdash \text{pread}\, x : \text{int} \dashv \{x := \text{pread}\, q\} \qquad \text{T-Update}\ \frac{\Gamma \rightsquigarrow \Gamma' \qquad \Gamma' \vdash e : \tau \dashv \Gamma''}{\Gamma \vdash e : \tau \dashv \Gamma''} \qquad \text{U-Refl}\quad \tau \rightsquigarrow \tau$$

$$\text{U-Pair}\ \frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{(\tau_1 \times \tau_2) \rightsquigarrow (\tau_1' \times \tau_2')} \qquad \text{U-R2W}\quad \text{pread}\ 1 \rightsquigarrow \text{pwrite}\ 1 \qquad \text{U-W2R}\quad \text{pwrite}\ 1 \rightsquigarrow \text{pread}\ 1$$

Fig. 16. Extension of MiniDet with Priority Writes

tests if the content y of the reference is greater than x. If $x < y$, the function returns, because $x \max y = y$. Else, the function attempts to overwrite y with x in r with a CAS, and loops if it fails.

As noted by Blelloch et al. [2012], if we break the abstractions of the priority reference, the implementation of pwrite is not internally deterministic: because pwrite reads r, a location that can be written by a parallel task, different interleavings might see different values. However, because pwrite is carefully designed, these nondeterministic observations are not externally visible and do not impact the safety of the program. As we will see, this latter fact allow us to derive a Musketeer triple API to priority writes. However, because nondeterminism is involved internally in the implementation, we conduct the proof at the level of ChainedLog.

Extension of MiniDet. Figure 16 shows how we extend our type system. We add two new type constructors, pwrite q and pread q, asserting that the reference is in a write phase with fraction q or a read phase with fraction q, respectively. The monoid on types is extended to sum fractions. This definition implies, as we will see, that writes can happen in parallel with writes, and reads can happen in parallel with reads.

The lower part of Figure 16 shows the new typing rules. T-PAlloc allocates a priority reference and returns a type pwrite 1. T-PWrite types a priority write on some reference x bound to the type pwrite q. In particular, this rule does *not* require the full fraction 1, meaning that the write operation can happen in parallel of other write operations. T-PRead types a read similarly. Again this rule does not require the full fraction. T-Update allows for updating a typing context Γ into Γ' as long as $\Gamma \rightsquigarrow \Gamma'$. This relation is defined pointwise over the elements of the environments as the update relation $\tau \rightsquigarrow \tau'$ which is defined last in Figure 16. U-Refl asserts that a type can stay the same, U-Pair distributes over pairs, U-R2W transforms a read type into a write one, if the fraction is the full permission 1. This precondition on the fraction is important: it asserts that no parallel task use the priority reference. U-W2R is symmetrical.

Extending the soundness proof. To extend the soundness proof to support these new rules, we first prove specifications for the priority reference operations in Musketeer, shown in the upper part of Figure 17. These specifications involve two predicates: ispw $\ell\, q\, i$, asserting that ℓ is a priority reference, and that ℓ is in its concurrent phase with fraction q and stores (at least) i. Symmetrically, ispr $\ell\, q\, i$ asserts that ℓ is in its read phase. The specification of palloc i asserts that this function

$$\begin{array}{c} \{\top\} \quad \mathsf{palloc}\, i \quad \{\lambda r\, \_.\, \mathsf{ispw}\, \ell\, 1\, i\} \\ \{\mathsf{ispw}\, \ell\, q\, i\} \ \mathsf{pwrite}\ \ell\, j \ \{\lambda r\, \ell.\, \ulcorner v = \ell \urcorner * \mathsf{ispw}\, \ell\, q\, (i \max j)\} \\ \{\mathsf{ispr}\, \ell\, q\, i\} \quad \mathsf{pread}\, \ell \quad \{\lambda r\, \_.\, \ulcorner v = i \urcorner * \mathsf{ispr}\, \ell\, 1\, i\} \end{array}$$

$$\begin{array}{rl} \mathsf{ispw}\, \ell\, (q_1 + q_2)\, (i \max j) \dashv\vdash \mathsf{ispw}\, \ell\, q_1\, i * \mathsf{ispw}\, \ell\, q_2\, j & \qquad s \triangleq \cdots \mid \mathsf{spwrite}\, i \mid \mathsf{spread}\, i \\ \mathsf{ispr}\, \ell\, (q_1 + q_2)\, i \dashv\vdash \mathsf{ispr}\, \ell\, q_1\, i * \mathsf{ispr}\, \ell\, q_2\, i & \qquad [\![\mathsf{pwrite}\, q \mid \mathsf{spwrite}\, i \mid v]\!] \triangleq \exists \ell.\, \ulcorner v = \ell \urcorner * \mathsf{ispw}\, \ell\, q\, i \\ \mathsf{ispw}\, \ell\, 1\, i \dashv\vdash \mathsf{ispr}\, \ell\, 1\, i & \qquad [\![\mathsf{pread}\, q \mid \mathsf{spwrite}\, i \mid v]\!] \triangleq \exists \ell.\, \ulcorner v = \ell \urcorner * \mathsf{ispr}\, \ell\, q\, i \end{array}$$

Fig. 17. Specifications of Priority Writes and Logical Interpretation

```
alloc_fill ≜ λn v. fill (alloc n) v

init ≜ λh n.
  assert (n ≥ 0);
  let d = ref () in
  let a = alloc_fill n d in
  (a, d, h)

elems ≜ λ(a, d, h).
  filter_compact a d

add ≜ λ(a, d, h) x.
  let put = μf x i.
    let y = a[i] in
    if x == y then () else
    if x == d then (if CAS a i d x then () else f x i) else
    let j = (i + 1) mod (length a) in
    if x < y then f x j else (if CAS a i y x then f y j else f x i) in
  put x ((h i) mod (length a))
```

Fig. 18. Implementation of a Deterministic Concurrent Hash Set

call returns a location ℓ such that $\mathsf{ispw}\, \ell\, q\, 1$ holds. The specification of $\mathsf{pwrite}\ \ell\, j$ updates a share $\mathsf{ispw}\, \ell\, q\, i$ into $\mathsf{ispw}\, \ell\, q\, (i \max j)$. The specification of $\mathsf{pread}\, \ell$ asserts that this function call returns the content of a priority reference, if this reference is in its read phase.

The central part of Figure 17 shows the splitting and joining rules of the ispw and ispr assertions. It also shows that one can update a ispw assertion into a ispr assertion, and vice-versa, as long as the fraction in 1 (formally, these conversions involve the so-called *ghost updates* [Jung et al. 2018]).

The lower part of Figure 17 intuits how we extend the logical relation backing the soundness of our type system. We add two shapes, one for each phase. We then extend the logical relation as expected, making use of the previous assertions.

7.3 Deterministic Concurrent Hash Sets

Next, we extend MiniDet with a *deterministic concurrent hash set*, inspired by Shun and Blelloch [2014]. This hash set allows for concurrent, lock-free insertion, and offers a function elems that returns an array with the inserted elements in some arbitrary but deterministic order. This hash set is implemented as an array, and makes use of *open addressing* and *linear probing* to handle collision. The key idea to ensure determinism is that neighboring elements in the array are *ordered* according to a certain total order relation. As we will see, insertion preserves the ordering, which in turn ensures determinism of the contents of the array. Shun and Blelloch [2014] also propose a deletion function, which we do not verify. The hash set usage must be *phased*: insertion is allowed to take place in parallel as long as no task calls the function elems.

Implementation of our hash set. Figure 18 presents the implementation of the deterministic hash set. While in our mechanization we support a hash set over arbitrary values, for space constraints we present here an implementation specialized to integers, equipped with the comparison function $<$.

A new hash set is initialized with the function $\mathsf{init}\, h\, n$, which returns a tuple (a, d, h), where a is the underlying array, d is a dummy element (in our case, a fresh reference containing the unit value) representing an empty slot in the array. The function h is the hash function. The implementation uses a helper routine, $\mathsf{alloc\_fill}\, n\, d$, which allocates an array and fills it with the value v using a

$$\tau \triangleq \cdots \mid \mathsf{intarray}\, q \mid \mathsf{intset}\, q$$

$$\mathsf{intarray}\, q_1 \cdot \mathsf{intarray}\, q_2 \triangleq \mathsf{intarray}\,(q_1 + q_2) \qquad \mathsf{intset}\, q_1 \cdot \mathsf{intset}\, q_2 \triangleq \mathsf{intset}\,(q_1 + q_2)$$

T-AAlloc
$$\frac{\Gamma_1 \vdash e_1 : \mathsf{int} \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : \mathsf{int} \dashv \Gamma_3}{\Gamma \vdash \mathsf{alloc\_fill}\, e_2\, e_1 : \mathsf{intarray}\, 1 \dashv \Gamma_3}$$

T-ALoad
$$\frac{\Gamma_1 \vdash e : \mathsf{int} \dashv \Gamma_2 \qquad \Gamma_2(x) = \mathsf{intarray}\, q}{\Gamma_1 \vdash x[e] : \mathsf{int} \dashv \Gamma_2}$$

T-AStore
$$\frac{\Gamma_1 \vdash e_1 : \mathsf{int} \dashv \Gamma_2 \qquad \Gamma_2 \vdash e_2 : \mathsf{int} \dashv \Gamma_3 \qquad \Gamma_3(x) = \mathsf{intarray}\, 1}{\Gamma \vdash x[e_2] \leftarrow e_1 : \mathsf{unit} \dashv \Gamma_3}$$

T-SAlloc
$$\frac{\Gamma \vdash e : \mathsf{int} \dashv \Gamma' \qquad (\forall x.\ \{\top\}\ h\, x\ \{\lambda v\ \_.\ \ulcorner v = hash(x) \urcorner\})}{\Gamma \vdash \mathsf{init}\, h\, e : \mathsf{intset}\, 1 \dashv \Gamma'}$$

T-SAdd
$$\frac{\Gamma \vdash e : \mathsf{int} \dashv \Gamma' \qquad \Gamma'(x) = \mathsf{intset}\, q}{\Gamma \vdash \mathsf{add}\, x\, e : \mathsf{unit} \dashv \Gamma'}$$

T-SElems
$$\frac{\Gamma \vdash e : \mathsf{intset}\, 1 \dashv \Gamma'}{\Gamma \vdash \mathsf{elems}\, e : \mathsf{intarray}\, 1 \dashv \Gamma'}$$

Fig. 19. Extension of MiniDet with Integer Arrays and Hash Set

function fill, which we omit for brevity. The function elems (a, d, h) returns a fresh array containing the elements of a obtained by filtering those equal to the dummy element d. The key challenge in the design is to ensure that this operation will be deterministic: in conventional linear probing hash tables, the order of elements in the array would depend on the order of insertions, so concurrent insertions would lead to nondeterministic orders.

To avoid this nondeterminism, the function add $(a, d, h)\ x$, which inserts x in the hash set (a, d, h), enforces an ordering on elements in the array according to the comparison function $<$. The code makes use of a recursive auxiliary function *put*, parameterized by an element x and an index i, which tries to insert x at i. The function *put* loads the content of the array a at offset i and names it y. If y is equal to x, then x is already in the set and the function returns. If y is equal to the dummy element, the function tries a CAS to replace y with x, and loops in case the CAS fails. Otherwise, y is an element distinct from x. The function names the next index $j = (i + 1) \bmod (\mathsf{length}\, a)$ and tests if $x < y$. If y is greater than x, the function tries to insert x at the next index j by doing a recursive call of $f\ x\ j$. If x is greater than y, the function tries to replace y with x with a CAS, and loops if the CAS fails. If the CAS succeeds, the function removed y from the hash set, and must hence insert it again by doing a recursive call $f\ y\ j$.

The function add then simply calls *put* to insert x at the initial index $(h\, x) \bmod (\mathsf{length}\, a)$.

Extension of MiniDet. Figure 19 presents the extension of MiniDet with this hash set. To avoid issues related to ownership of the elements in the set, we consider a hash set containing integers.

We add two new types: $\mathsf{intarray}\, q$ describing an array of integers with a fraction q and $\mathsf{intset}\, q$ a hash set of integers with a fraction q. The monoid on types is extended to sum the fractions.

T-AAlloc types the allocation of an array filled with a default element. T-ALoad types a load operation on an array bound to the variable x. This operation requires any fraction of intarray. T-AStore types a store operation but requires full ownership of the array—that is, the fraction 1.

T-SAlloc allocates a hash set. This rule has one non-syntactical precondition, which cannot be handled by a type system. It requires that the hash function h, the first parameter of add, implements some arbitrary pure function $hash : \mathcal{V} \to Z$. This proof can be derived in Musketeer, and ensures that calls to the hash function are deterministic. T-SAlloc returns a intset type with fraction 1.

T-SAdd types an add operation on a hash set x with an arbitrary fraction q, meaning that this operation can happen in parallel. T-SElems types the elems operation, requiring the full ownership

$$\frac{(\forall x.\ \{\top\}\ h\,x\ \{\lambda v\ \_.\ \ulcorner v = hash(x)\urcorner\})}{\{\top\}\ \mathsf{init}\,h\,i\ \{\lambda v\ \_.\ \mathsf{hashset}\,v\,1\,\emptyset\}} \qquad \begin{array}{l}\{\mathsf{hashset}\,v\,q\,X\}\ \mathsf{add}\,v\,i\ \{\lambda r\ \_.\ \mathsf{hashset}\,v\,q\,(\{i\}\cup X)\}\\ \{\mathsf{hashset}\,v\,1\,X\}\ \mathsf{elems}\,v\ \{\lambda v'\,(\ell,\vec{w}).\ \ulcorner v' = \ell\urcorner * \ell \mapsto \vec{w}\}\end{array}$$

$$\mathsf{hashset}\,v\,(q_1+q_2)\,(X_1\cup X_2) \dashv\vdash \mathsf{hashset}\,v\,q_1\,X_1 * \mathsf{hashset}\,v\,q_2\,X_2$$

$$s \triangleq \cdots \mid \mathsf{sintset}\,X \qquad [\![\,\mathsf{intset}\,q \mid \mathsf{sintset}\,X \mid v\,]\!] \triangleq \mathsf{hashset}\,v\,q\,X$$

Fig. 20. Specifications of a Deterministic Hash Set and Logical Interpretation

```
parfor ≜ μ̂f. λi j k.
  if (j − i) == 0 then ()
  else if (j − i) == 1 then k i
  else let mid = i + ((j − i)/2) in
    par (f i mid k) (f mid j k)
```

```
dedup ≜ λh a.
  let start = 0 in
  let len = length a in
  let s = init h (len + 1) in
  parfor start len (λi. add s (a[i]));
  prod a (elems s)
```

Fig. 21. Implementation of parfor and dedup Functions

of a hash set, and producing a fresh array. This operation consumes the hash set argument; this is for simplicity: the hash set is only read and is in fact preserved by the operation.

Extending the soundness proof. The upper part of Figure 20 presents the Musketeer specifications of the hash set operations. These specifications make use of an assertion $\mathsf{hashset}\,v\,q\,X$ asserting that v is a hash set with fraction q and content at least X, a set of values. When $q = 1$, then X is exactly the set of values in the set. The specification of $\mathsf{init}\,h\,i$ returns a fresh set with fraction 1 and no elements, provided that the parameter h behaves correctly. The specification of $\mathsf{add}\,v\,i$ verifies the insertion of an integer i in a hash set v with an arbitrary fraction q and current content X, which the function call updates to $(\{i\} \cup X)$. Since we specialize to hash sets of integers, we know that the inserted value will not be the dummy element. In our mechanization, we offer a more general specification, allowing the user to insert other pointers as long as they ensure that the inserted pointer is not the dummy element. Perhaps most importantly, the specification of $\mathsf{elems}\,v$ consumes an assertion $\mathsf{hashset}\,v\,1\,X$ with fraction 1 and produces an array ℓ with a *deterministic* content $\vec{w}$. Figure 20 then gives the reasoning rule for splitting a hashset assertion, enabling parallel use.

The lower part of Figure 20 shows how we extend the logical relation. We add a shape $\mathsf{sintset}\,X$, where X a set of integers. The interpretation of $\mathsf{intset}\,q$ with shape $\mathsf{sintset}\,X$ and value v is then simply $\mathsf{hashset}\,v\,q\,X$.

7.4 Deduplication via Concurrent Hashing

For our last example, we consider *array deduplication*, one of the parallel benchmark problems proposed by Blelloch et al. [2012]. The task is to take an array of elements and return an array containing the same elements but with duplicates removed. The solution proposed by Blelloch et al. [2012] is to simply insert all the elements in parallel into a deterministic hash set and then return the elements of the hash set. Figure 21 presents dedup, an implementation of this algorithm in MusketLang. To do the parallel inserts, it uses a helper routine called $\mathsf{parfor}\,i\,j\,k$, which runs $(k\,n)$ in parallel for all n between i and j. Our goal is to prove that dedup satisfies schedule-independent safety, and then prove a specification in Angelic. Throughout this proof, we assume that we have some hash function h such that $\forall x.\ \{\top\}\ (h\,x)\ \{\lambda v\ \_.\ \ulcorner v = hash\,x\urcorner\}$ and $\forall x.\ \mathsf{run}\,(h\,x)\,\{\lambda v.\ \ulcorner v = hash\,x\urcorner\}$, where *hash* is some function in the meta-logic.

Our first step is to show that dedup can be typed in MiniDet. This follows by using a typing rule for parfor (given in Appendix C.1), and the earlier typing rules we derived for the hash set. Using

these, we derive $\emptyset \vdash \mathsf{dedup}\, h : \mathsf{intarray}\, q \to (\mathsf{intarray}\, q \times \mathsf{intarray}\, 1) \dashv \emptyset$. Thus, for a well-typed input array a, $\mathsf{dedup}\, h\, a$ satisfies schedule-independent safety.

We then verify dedup using Angelic. The proof uses Angelic reasoning rules for the hash set, shown in Appendix C.2, which are similar to the earlier Musketeer specifications (§ 7.3), except for three key points. First, the Angelic specification shows that, for a set v with content X, $\mathsf{elems}\, v$ returns an array $\vec{w}$ which contains just the elements of the set X. Second, the representation predicate for the hash set has no fraction: there is never a need for splitting it in Angelic. Third, as we require the user to prove termination, the representation predicate tracks how many elements have been inserted, and does not allow inserting into a full table.

Finally, we use a derived specification for $\mathsf{parfor}\, i\, j\, k$ that allows us to reason about it as if it were a sequential for-loop:

$$\mathsf{forspec}\, i\, j\, k\, \varphi \;\vdash\; \mathsf{run}\,(\mathsf{parfor}\, i\, j\, k)\,\{\lambda v.\, \ulcorner v = () \urcorner * \varphi\}$$

Here, $\mathsf{forspec}\, i\, j\, k\, \varphi$ is defined recursively as

$$\mathsf{forspec}\, i\, j\, k\, \varphi \;\triangleq\; \left(\ulcorner i \geq j \urcorner * \varphi\right) \;\vee\; \left(\ulcorner i < j \urcorner * \mathsf{run}\,(k\, i)\,\{\lambda v.\, \ulcorner v = () \urcorner * \mathsf{forspec}\,(i+1)\, j\, k\, \varphi\}\right)$$

In this definition, either $i \geq j$ and the postcondition holds (since there are no recursive calls to be done), or $i < j$, and the user has to verify $k\, i$, and show that $\mathsf{forspec}\,(i+1)\, j\, k\, \varphi$ holds afterward. Essentially, this generalizes the idea we saw earlier in A-ParSeqL, by having us verify an interleaving that executes each task sequentially from i to j. With these specifications, we deduce the following Angelic specification for dedup:

$$\ell \mapsto_q \vec{v} \;\vdash\; \mathsf{run}\,(\mathsf{dedup}\, h\, \ell)\,\{\lambda v.\, \exists \ell'\, \vec{w}.\, \ell \mapsto_q \vec{v} * \ell' \mapsto \vec{w} * \ulcorner \mathsf{deduped}\, \vec{w}\, \vec{v} \urcorner\}$$

8 Related Work

Deterministic parallel languages. As shown in Section 7.1, Musketeer can be used to prove the soundness of language-based techniques for enforcing determinism. A large body of such techniques exist, and it would be interesting to apply Musketeer to some of these. In general, these languages typically ensure determinism by restricting side effects (e.g., in purely functional languages) or by providing the programmer with fine-grained control over scheduling of effects (e.g., in the form of a powerful type-and-effect system). Examples include seminal works such as Id [Arvind et al. 1989] and NESL [Blelloch et al. 1994] as well as related work on Deterministic Parallel Java [Bocchino Jr. et al. 2009, 2011], parallelism in Haskell [Jones et al. 2008; Keller et al. 2010; Chakravarty et al. 2011, 2001; Marlow et al. 2011], the LVars/LVish framework [Kuper et al. 2014a,b; Kuper and Newton 2013], Liquid Effects [Kawaguchi et al. 2012]. Manticore [Fluet et al. 2007], SAC [Scholz 2003], Halide [Ragan-Kelley et al. 2013], Futhark [Henriksen et al. 2017], and many others.

It is typically challenging to formally prove sequentialization or determinization results for these kinds of languages, particularly in an expressive language with features like higher-order state and recursive types. For example, Krogh-Jespersen et al. [2017] point out that it took 25 years for the first results proving that in a type-and-effect system, appropriate types can ensure that a parallel pair is contextually equivalent to a sequential pair. They show how a program-logic based logical relation, like the one we used in Section 7, can vastly simplify such proofs. Musketeer provides a program logic that is well-suited for constructing models to prove whole-language determinism properties. Although not discussed in this paper, we have already completed a proof of schedule-independent safety for a simplified model of the LVars framework. We believe similar results may be possible for other deterministic-by-construction languages.

Logic for hyperproperties. So-called *relational* program logics have been developed to prove hyperproperties. Naumann [2020] provides an extensive survey of these logics. A number of such

logics support very general classes of hyperproperties [D'Osualdo et al. 2022; Sousa and Dillig 2016]. However, most of the relational logics building on concurrent separation logic have been restricted ∀∃ hyperproperties [Liang and Feng 2016; Frumin et al. 2021; Gäher et al. 2022; Timany et al. 2024a]. Because schedule-independent safety is a ∀∀ property, it falls outside the scope of these logics, which motivated our development of ChainedLog. In the world of ∀∀ hyperproperties, several Iris-based relational logics have been proposed. For example, Frumin et al. [2021] and Gregersen et al. [2021] verify variations of *non-interference* in a sequential setting. Both works require that both executions terminate. However, their underlying relational logics do not support Musketeer's distinctive feature: the chaining rule C-Chain. In another setting, Timany et al. [2017] present a logical relation showing that Haskell's ST monad [Launchbury and Peyton Jones 1995] properly encapsulates state. They show such a result using a *state-independence* property which intuitively asserts that, for a well-typed program, if one execution terminates with a particular initial heap, then every execution terminates with any other initial heap.

Most logics for hyperproperties are structured as relational logics. However, some, like Musketeer, prove a hyperproperty through unary reasoning. For example, Dardinier and Müller [2024], target arbitrary hyperproperties for a pure language, with a triple referring to a single expression, but with pre/post-conditions describing multiple executions. Eilers et al. [2023] present CommCSL, a concurrent separation logic for proving *abstract commutativity*, that is, where two operations commute *up-to* some abstract interface. This idea appears for example in the API for priority writes, which implies that writes commutes (§7.2). In contrast with our approach, CommCSL is globally parameterized by a set of specifications the logic ensures commute. In Musketeer, no such parameterization is needed: proof obligations are entirely internalized.

Commutativity-Based Reasoning. Schedule-independent safety reduces the problem of verifying safety for all executions of a program to just verifying safety of any one terminating execution. This can be seen as an extreme form of a common technique in concurrent program verification, in which the set of possible executions of a program is partitioned into equivalence classes, and then a representative element of each equivalence class is verified [Farzan 2023]. This approach has its origins in the work of Lipton [1975], and typically uses some form of analysis to determine when statements in a program commute in order to restructure programs into an equivalent form that reduces the set of possible nondeterministic outcomes [Elmas et al. 2009; Kragl and Qadeer 2021; von Gleissenthall et al. 2019; Farzan et al. 2022]. For programs satisfying schedule-independent safety, there is effectively only one equivalence class, allowing a user of Angelic to dynamically select one ordering to verify.

9 Conclusion and Future Work

Schedule-independent safety captures the essence of why internal determinism simplifies reasoning about parallel programs. In this paper, we have shown how Musketeer provides an expressive platform for proving that language-based techniques ensure schedule-independent safety, and how Angelic can take advantage of schedule-independent safety. One limitation of schedule-independent safety is that it is restricted to safety properties. In future work, it would be interesting to extend Musketeer for proving that liveness properties, such as termination, are also schedule-independent.

Acknowledgments

This work was supported by the National Science Foundation through grant no. 2318722 and 2319168. The authors thank the anonymous reviewers of the paper and associated artifact for their feedback.

Data Availability Statement

The Musketeer and Angelic logics, their soundness proofs, and all our case studies are mechanized in the Rocq prover using the Iris framework. This mechanization is recorded in an artifact available on Zenodo [Moine et al. 2025].

$$\begin{array}{lll} \mathsf{unsafe} & \triangleq & \mathsf{let}\, r \,=\, \mathsf{ref}\, \mathsf{true}\, \mathsf{in} \\ & & \mathsf{par}\, (\mathsf{set}\, r\, \mathsf{true})\, (\mathsf{set}\, r\, \mathsf{false}); \\ & & \mathsf{assert}\, (\mathsf{get}\, r) \end{array}$$

Fig. 22. An Unsafe Code

A A Counter-Example to the Existantial Elimination Rule in Musketeer

In this section, we explain in more detail why adding the standard separation logic rule for eliminating an existential to Musketeer would be unsound.

Consider the unsafe program presented in Figure 22. This program allocates a reference r initialized to false, then executes in parallel two writes, the left one setting r to true, and the right one to false. After the parallel phase, the program asserts that the content of r is true. This program does not satisfy schedule-independent safety. Indeed, if the left write is scheduled before the right one, r will contain false and the assert will fail. Hence, Musketeer must reject the unsafe program.

Yet, let us attempt a Musketeer proof, and let us show that this proof would succeed if the user is able to eliminate an existential without restriction. Let us attempt to verify the following triple

$$\{\top\}\ \mathsf{unsafe}\ \{\lambda\_\_.\ \top\}$$

After the allocation of r, we have to verify

$$\{r \mapsto \mathsf{false}\}\ \mathsf{par}\, (\mathsf{set}\, r\, \mathsf{true})\, (\mathsf{set}\, r\, \mathsf{false});\ \mathsf{assert}\, (\mathsf{get}\, r)\ \{\lambda\_\_.\ \top\}$$

Because the program next performs a concurrent write on the same location r, we have to use an *invariant* in order to share the ownership of r between the two tasks. Let us pick the invariant $(\exists x.\, r \mapsto x)$. We now have to verify

$$\{\boxed{\exists x.\, r \mapsto x}\}\ \mathsf{par}\, (\mathsf{set}\, r\, \mathsf{true})\, (\mathsf{set}\, r\, \mathsf{false});\ \mathsf{assert}\, (\mathsf{get}\, r)\ \{\lambda\_\_.\ \top\}$$

Using M-Store and standard rules for invariants, we can easily establish the intermediate triple $\forall n.\, \{\boxed{\exists x.\, r \mapsto x}\}\ \mathsf{set}\, r\, n\ \{\lambda\_\_.\ \top\}$. Using this intermediate triple, the fact that we can duplicate invariants and rules M-Par and M-Bind, we are left with proving

$$\{\boxed{\exists x.\, r \mapsto x}\}\ \mathsf{assert}\, (\mathsf{get}\, r)\ \{\lambda\_\_.\ \top\}$$

We apply M-Bind to focus on the sub-expression $(\mathsf{get}\, r)$ with the weakest possible intermediate postcondition Q', that is we instantiate Q' with $(\lambda\_\_.\top)$. The two preconditions of M-Bind are

$$\{\boxed{\exists x.\, r \mapsto x}\}\ \mathsf{get}\, r\ \{\lambda\_\_.\ \top\} \qquad \text{and} \qquad \forall v.\, \{\top\}\ \mathsf{assert}\, v\ \{\lambda\_\_.\ \top\}$$

The right triple immediately follows from M-Assert. Using the opening rule of invariants on the left triple, we have to verify

$$\{\exists x.\, r \mapsto x\}\ \mathsf{get}\, r\ \{\lambda\_\_.\ \exists x.\, r \mapsto x\}$$

In order to conclude, one has to eliminate the existential, before using M-Load. Thankfully, Musketeer prevents the user from eliminating the existential, and the proof ultimately fails. What happened here? The existential quantification on x hid the fact that there are two possible *scheduling-dependent* witnesses (true and false).

$$\textsc{S-Empty}\ \ \mathsf{empty} \approx \mathsf{empty} \qquad \textsc{S-Unit}\ \ \mathsf{unit} \approx \mathsf{unit} \qquad \textsc{S-Bool}\ \ \mathsf{bool} \approx \mathsf{bool} \qquad \textsc{S-Int}\ \ \mathsf{int} \approx \mathsf{int} \qquad \textsc{S-HashSet}\ \ \mathsf{intset}\, q_1 \approx \mathsf{intset}\, q_2$$

$$\textsc{S-Array}\ \ \mathsf{intarray}\, q_1 \approx \mathsf{intarray}\, q_2 \qquad \textsc{S-Ref}\ \ \mathsf{ref}\, \tau_1 \approx \mathsf{ref}\, \tau_2 \qquad \textsc{S-Arrow}\ \ \tau_1 \to \tau_2 \approx \tau_1 \to \tau_2 \qquad \textsc{S-Prod}\ \ \frac{\tau_1 \approx \tau_1' \qquad \tau_2 \approx \tau_2'}{(\tau_1 \times \tau_2) \approx (\tau_1' \times \tau_2')}$$

$$\textsc{S-PRead}\ \ \frac{\tau = \mathsf{pread}\, q_2 \ \lor \ \tau = \mathsf{pwrite}\, q_2}{\mathsf{pread}\, q_1 \approx \tau} \qquad \textsc{S-PWrite}\ \ \frac{\tau = \mathsf{pread}\, q_2 \ \lor \ \tau = \mathsf{pwrite}\, q_2}{\mathsf{pwrite}\, q_1 \approx \tau}$$

Fig. 23. Similar Predicate on Types

$$\textsc{S-SNone}\ \ \mathsf{snone} \approx \mathsf{snone} \qquad \textsc{S-SHashSet}\ \ \mathsf{sintset}\, X_1 \approx \mathsf{sintset}\, X_2 \qquad \textsc{S-SArray}\ \ \mathsf{sintarray}\, \vec{v}_1 \approx \mathsf{sintarray}\, \vec{v}_2 \qquad \textsc{S-SRef}\ \ \mathsf{sref}\, v_1 \approx \mathsf{ref}\, v_2$$

$$\textsc{S-SArrow}\ \ \mathsf{sarrow}\, \gamma \approx \mathsf{sarrow}\, \gamma \qquad \textsc{S-SProd}\ \ \frac{s_1 \approx s_1' \qquad s_2 \approx s_2'}{\mathsf{sprod}\, s_1\, s_2 \approx \mathsf{sprod}\, s_1'\, s_2'} \qquad \textsc{S-SPRead}\ \ \frac{s = \mathsf{spread}\, i_2 \ \lor \ s = \mathsf{spwrite}\, i_2}{\mathsf{spread}\, i_1 \approx s}$$

$$\textsc{S-SWrite}\ \ \frac{s = \mathsf{spread}\, i_2 \ \lor \ s = \mathsf{spwrite}\, i_2}{\mathsf{spwrite}\, i_1 \approx s}$$

Fig. 24. Similar Predicate on Shapes

$$\textsc{T-ParFor}\ \ \frac{\Gamma(x) = \Gamma(y) = \mathsf{int} \qquad \mathsf{Fractional}\, \Gamma\, \Gamma_f \qquad \forall q.\ [i := \mathsf{int}](\Gamma_f\, q) \vdash e : \mathsf{unit} \dashv \Gamma_f\, q}{\Gamma \vdash \mathsf{parfor}\, x\, y\, (\lambda i.\, e) : \mathsf{unit} \dashv \Gamma}$$

Fig. 25. A MiniDet Type for parfor

B Definition of the Similarity between Typing and Shape Environments

Figure 23 shows the definition of $\tau_1 \approx \tau_2$, asserting that the two MiniDet types τ_1 and τ_2 are similar (§7.1). This property asserts that both types have the same structure except that functions must be equal and that priority reads and priority writes are identified. Figure 24 shows the definition of $s_1 \approx s_2$, asserting that the two shapes s_1 and s_2 are similar. This property asserts that both shapes have the same structure, except that function shapes must be equal and that priority reads and priority writes are identified.

We extend these two predicates to maps $m_1 \approx m_2$ as the trivial predicate for the keys not in the intersection of $\mathrm{dom}(m_1)$ and $\mathrm{dom}(m_2)$ and the similar predicate when a key appears in both maps.

C Additional Explanations on the Concurrent Hash Set Example

C.1 A Typing Rule for parfor

In order to give a type in MiniDet to dedup (§7.4), we first give parfor a type, which we prove sound by dropping to the semantic model. T-ParFor, which appear in Figure 25, requires the two indices to be variables bound to integers, for simplicity. It then requires the environment Γ to be *fractional*, that is, to contain only fractional assertion. This is witnessed by the precondition $\mathsf{Fractional}\, \Gamma\, \Gamma_f$ which is defined as $(\forall n.\, n \neq 0 \implies \Gamma = \cdot^n(\Gamma_f\, n))$, that is, for every positive integer n, $\Gamma_f\, n$ represents a n-th share of Γ. Finally, T-ParFor requires to type the last argument of parfor, which must be a function of the form $\lambda i.\, e$. The precondition requires that e is typeable while borrowing a share $\Gamma_f\, n$ of the environment.

$$\text{A-HAlloc}\quad \frac{\square(\forall x.\, \mathsf{run}\,(h\,x)\,\{\lambda v.\ulcorner v = \mathit{hash}\,x\urcorner\})}{\mathsf{run}\,(\mathsf{init}\,h\,i)\,\{\lambda v.\ \mathsf{ahashset}\,i\,v\,\emptyset\}} \qquad \text{A-HAdd}\quad \frac{\ulcorner \mathit{size}\,X < i\urcorner \qquad \mathsf{ahashset}\,i\,v\,X}{\mathsf{run}\,(\mathsf{add}\,v\,x)\,\{\lambda w.\ulcorner w = ()\urcorner * \mathsf{ahashset}\,i\,v\,(\{x\}\cup X)\}}$$

$$\text{A-HElems}\quad \frac{\mathsf{ahashset}\,i\,v\,X}{\mathsf{run}\,(\mathsf{elems}\,v)\,\{\lambda v'.\,\exists \ell\,\vec{w}.\ulcorner v' = \ell\urcorner * \ell \mapsto \vec{w} * \ulcorner \mathsf{deduped}\,X\,\vec{w}\urcorner\}}$$

Fig. 26. Angelic Specifications for a Concurrent Hash Set

C.2 Angelic Reasoning Rules for our Concurrent Hash Set

Figure 26 presents the Angelic reasoning rules for our councurrent hash set (§7.3). These specifications involve a representation predicate $\mathsf{ahashset}\,i\,v\,X$, where i is the capacity (that is, the maximum number that can be contained in the set), v is the hash set and X the logical set with the inserted element. Note that this predicate is *not* fractional, as there is no need to ever split it.

A-HAlloc verifies $\mathsf{init}\,h\,i$. The precondition requires that the hash function h implements a hash function in the meta-logic.

A-HAdd verifies $\mathsf{add}\,v\,x$. The precondition requires that v is a set with content X and capacity i. The user must ensure that the size of the set X is less than the capacity, in order to guarantee termination. The postcondition returns the set with an updated model.

A-HElems verifies $\mathsf{elems}\,v$. The precondition requires $C_3\,i$ that v is a set with content X. The postcondition returns a fresh array ℓ pointing to $\vec{w}$ such that $\mathsf{deduped}\,X\,\vec{w}$ holds.

References

Arvind, Rishiyur S. Nikhil, and Keshav Pingali. 1989. I-Structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (1989). doi:10.1145/69558.69562

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-Enforced Deterministic Parallelism. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association. http://www.usenix.org/events/osdi10/tech/full_papers/Aviram.pdf

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *PPoPP '12* (New Orleans, Louisiana, USA).

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distributed Comput.* 21, 1 (1994). doi:10.1006/JPDC.1994.1038

Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM. doi:10.1145/1640089.1640097

Robert L. Bocchino Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, Thomas Ball and Mooly Sagiv (Eds.). ACM. doi:10.1145/1926385.1926447

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*. 259–270. http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/permissions_paper.pdf

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 2694)*. Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. 2001. Nepal - Nested Data Parallelism in Haskell. In *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2150)*, Rizos Sakellariou, John A. Keane, John R. Gurd, and Len Freeman (Eds.). Springer. doi:10.1007/3-540-44681-8_76

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP*, Manuel Carro and John H. Reppy (Eds.). ACM. doi:10.1145/1926354.1926358

Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (2010). doi:10.3233/JCS-2009-0393

Thibault Dardinier and Peter Müller. 2024. Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties. *Proc. ACM Program. Lang.* 8, PLDI (2024). doi:10.1145/3656437

Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 2022. Proving hypersafety compositionally. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022). doi:10.1145/3563298

Marco Eilers, Thibault Dardinier, and Peter Müller. 2023. CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity. *Proc. ACM Program. Lang.* 7, PLDI (June 2023). doi:10.1145/3591289

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM. doi:10.1145/1480881.1480885

Azadeh Farzan. 2023. Commutativity in Automated Verification. In *38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*. IEEE. doi:10.1109/LICS56636.2023.10175734

Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound sequentialization for concurrent program verification. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Ranjit Jhala and Isil Dillig (Eds.). ACM. doi:10.1145/3519939.3523727

Matthew Fluet, Mike Rainey, John H. Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: a heterogeneous parallel language. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP*, Neal Glew and Guy E. Blelloch (Eds.). ACM. doi:10.1145/1248648.1248656

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* 17, 3 (2021). https://arxiv.org/abs/2006.13635v3

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.* 5, POPL (Jan. 2021). doi:10.1145/3434291

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, Albert Cohen and Martin T. Vechev (Eds.). ACM. doi:10.1145/3062341.3062354

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (LIPIcs, Vol. 2)*, Ramesh Hariharan, Madhavan Mukund, and V. Vinay (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. doi:10.4230/LIPICS.FSTTCS.2008.1769

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* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

Ming Kawaguchi, Patrick Maxim Rondon, Alexander Bakst, and Ranjit Jhala. 2012. Deterministic parallelism via liquid effects. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM. doi:10.1145/2254064.2254071

Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP*, Paul Hudak and Stephanie Weirich (Eds.). ACM. doi:10.1145/1863543.1863582

Bernhard Kragl and Shaz Qadeer. 2021. The Civl Verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE. doi:10.34727/2021/ISBN.978-3-85448-046-4_23

Robbert Krebbers, Luko van der Maas, and Enrico Tassi. 2025. Inductive Predicates via Least Fixpoints in Higher-Order Separation Logic. In *16th International Conference on Interactive Theorem Proving (ITP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 352)*, Yannick Forster and Chantal Keller (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany. doi:10.4230/LIPIcs.ITP.2025.27

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM. doi:10.1145/3009837.3009877

Lindsey Kuper and Ryan R. Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, Clemens Grelck, Fritz Henglein, Umut A. Acar, and Jost Berthold (Eds.). ACM. doi:10.1145/2502323.2502326

Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014a. Taming the parallel effect zoo: extensible deterministic parallelism with LVish. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM. doi:10.1145/2594291.2594312

Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014b. Freeze after writing: quasi-deterministic parallel programming with LVars. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, Suresh Jagannathan and Peter Sewell (Eds.). ACM. doi:10.1145/2535838.2535842

John Launchbury and Simon Peyton Jones. 1995. State in Haskell. *LISP and Symbolic Computation* 8, 4 (1995), 293–341. http://dx.doi.org/10.1007/BF01018827

Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. *SIGPLAN Not.* 51, 1 (Jan. 2016). doi:10.1145/2914770.2837635

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975). doi:10.1145/361227.361234

Simon Marlow, Ryan Newton, and Simon L. Peyton Jones. 2011. A monad for deterministic parallelism. In *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell*, Koen Claessen (Ed.). ACM. doi:10.1145/2034675.2034685

Alexandre Moine, Sam Westrick, and Joseph Tassarotti. 2025. *All for One and One for All: Program Logics for Exploiting Internal Determinism in Parallel Programs (Artifact)*. doi:10.5281/zenodo.17259376

David A. Naumann. 2020. Thirty-Seven Years of Relational Hoare Logic: Remarks on Its Principles and History. In *9th International Symposium on Leveraging Applications of Formal Methods (Lecture Notes in Computer Science, Vol. 12477)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer. doi:10.1007/978-3-030-61470-6_7

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM. doi:10.1145/2491956.2462176

Sven-Bodo Scholz. 2003. Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.* 13, 6 (2003). doi:10.1017/S0956796802004458

Julian Shun and Guy E. Blelloch. 2014. Phase-concurrent hash tables for determinism. In *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, Guy E. Blelloch and Peter Sanders (Eds.). ACM. doi:10.1145/2612669.2612687

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, Chandra Krintz and Emery D. Berger (Eds.). ACM. doi:10.1145/2908080.2908092

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024a. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8, POPL (Jan. 2024). doi:10.1145/3632851

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024b. A Logical Approach to Type Soundness. *J. ACM* 71, 6 (Nov. 2024). doi:10.1145/3676954

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017). doi:10.1145/3158152

Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.* 3, POPL (2019). doi:10.1145/3290372

Received 2025-07-10; accepted 2025-11-06