DisLog: A Separation Logic for Disentanglement

ALEXANDRE MOINE, Inria, France
SAM WESTRICK, Carnegie Mellon University, USA
STEPHANIE BALZER, Carnegie Mellon University, USA

Disentanglement is a run-time property of parallel programs that facilitates task-local reasoning about the memory footprint of parallel tasks. In particular, disentanglement ensures that a task does not access any memory objects allocated by another concurrently executing task. For example, disentanglement is exploited to implement a high-performance parallel memory manager in the MPL (MaPLe) compiler for Parallel ML. Prior research on disentanglement has focused on the design of optimizations, either trusting the programmer to provide a disentangled program or relying on runtime instrumentation for detecting and managing entanglement. This paper provides the first static approach to verify that a program is disentangled: it contributes DisLog, a concurrent separation logic for disentanglement. DisLog enriches concurrent separation logic with the notions necessary for reasoning about the fork-join structure of parallel programs, allowing to verify that memory accesses are effectively disentangled. A large class of programs, including race-free programs, exhibit memory access patterns that are disentangled "by construction". To reason about these patterns, the paper distills from DisLog an almost standard concurrent separation logic, called DisLog+. In this high-level logic, no specific reasoning about memory accesses is needed: functional correctness proofs entail disentanglement. The paper illustrates the use of DisLog and DisLog+ on a range of case studies, including two different implementations of parallel deduplication via concurrent hashing. All our results are mechanized in the Coq proof assistant using Iris.

1 INTRODUCTION

Recent work has shown that parallel functional programming can deliver the same efficiency and scalability as imperative and procedural approaches. The key to this line of work is a memory property known as *disentanglement* [Arora et al. 2021, 2023; Guatto et al. 2018; Raghunathan et al. 2016; Westrick et al. 2022, 2020], which restricts parallel tasks to access data that was allocated "before" the task executed. This restriction enables tasks to allocate and garbage-collect memory locally and independently—that is, without synchronizing with other parallel tasks. Utilizing disentanglement, Arora et al. [2023] developed a provably efficient memory manager for functional programs which also provides full support for effects. All of this work is implemented in MPL ("maple"), an open-source[1] compiler for Parallel ML. In practice, MPL has been shown to be fast, scalable, and competitive with lower-level and imperative language implementations.

This line of work relies on disentanglement to ensure efficiency and scalability, and leaves the burden of reasoning about disentanglement to the programmer. A strength is that this burden can be removed for some programming tasks. That is, disentanglement can be guaranteed by construction by restricting mutation, in particular, by writing purely functional code. Programmers can also use pure libraries that are implemented under-the-hood with in-place updates for efficiency. For example, common parallel operations (such as `map`, `reduce`, `scan`, `filter`, etc.) can be implemented efficiently with mutable arrays, hidden behind a pure interface, and can then be used to write disentangled code. However, in this example, the developers of high-performance libraries still to reason carefully about disentanglement. More generally, whenever in-place updates and other low-level optimizations are necessary for efficiency, disentanglement needs to be taken into account.

In the context of in-place updates and other memory effects, reasoning about disentanglement is subtle. Programmers may wish to use concurrent data structures (e.g. lock-free hash tables) to

[1] https://github.com/mpllang/mpl

Authors' addresses: Alexandre Moine, Inria, Paris, France, alexandre.moine@inria.fr; Sam Westrick, Carnegie Mellon University, USA, swestric@cs.cmu.edu; Stephanie Balzer, Carnegie Mellon University, USA, balzers@cs.cmu.edu.

improve efficiency. Such data structures can be made disentangled [Westrick 2022], but reasoning about their correctness is challenging, even for experts. If disentanglement is violated, there can be significant consequences for performance, in terms of increased time and space usage [Arora et al. 2023]. In this sense, disentanglement can be considered a "safety" condition for performance-oriented code.

Therefore, we shift our attention to static verification of disentanglement. Our goal is to support reasoning about both high-level and low-level code, including atomic in-place updates and concurrent data structures, which can require identifying intricate invariants. In this setting, concurrent separation logic [Brookes 2007; O'Hearn 2007] and its modern variants [Jung et al. 2018; Nanevski et al. 2014] have proven to be successful vehicles for verifying safety and correctness properties of programs in the presence of challenging concurrent features [Chajed et al. 2021; Kaiser et al. 2017]. An intriguing question is whether or not separation logic can be used to prove disentanglement, which we address in this paper.

To verify disentanglement statically, we develop **DisLog**, the first program logic for proving disentanglement, and formally prove its soundness. At a high level, DisLog is a concurrent separation logic built on Iris [Jung et al. 2018] endowed with assertions which describe dependencies between parallel tasks and permissions to make disentangled loads from the heap. This approach makes the logic powerful enough to verify disentanglement even in complex and subtle situations, such as programs with lock-free data structures and algorithms using atomic in-place reads and writes.

Going further, on top of DisLog, we develop **DisLog+**, a standard concurrent separation logic which hides the details of disentanglement, allowing for standard separation logic proofs while also getting proofs of disentanglement for free. DisLog+ is applicable for a wide variety of programs, including purely functional programs, race-free programs, and even programs that have "benign" memory races (e.g., write-write races). Importantly, DisLog and DisLog+ work seamlessly together, allowing for DisLog+ proofs to drop into the more powerful DisLog where needed (e.g. for verifying non-pure segments of mostly pure programs), and otherwise stay at a high level of abstraction.

To evaluate DisLog and DisLog+, we consider several case studies, including key parallel primitives, as well as sophisticated parallel algorithms involving concurrent data structures. In all cases, we prove that the programs are disentangled. Our experience has shown that, using the logics developed in this paper, the effort of proving disentanglement is typically small. Furthermore, when a formal proof of functional correctness is desired, using DisLog+ often yields a proof of disentanglement for free.

Our contributions include:

- DisLog, the first program logic to verify that a program is disentangled (§ 4). It employs the notion of ***timestamps*** to reason about the nested fork-join parallelism of a program and introduces a novel ***clock*** assertion to prove that memory accesses do not cause entanglement.
- DisLog+, a high-level logic built on top of DisLog that shields the user from timestamp management (§ 5). As a result, race-free programs can be verified in DisLog+ with the standard reasoning rules of concurrent separation logic.
- Two mechanisms allowing to reason about benign races in DisLog+, including fractional write-only assertions for write-write races (§ 5.4), and a set of rules for read-write races on pre-allocated data (§ 5.5).
- A range of case studies (§ 6), including multiple parallel primitives, parallel lookup in a lazy collection, and two examples of deduplication via concurrent hashing.
- A formalization in the Coq proof assistant using Iris [Jung et al. 2018]. All our results are mechanized (§ 7) in Coq, including: the two program logics, their soundness theorems, and the case studies.

```
fun scratch () =
  let
    val shared = newScratchpad()
    val {tryLock, releaseLock} =
      newLock()
    fun myTask() =
      if tryLock() then
        (doWork shared;
         clearScratchpad shared;
         releaseLock())
      else
        let val x = newScratchpad()
        in doWork x
  in
    // two calls in parallel (could be
    // generalized to many calls if desired)
    (myTask() || myTask())
```

(a) The scratch function calls doWork multiple times in parallel, and provides a suitable scratchpad for each call.

```
fun newLock () =
  let val r = ref false
  in { tryLock = fn () ⇒ CAS(r,false,true),
       releaseLock = fn () ⇒ r := false }

type elem = ...
val defaultElem : elem = ...

type scratchpad = elem array
fun newScratchpad () =
  Array.allocate (N, defaultElem)
fun clearScratchpad scratch =
  for i from 0 to N − 1 do
    scratch[i] := defaultElem

fun doWork scratchpad =
  // ... read and write to scratchpad
```

(b) Auxiliary code for scratch, including locks, scratchpads, and a function doWork that requires a scratchpad as temporary space.

Fig. 1. The scratch example used to illustrate our approach.

2 KEY IDEAS

2.1 Background

Nested Fork-Join Parallelism. We consider programs written using a single parallel primitive, the parallel tuple $e_1 \,||\, e_2$. It executes e_1 and e_2 in parallel, and returns their results as a pair. There are two reductions to be distinguished: a ***fork***, when two child tasks are spawned to execute e_1 and e_2 in parallel, and a ***join***, when the child tasks complete and return their results as a pair. This style of programming is known as nested fork-join parallelism, as parallel tuples may be arbitrarily nested. In particular, programmers are able to write parallel recursive divide-and-conquer style algorithms. For example, a “parallel for-loop” can be implemented by splitting the index range in half and then recursively execute the two halves in parallel (§6.2).

Disentanglement. While the program executes, it allocates objects in memory and performs memory effects such as reads and writes on those objects. Disentanglement is a restriction on these effects that limits communication between parallel tasks. This restriction is determined by keeping track of which task allocated each object. Tasks may always access their own allocations. Additionally, each task may access any object allocated “before” the task being executed. The notion of “before” relates to the dependencies induced by forks and joins. A forking task comes before the two tasks it forks, and conversely, two joining tasks come before their join point. If a task ever acquires a pointer to an object allocated by some other task that is executing concurrently, this constitutes *entanglement.* **The logics developed in this paper allow proving that a program is disentangled**, i.e., that in every possible execution of the program, entanglement will never occur.

2.2 Running Example

To illustrate the ideas in the paper, we use a running example, called scratch, shown in Fig. 1. This function is disentangled and also non-deterministic. At a high level, scratch calls the function doWork two times in parallel. Each call to doWork uses an array (called a “scratchpad”) as temporary space. Note that it would be safe to allocate a fresh scratchpad for every call to doWork. The goal of the example is to optimize performance by reducing the number of scratchpads that are allocated. (The example only calls `doWork` twice, but this could be generalized to any number of calls in parallel, which would make the optimization more significant.)

To reduce the number of allocated scratchpads, `scratch` implements a simple strategy. First, a shared scratchpad is allocated together with a lock to protect it (Fig. 1, lines 3–5). Then, before each call to `doWork`, `scratch` will attempt to claim access to the shared scratchpad by calling `tryLock` (line 7). If this succeeds, then `doWork` may use the shared scratchpad (line 8); otherwise, `scratch` falls back on allocating a fresh scratchpad (line 12). Whenever a call to `doWork` is finished using the shared scratchpad, then the shared scratchpad is cleared (line 9), before finally releasing the lock (line 10). In this way, `scratch` reduces the number of allocations by reusing the shared scratchpad as much as possible. In particular, if `scratch` is executed using only a single processor, then every call to `doWork` will be able to use the shared scratchpad, and no additional scratchpads will be allocated.

The auxiliary code for the example is shown in Fig. 1b, which defines scratchpads and locks. The details of `doWork` are not important as long as it performs reads and writes on the scratchpad. Locks are implemented by a pair of closures with a mutable boolean, indicating whether the lock has been locked. The closure `tryLock` is implemented using an atomic compare-and-swap (CAS) which returns **true** if the CAS succeeds, and **false** otherwise.

Disentanglement in `scratch`*.* Proving that the `scratch` example is disentangled is subtle. In particular, `doWork` may read or write to the `scratchpad`, and we need to show that reading from the `scratchpad` will never return a value allocated by a concurrent task. Thankfully, the `scratch` function guarantees a strong precondition: when `doWork` begins, the argument `scratchpad` will contain only the value `defaultElem`, which is allocated before every call to `doWork`, and therefore is safe with respect to disentanglement.

The precondition on `doWork` is easily satisfied on line 13, because the scratchpad is freshly allocated. Showing that the precondition is also satisfied on line 8 is more subtle, because there is an invariant on the `shared` scratchpad which is determined by the state of the lock. Informally, the invariant is: "while the lock is not held, for every i, `shared[i] = defaultElem`." This invariant is re-established by calling `clearScratchpad` (line 9) before releasing the lock. Note that removing this call to `clearScratchpad` may lead to an entangled state. Indeed, after a call to `doWork`, the scratchpad may contain locally-allocated data, which is hence available for the other task.

2.3 Disentanglement: Timestamps, Hazardous Loads and How to Reason about Them

Partial orders on tasks through timestamps. In Sec. 3, we present a semantics enforcing disentanglement. Our approach is to (1) assign every parallel task a unique identifier called a *timestamp*, (2) assign every heap-allocated object a timestamp, marking *when* the object was allocated, and (3) restrict every task to only depend on objects allocated at timestamps that come *before* their timestamp. Timestamps form a partial order which respects the dependencies induced by forks and joins. When a task forks, the semantics generates two (one for each task) timestamps that are preceded by the timestamp of the forking task. Conversely, when two tasks join, the semantics generates a timestamp that is preceded by both timestamps of the two joining tasks. Forks and joins are the only operations extending the partial order of timestamps. Tasks otherwise step independently.

Fig. 2 visualizes one possible execution of the running example. Each shaded oval represents a task, and is labeled by its timestamp t_i. The framed content is not relevant yet. Execution begins on a task t_1, which allocates the scratchpad and the lock. Then, a fork occurs, generating two tasks t_2 and t_3. The task t_2 fails to acquire the lock, whereas the task t_3 succeeds. After completing, the two tasks join, forming a new task t_4.

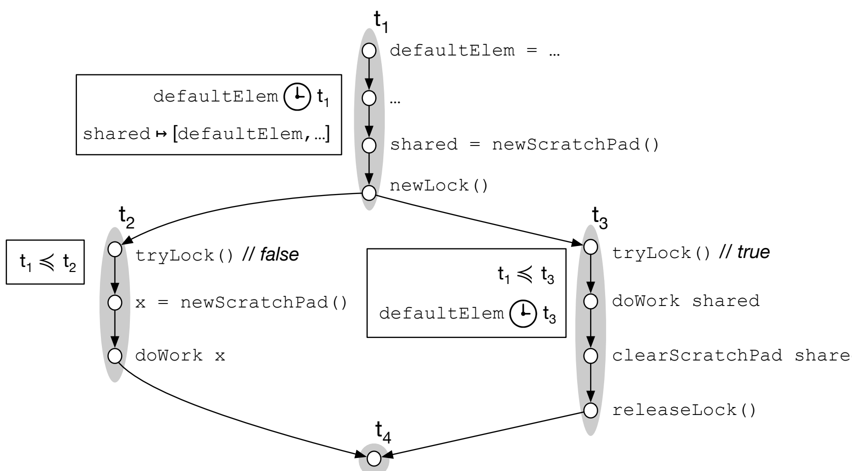

Fig. 2. One possible execution of the example of Fig. 1

A program logic for timestamp orders. In Sec. 4, we blend timestamps in DisLog by associating expressions with a *current* timestamp and an *end* timestamp. The program logic is built around a *weakest precondition* (WP) modality which takes the form

$$\mathsf{wp}\ \langle t,\ e\rangle\ \{\lambda\, t'\, v.\ \Phi\}$$

asserting that the expression e is currently evaluated on a task at timestamp t, that e is disentangled and can reduce, and if its reduction terminates, then it does so at end timestamp t', yielding a value v such that Φ holds. The partial order of timestamps is encoded into the assertions of the logic, via the precedence assertion $t \preccurlyeq t'$. The precedence assertion is *persistent* and hence duplicable at will. This assertion describes the parallel structure of the program being verified.

Examples of the precedence assertion $t \preccurlyeq t'$ appear in the framed boxes of Fig. 2. To reason about the task t_2, the user gets an assertion $t_1 \preccurlyeq t_2$. Dually, the user gets an assertion $t_1 \preccurlyeq t_3$ to reason about the task t_3. At the join point t_4, the user gets the assertions $t_2 \preccurlyeq t_4$ and $t_3 \preccurlyeq t_4$. Making use of the fact the $\preccurlyeq$ is a pre-order, the user can use transitivity and deduce for example that $t_1 \preccurlyeq t_4$.

Preserving disentanglement. Acquiring a memory location ℓ from the heap is an entanglement hazard. Disentanglement is preserved if and only if ℓ was allocated at some timestamp preceding the timestamp t of the acquiring task—in that case, we say that ℓ was allocated before t. To represent such a requirement, we introduce the clock assertion, written $\ell \circledcirc t$, which precisely asserts that the ℓ was allocated before t. This assertion appears for example in the precondition of DisLog's Load rule (§4.3). To illustrate this rule, we show a specialized instantiation allowing to load the element at index 0 in the shared scratchpad, here named ℓ. The premises are implicitly separated by the separating conjunction $*$.

$$\textsc{SpecializedLoad}\ \frac{\mathsf{shared} \mapsto [\ell; \ldots] \qquad \ell \circledcirc t}{\mathsf{wp}\ \langle t,\ \mathsf{shared}[0]\rangle\ \{\lambda\, t'\, v.\ \ulcorner t' = t \wedge v = \ell \urcorner\ *\ \mathsf{shared} \mapsto [\ell; \ldots]\ \}}$$

The SpecializedLoad rule first requires, as in standard separation logic, ownership of the shared location via a points-to assertion. Crucially, this rule also requires that the loaded value ℓ was allocated before the current timestamp t via the $\ell \circledcirc t$ assertion. In the postcondition of the WP, the rule asserts that the end timestamp t' is equal to the previous timestamp t, the returned value v is ℓ, and the user still has the points-to ownership.

The clock assertion is persistent, giving the user great flexibility. Moreover, the crux of our approach is that *the clock assertion is monotonic with respect to the precedence pre-order*. Hence, if the user knows that the location ℓ was allocated before t and that t precedes t' they can then deduce ℓ was allocated before t'. This mechanism is illustrated in Fig. 2. Indeed, the user can produce an assertion `defaultElem` $\circlearrowright t_1$ upon the allocation of `defaultElem`. Then, while reasoning about t_3 the user can use the assertion $t_1 \preccurlyeq t_3$ to generate a new clock assertion `defaultElem` $\circlearrowright t_3$.

2.4 Going High-Level: Simple Programs Should Have Simple Proofs

Readers familiar with proofs of realistic programs may be worried by timestamps polluting the logic, and the additional proof burden on a common rule such as Load. In practice, many programs "don't poke the bear" and subtle reasoning about timestamps should not be needed. For example, Westrick et al. [2020] show that race-free programs are always disentangled, as they never read hazardous shared locations. Verifying such programs should be as cheap as a standard separation logic proof.

In Sec. 5, we present DisLog+, a high-level separation logic logic where timestamps, clocks, and precedence are confined to very few occurrences. DisLog+ allows reasoning on race-free programs with the standard reasoning rules of concurrent separation logic. The sole difference is a restriction on ghost state, effectively preventing races (§ 5.3). What is the secret of the DisLog+ logic? The key observation we make on race-free programs is that (1) the content of a freshly allocated location is always safe to acquire for the allocating task and (2) "being safe to acquire" is a *monotonic* property: if an object is safe to acquire for a given task, it is safe to acquire for every subsequent task. As long as a program does not write carelessly to a shared location and breaks monotonicity, objects are always safe to acquire and no reasoning about timestamps is needed.

Technically, we define assertions of DisLog+ as monotonic predicates of DisLog over an ambient timestamp, the latter being implicitly threaded through during the proof. This definition takes inspiration from a technical mechanism introduced by work on weak-memory models [Kaiser et al. 2017; Mével et al. 2020]. Points-to assertions of DisLog+ store not only ownership information but also the proof that all pointed objects are safe to acquire. Hence, DisLog+ provides a standard Load reasoning rule. We stress that DisLog+ is a light abstraction over DisLog. At any moment during the proof, the user of DisLog+ can fall back to DisLog for fine timestamp-related reasoning.

Beyond race freedom. Whereas we just explained how DisLog+ can be used to reason about race-free programs, can we also provide high-level reasoning rules for the most elementary disentangled races? We answer with the positive and provide two new reasoning mechanisms. The first one consists of *fractional write-only assertions* (§ 5.4) allowing the user to reason about write-write races within DisLog+. As a write-write race does not acquire any memory, such a race is always disentangled. The second one consists of a set of rules unveiling just enough timestamps to reason about races on "obviously safe" data (§ 5.5). These data include data that was allocated before the beginning of the parallel phase, and *unboxed* data—that is, data that is not allocated in the heap.

The language we model supports the atomic operation compare-and-swap (CAS). A CAS is an entanglement hazard. Indeed, a CAS acquires the scrutinized value, which must be safe to acquire. In the `scratch` running example (Fig. 1b), we use CAS to implement a spin-lock. Here, we exploit unboxed data to allow parallel tasks to safely communicate via a race on shared reference r. A "race on unboxed data" should ring a bell: it perfectly fits in the realm of DisLog+ and its extensions. We show in Sec. 6 how to reason on our locks and `scracth` entirely within the high-level DisLog+.

3 LANGUAGE AND SEMANTICS

Our language, DisLang, is an imperative lambda-calculus with fork-join parallelism. We equip DisLang with a small-step, substitution-based, call-by-value semantics guaranteeing disentanglement.

$$
\begin{array}{lll}
\text{Values } \mathcal{V} & v, w ::= () \mid b \in \{\text{true}, \text{false}\} \mid i \in \mathbb{Z} \mid \ell \in \mathcal{L} \mid \hat{\mu} f.\, \lambda \vec{x}.\, e \text{ where } fv(e) \subseteq (\{f\} \cup \vec{x}) \\
\text{Blocks} & r ::= \vec{w} \mid \mu f.\, \lambda \vec{x}.\, e \\
\text{Primitives} & \bowtie\, ::= + \mid - \mid \times \mid \div \mid \text{mod} \mid == \mid < \mid \leq \mid > \mid \geq \mid \vee \mid \wedge
\end{array}
$$

$$
\begin{array}{llllll}
\text{Expressions} & e ::= v & \textit{value} & \text{alloc } e\ e & \textit{array allocation} \\
& \quad x & \textit{variable} & e[e] & \textit{array load} \\
& \quad \text{let } x = e \text{ in } e & \textit{sequencing} & e[e] \leftarrow e & \textit{array store} \\
& \quad \text{if } e \text{ then } e \text{ else } e & \textit{conditional} & \text{length } e & \textit{array length} \\
& \quad e\ \vec{e} & \textit{call} & e \mid\mid e & \textit{parallel tuple} \\
& \quad e \bowtie e & \textit{primitive operation} & \text{CAS } e\ e\ e\ e & \textit{compare-and-swap} \\
& \quad \mu f.\, \lambda \vec{x}.\, e & \textit{closure allocation} & &
\end{array}
$$

$$
\begin{array}{llllll}
\text{Contexts} & K ::= \text{let } x = \square \text{ in } e & \mid \text{if } \square \text{ then } e \text{ else } e & \mid \text{alloc } \square\ e & \mid \text{alloc } v\ \square & \mid \text{length } e \\
& \quad \mid \square[e] & \mid v[\square] & \mid \square[e] \leftarrow e & \mid v[\square] \leftarrow e & \mid v[v] \leftarrow \square \\
& \quad \mid \square \bowtie e & \mid v \bowtie \square & \mid \square\ \vec{t} & \mid v\ (\vec{v} \mathbin{+\!\!+} \square \mathbin{+\!\!+} \vec{t}) & \\
& \quad \mid \text{CAS } \square\ t\ t\ t & \mid \text{CAS } v\ \square\ t\ t & \mid \text{CAS } v\ v\ \square\ t & \mid \text{CAS } v\ v\ v\ \square &
\end{array}
$$

Fig. 3. Syntax of DisLang

3.1 Syntax

The syntax of DisLang appears in Fig. 3. A value $v \in \mathcal{V}$ can be the unit value (), a boolean $b \in \{\text{true}, \text{false}\}$, an idealized integer $i \in \mathbb{Z}$, a *memory location* $\ell \in \mathcal{L}$, where $\mathcal{L}$ is an infinite set of locations, or a *top-level function* $\hat{\mu} f.\, \lambda \vec{x}.\, e$. A top-level function is closed in the sense that the only variables available in the function body e are the function's name f and the formal parameters $\vec{x}$.

A *block* describes the contents of a heap cell, amounting to either an array of values, written $\vec{w}$, or a *λ-abstraction* $\mu f.\, \lambda \vec{x}.\, e$. Lambdas, as opposed to top-level functions $\hat{\mu} f.\, \lambda \vec{x}.\, e$, are not values. Instead, they are compiled to heap-allocated *closures* [Appel 1992; Landin 1964]. Hence, acquiring a lambda can create entanglement. Top-level functions can be seen as closures that are pre-allocated outside the heap, which thus cannot create entanglement. In DisLang, fork-join parallelism is available via the parallel tuple $e_1 \mid\mid e_2$, representing the expressions e_1 and e_2 to be computed in parallel. DisLang supports a compare-and-swap instruction CAS $e\ e\ e\ e$, which targets an array, and is parameterized by 4 arguments: the location of the array, the offset in the array, the old value and the new value. The syntax of evaluation contexts describes a left-to-right call-by-value evaluation.

3.2 Computation Graphs and Disentanglement

The dynamics of DisLang, presented in the next section, makes use of a *computation graph*, capturing the nested fork-join parallel structure of a program. A computation graph is a directed acyclic graph where vertices, or *tasks*, represent sequential computations, and edges represent the dependencies between them [Acar et al. 2016]. We label each task with a unique *timestamp* t, from an infinite set $\mathcal{T}$. When a task t_0 forks two fresh children t_1 and t_2, the computation graph is extended with edges (t_0, t_1) and (t_0, t_2). Conversely, when two completed tasks t_1 and t_2 join to form a fresh task t_3 the computation graph is extended with edges (t_1, t_3) and (t_2, t_3). As discussed earlier, an example computation graph for the `scratch` example (§2) is shown in Fig. 2.

In a computation graph G, we say that t *precedes* t' and write $t \preccurlyeq_G t'$ when there exists a sequence of edges in G from t to t'. In particular, we say that two tasks are *concurrent* when neither precedes the other. Entanglement occurs when a task acquires a location that was allocated by a concurrent task. Recall our running example (§2, Fig. 1): a particular implementation of `doWork` can return locally-allocated data in the shared scratchpad. If it were possible for two concurrent tasks to both win the lock, without proper cleaning of the scratchpad, this locally-allocated data could be acquired by the concurrent task, generating entanglement.

$$\textsc{HeadAlloc}\quad \frac{0 \le n \qquad \ell \notin \mathsf{dom}(\sigma) \qquad \ell \notin \mathsf{dom}(\alpha)}{G, t \vdash \sigma \setminus \alpha \setminus \mathsf{alloc}\ v\ n \longrightarrow [\ell := v^n]\sigma \setminus [\ell := t]\alpha \setminus \ell}$$

$$\textsc{HeadCallPrim}\quad \frac{v_1 \bowtie v_2 \xrightarrow{\mathsf{pure}} v}{G, t \vdash \sigma \setminus \alpha \setminus v_1 \bowtie v_2 \longrightarrow \sigma \setminus \alpha \setminus v}$$

$$\textsc{HeadClosure}\quad \frac{\ell \notin \mathsf{dom}(\sigma) \qquad \ell \notin \mathsf{dom}(\alpha)}{G, t \vdash \sigma \setminus \alpha \setminus \mu f.\lambda \vec{x}.e \longrightarrow [\ell := \mu f.\lambda \vec{x}.e]\sigma \setminus [\ell := t]\alpha \setminus \ell}$$

$$\textsc{HeadLength}\quad \frac{\sigma(\ell) = \vec{w}}{G, t \vdash \sigma \setminus \alpha \setminus \mathsf{length}\ \ell \longrightarrow \sigma \setminus \alpha \setminus |\vec{w}|}$$

$$\textsc{HeadStore}\quad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}|}{G, t \vdash \sigma \setminus \alpha \setminus \ell[i] \leftarrow v \longrightarrow [\ell := [i := v]\vec{w}]\sigma \setminus \alpha \setminus ()}$$

$$\textsc{HeadLetVal}\quad G, t \vdash \sigma \setminus \alpha \setminus \mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow \sigma \setminus \alpha \setminus [v/x]e$$

$$\textsc{HeadIfTrue}\quad G, t \vdash \sigma \setminus \alpha \setminus \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \sigma \setminus \alpha \setminus e_1$$

$$\textsc{HeadIfFalse}\quad G, t \vdash \sigma \setminus \alpha \setminus \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \sigma \setminus \alpha \setminus e_2$$

$$\textsc{HeadLoad}\quad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v \qquad (v \in \mathcal{L} \implies \alpha(v) \preccurlyeq_G t)}{G, t \vdash \sigma \setminus \alpha \setminus \ell[i] \longrightarrow \sigma \setminus \alpha \setminus v}$$

$$\textsc{HeadCall}\quad \frac{(v = \hat{\mu} f.\lambda \vec{x}.e) \lor (v \in \mathcal{L} \land \sigma(v) = \mu f.\lambda \vec{x}.e) \qquad |\vec{x}| = |\vec{w}| \qquad (\forall \ell.\ \ell \in locs(e) \implies \alpha(\ell) \preccurlyeq_G t)}{G, t \vdash \sigma \setminus \alpha \setminus v\ \vec{w} \longrightarrow \sigma \setminus \alpha \setminus [v/f][\vec{w}/\vec{x}]e}$$

$$\textsc{HeadCASSucc}\quad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v_0 \qquad v_0 = v \qquad (v_0 \in \mathcal{L} \implies \alpha(v_0) \preccurlyeq_G t)}{G, t \vdash \sigma \setminus \alpha \setminus \mathsf{CAS}\ \ell\ i\ v\ v' \longrightarrow [\ell := [i := v']\vec{w}]\sigma \setminus \alpha \setminus \mathsf{true}}$$

$$\textsc{HeadCASFail}\quad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v_0 \qquad v_0 \ne v \qquad (v_0 \in \mathcal{L} \implies \alpha(v_0) \preccurlyeq_G t)}{G, t \vdash \sigma \setminus \alpha \setminus \mathsf{CAS}\ \ell\ i\ v\ v' \longrightarrow \sigma \setminus \alpha \setminus \mathsf{false}}$$

Fig. 4. Head reduction. The disentanglement proof obligation is highlighted.

3.3 Operational Semantics

Head Reduction. Fig. 4 defines the head reduction relation $G, t \vdash \sigma \setminus \alpha \setminus e \longrightarrow \sigma' \setminus \alpha' \setminus e'$ between two *head configurations* $\sigma \setminus \alpha \setminus e$ and $\sigma' \setminus \alpha' \setminus e'$, where G is the (global) computation graph and t the timestamp of the (local) task at which the reduction takes place. A head configuration consists of the expression e being evaluated, the *store* σ, and an *allocation map* α. A store σ is a finite map of locations to blocks, representing the heap, and an allocation map α is a finite map of locations to timestamps, recording the timestamps at which locations were allocated.

We write $\sigma(\ell)$ to denote the block stored at the location ℓ in the store σ. To insert a block into the store or update the store, we write $[\ell := r]\sigma$. Note that only arrays can be updated; closures are immutable. To refer to the index i of an array $\vec{w}$, we write $\vec{w}(i)$, and to update an array, we write $[i := v]\vec{w}$. We similarly write $[\ell := t]\alpha$ for an insertion in the allocation map. We write v^n for an array of length n, where each element of the array is initialized with the value v.

The HeadAlloc and HeadClosure reductions allocate heap blocks, arrays and closures, respectively, extending the store with the desired block and the allocation map with the current timestamp. The HeadCallPrim reduction encompasses a reduction $\xrightarrow{\mathsf{pure}}$ to compute a primitive operation. The HeadStore reduction updates the field of an array, and the HeadLength reduction returns the length of an array. The HeadLetVal reduction substitutes a variable by its value. The HeadIfTrue and HeadIfFalse reductions reduce an if-then-else construction where the conditional is evaluated.

Entanglement may only occur when a task acquires a location. Locations are acquired during the reductions HeadLoad, HeadCall, HeadCASSucc and HeadCASFail. A load acquires the

$$\textsc{SchedHead}\quad \frac{G, t \vdash \sigma \setminus \alpha \setminus e \longrightarrow \sigma' \setminus \alpha' \setminus e'}{\sigma / \alpha / G / t / e \xrightarrow{\text{sched}} \sigma' / \alpha' / G / t / e'}$$

$$\textsc{SchedFork}\quad \frac{t_1, t_2 \notin \text{leaves}(G) \qquad G' = G \cup \{(t_0, t_1), (t_0, t_2)\}}{\sigma / \alpha / G / t_0 / e_1 \,||\, e_2 \xrightarrow{\text{sched}} \sigma / \alpha / G' / t_1 \otimes t_2 / e_1 \,||\, e_2}$$

$$\textsc{SchedJoin}\quad \frac{t_3 \notin \text{leaves}(G) \qquad \sigma' = [\ell := [v_1; v_2]]\sigma \qquad \alpha' = [\ell := t_3]\alpha \qquad G' = G \cup \{(t_1, t_3), (t_2, t_3)\}}{\sigma / \alpha / G / t_1 \otimes t_2 / v_1 \,||\, v_2 \xrightarrow{\text{sched}} \sigma' / \alpha' / G' / t_3 / \ell}$$

$$\textsc{StepSched}\quad \frac{\sigma / \alpha / G / T / e \xrightarrow{\text{sched}} \sigma' / \alpha' / G' / T' / e'}{(\sigma, \alpha, G) / T / e \xrightarrow{\text{step}} (\sigma', \alpha', G') / T' / e'}$$

$$\textsc{StepBind}\quad \frac{S / T / e \xrightarrow{\text{step}} S' / T' / e'}{S / T / K[e] \xrightarrow{\text{step}} S' / T' / K[e']}$$

$$\textsc{StepParL}\quad \frac{S / T_1 / e_1 \xrightarrow{\text{step}} S' / T_1' / e_1'}{S / T_1 \otimes T_2 / e_1 \,||\, e_2 \xrightarrow{\text{step}} S' / T_1' \otimes T_2 / e_1' \,||\, e_2}$$

$$\textsc{StepParR}\quad \frac{S / T_2 / e_2 \xrightarrow{\text{step}} S' / T_2' / e_2'}{S / T_1 \otimes T_2 / e_1 \,||\, e_2 \xrightarrow{\text{step}} S' / T_1 \otimes T_2' / e_1 \,||\, e_2'}$$

Fig. 5. Reduction under a context and parallelism

indexed value, a call the environment of the closure, and a CAS the scrutinized value. The HeadCall reduction distinguishes between invoking a top-level function and closure. Only the latter possibly acquires locations. To prevent entanglement, all these rules include the same kind of precondition (highlighted in Fig. 4): if ℓ is acquired, then its allocation timestamp $\alpha(\ell)$ must precede the timestamp t of the task at which the reduction takes place. These preconditions amount to a *proof obligation* during the verification of a program. Verified programs will satisfy the obligation and will thus never get stuck. As we will see in Sec. 4.4 and in Sec. 5.3, soundness of both of our logics entail the invariant that the physical program state are always disentangled.

Parallelism and Reduction under a Context. To keep track of the currently active and suspended tasks of an executing parallel program, we follow Westrick et al. [2020] and enrich the semantics with an auxiliary structure called a *task tree*, written T, of the following formal grammar: $T \triangleq t \in \mathcal{T} \mid T \otimes T$. A leaf represents an active task and is denoted by its timestamp t. A node $T_1 \otimes T_2$ represents a suspended task that has forked two parallel computations, recursively described by the task trees T_1 and T_2.

Taking advantage of task trees, we define the semantics of parallel reductions and reductions under a context in Fig. 5. We define a scheduling reduction $\sigma / \alpha / G / T / e \xrightarrow{\text{sched}} \sigma' / \alpha' / G' / T' / e'$ as either a head step, a fork, or a join. In this reduction relation, σ is a store, α an allocation map, G a computation graph, T a task tree, and e an expression. The SchedHead reduction describes a head reduction. The SchedFork reduction describes a fork: the task tree is at a leaf t_0 and faces a parallel tuple. The reduction generates two fresh timestamps t_1 and t_2, adds the corresponding edges to the computation graph and updates the task tree to the node with two leaves $t_1 \otimes t_2$. The SchedJoin reduction describes a join: the task tree is at a node with two leaves $t_1 \otimes t_2$, and both leaves reached a value. The reduction generates a fresh timestamp t_3, updates the computation graph, and allocates a memory cell to store the result of the parallel tuple. It then updates the task tree to the leaf t_3.

The main reduction relation $S / T / e \xrightarrow{\text{step}} S' / T' / e'$ describes a scheduling reduction inside the whole parallel program. A tuple $S / T / e$ consists of the program state S, the task tree T, and an expression e. A state S consists of the tuple (σ, α, G), denoting a store σ, an allocation map α, and a computation graph G. The StepSched reduction describes a scheduling step. The other reductions describe *where* the scheduling reduction takes place. The StepBind reduction describes a reduction under an evaluation context. The StepParL and StepParR reductions unveil the non-determinism of the parallel reduction. If a node of the task tree is encountered facing a parallel tuple, the left side or the right side can reduce.

4 DISLOG, A PROGRAM LOGIC FOR DISENTANGLEMENT

In this section, we present the details of DisLog. We first give an Iris primer and explain our notations (§ 4.1). Then, we showcase how timestamps appear in the program logic (§ 4.2), and present other reasoning rules (§ 4.3). Finally, we discuss the soundness theorem of DisLog (§ 4.4).

4.1 Assertions and Weakest Preconditions

We build DisLog on top of Iris [Jung et al. 2018], adopting Iris' syntax. In particular, we write Φ for an Iris assertion (of type *iProp*), $\Phi * \Phi'$ for a separating conjunction, and $\Phi \mathrel{-\!\!*} \Phi'$ for a separating implication. If U is a proposition of the meta logic, we call U *pure* and write $\ulcorner U \urcorner$. We write $\Phi \dashv\vdash \Phi'$ for the equivalence of assertions.

Our program logic features a weakest precondition (WP) modality which takes the form:

$$\mathsf{wp}\,\langle t,\ e\rangle\,\{\lambda\, t'\, v.\ \Phi\}$$

This modality adapts a standard Iris' WP to the semantics of DisLang, and in particular, enriches it to account for timestamps. In the above assertion, t is the timestamp of the task which symbolically executes the expression e. We call this timestamp the *current timestamp* of the expression. A postcondition takes the form $\lambda\, t'\, v.\ \Phi$ where the variables t' and v are bound in Φ. The variable v denotes the resulting value and the variable t' the *end* timestamp, the timestamp of the returning task. We write $\mathsf{wp}\,\langle t,\ e\rangle\,\{\lambda\, t'\, \ell.\ \Phi\}$, where the variable ℓ denotes a location, as a syntax sugar for $\mathsf{wp}\,\langle t,\ e\rangle\,\{\lambda\, t'\, v.\ \exists \ell.\ \ulcorner v = \ell \urcorner\ *\ \Phi\}$. We similarly do so for booleans b and integers i. If we want to abstract over the details of the postcondition, we write Ψ instead of $\lambda\, t'\, v.\ \Phi$.

Our WP is subject to the standard structural rules of separation logic. DisLog supports in particular the Frame rule that we present below, as a warm-up to our notations. We write reasoning rules as inference rules, where premises are separated by the separating conjunction $*$ and entail the conclusion. In particular, if the conclusion is a WP, premises amount to preconditions.

$$\textsc{Frame}\ \frac{\Phi_0 \qquad \mathsf{wp}\,\langle t,\ e\rangle\,\{\lambda\, t'\, v.\ \Phi_1\}}{\mathsf{wp}\,\langle t,\ e\rangle\,\{\lambda\, t'\, v.\ \Phi_0\ *\ \Phi_1\}}$$

Separation logic triples can be obtained with the standard definition $\{\Phi\}\,\langle t,\ e\rangle\,\{\Psi\} \triangleq \Box(\Phi \mathrel{-\!\!*} \mathsf{wp}\,\langle t,\ e\rangle\,\{\Psi\})$, where $\Box$ stands for the persistence modality of Iris. The persistence modality characterize *persistent* assertions (an assertion Φ is persistent when $\Phi \dashv\vdash \Box\,\Phi$). Once a persistent assertion holds, it holds forever. In particular, persistent assertions are duplicable.

Iris features *ghost state*, which is hence available in DisLog. We write $\Phi \Rrightarrow \Phi'$ for a *ghost update* (or *fancy update*) that updates the ghost state. We omit the so-called *masks* for the sake of readability. Thanks to the ghost state, DisLog supports Iris *invariants* [Jung et al. 2018, §2.2], with a standard interface. Our WP allows the user to assume (or *open*) an invariant before reasoning about an *atomic* expression and generates an obligation to restore (or *close*) the invariant in the postcondition. An atomic expression is an expression that can reduce to a value in a single head step of computation. We syntactically characterize such assertions with the Atomic e pure predicate.

DisLog makes use of fractional [Bornat et al. 2005; Boyland 2003] and discardable [Vindum and Birkedal 2021] points-to assertions of the form $\ell \mapsto_p \vec{w}$, where p denotes either a positive fraction less than or equal to 1, or a discarded fraction written $\Box$. The latter makes the points-to assertion persistent. When $p = 1$ we write $\ell \mapsto \vec{w}$. Points-to assertions of DisLog do not carry information about timestamps: this role is devoted to two new assertions described in the next Section.

$$\textsc{ClockMono}\quad \frac{\ell \circleddash t \quad t \preccurlyeq t'}{\ell \circleddash t'} \qquad \textsc{PrecRefl}\quad t \preccurlyeq t \qquad \textsc{PrecTrans}\quad \frac{t \preccurlyeq t' \quad t' \preccurlyeq t''}{t \preccurlyeq t''} \qquad \textsc{MementoPre}\quad \frac{\ulcorner \ell \in locs(e) \urcorner \quad \ell \circleddash t \twoheadrightarrow \mathrm{wp}\,\langle t,\, e\rangle\,\{\Psi\}}{\mathrm{wp}\,\langle t,\, e\rangle\,\{\Psi\}}$$

$$\textsc{MementoPost}\quad \frac{\mathrm{wp}\,\langle t,\, e\rangle\,\{\lambda\, t'\, v.\, v \circleddash t' \twoheadrightarrow \Psi\, t' v\}}{\mathrm{wp}\,\langle t,\, e\rangle\,\{\Psi\}} \qquad \textsc{TempusFugit}\quad \frac{\mathrm{wp}\,\langle t,\, e\rangle\,\{\lambda\, t'\, v.\, t \preccurlyeq t' \twoheadrightarrow \Psi\, t' v\}}{\mathrm{wp}\,\langle t,\, e\rangle\,\{\Psi\}} \qquad \textsc{TempusAtomic}\quad \frac{\ulcorner \mathrm{Atomic}\, e \urcorner \quad \mathrm{wp}\,\langle t,\, e\rangle\,\{\lambda\, \_\, v.\, \Psi\, t\, v\}}{\mathrm{wp}\,\langle t,\, e\rangle\,\{\Psi\}}$$

Fig. 6. Reasoning rules for clocks and precedence assertions

4.2 Timestamps Management

A central aspect of our disentanglement logic is the management of timestamps. To this end, DisLog features two new assertions.

- The *clock* assertion $\ell \circleddash t$, indicating that the location ℓ was allocated before the timestamp t in the underlying computation graph.
- The *precedence* assertion $t \preccurlyeq t'$, witnessing that the timestamp t precedes the timestamp t' in the underlying computation graph.

Both assertions are persistent and work hand-in-hand. Given the assertion $\ell \circleddash t$, a task at timestamp t can safely acquire the location ℓ. Moreover, given both the assertions $\ell \circleddash t$ and $t \preccurlyeq t'$, a task at timestamp t' can safely acquire the location ℓ as well. A benefit of phrasing a location's allocation timestamp relative to another timestamp, rather than absolute, is that the user never needs to know precisely at which timestamp a location was allocated: disentanglement is ensured as soon as the acquired location was allocated by a preceding task. Similarly, the user never needs to know the whole computation graph: precedence information suffices for proving disentanglement. We overload the clock assertion to arbitrary values v and introduce assertions of the form $v \circleddash t$. If v is a location ℓ, then this assertion is defined as $\ell \circleddash t$. Otherwise, it is defined as $\ulcorner True \urcorner$. We overload again this assertion to a collection of values, and write $\vec{w} \circleddash t$ the iterated conjunction $\mathop{\ast}_{(v \in \vec{w})} (v \circleddash t)$.

Fig. 6 summarizes the rules governing the clock and the precedence assertions. The ClockMono rule illustrates the monotonicity of the clock predicate with respect to the precedence pre-order: if the location ℓ was allocated before t and t precedes t', then it is safe to conclude that ℓ was allocated before t'. We emphasize that the precedence assertion forms a pre-order: this assertion is reflexive (PrecRefl) and transitive (PrecTrans). The MementoPre and MementoPost rules are the only rules generating a clock predicate. The MementoPre rule asserts that if the location ℓ occurs in the expression e at current timestamp t, then the user can gain a witness $\ell \circleddash t$ that ℓ was allocated at a timestamp preceding t. The MementoPost rule asserts that the value returned by a task was allocated before the end timestamp of this task.

The TempusFugit rule distills the semantics of DisLang: it is safe to suppose that the current timestamp precedes the end timestamp. The TempusAtomic rule asserts that the current timestamp and the end timestamp of an atomic expression are the same. The TempusAtomic rule is more precise than needed: the clock predicate and the precedence predicate are both monotonic with respect to the precedence pre-order, via the ClockMono rule and the PrecTrans rule, respectively. However, the TempusAtomic rule relieves the user from the burden of always applying the ClockMono and PrecTrans rules by hand when the timestamp is effectively preserved.

4.3 Reasoning Rules for Expressions

Fig. 7 gives the syntax-directed reasoning rules of DisLog (we hide "later" modalities for brevity). The rules Alloc, Length, CallPrim, LetVal, IfTrue, IfFalse, and Store are standard, apart from their mention of timestamps. In particular, the Alloc rule does not generate a clock assertion. If desired, such an assertion can be obtained by applying the MementoPost rule.

$$\textsc{Value}\quad \frac{\Psi\, t\, v}{\mathsf{wp}\,\langle t,\ v\rangle\,\{\Psi\}}$$

$$\textsc{Alloc}\quad \frac{\ulcorner 0 \le n \urcorner}{\mathsf{wp}\,\langle t,\ \mathsf{alloc}\ n\ v\rangle\,\{\lambda\_\ell.\ \ell \mapsto v^n\}}$$

$$\textsc{Load}\quad \frac{\ulcorner 0 \le i < |\vec{w}| \ \wedge\ \vec{w}(i) = v \urcorner \qquad \ell \mapsto_p \vec{w} \qquad v \circleddash t}{\mathsf{wp}\,\langle t,\ \ell[i]\rangle\,\{\lambda\_v'.\ \ulcorner v' = v \urcorner \ *\ \ell \mapsto_p \vec{w}\}}$$

$$\textsc{Closure}\quad \mathsf{wp}\,\langle t,\ \mu f.\,\lambda\vec{x}.\,e\rangle\,\{\lambda\_\ell.\ \mathsf{Func}\,\ell\,f\,\vec{x}\,e\}$$

$$\textsc{TopLevel}\quad \frac{\ulcorner v = \hat{\mu} f.\,\lambda\vec{x}.\,e \urcorner}{\mathsf{Func}\,v\,f\,\vec{x}\,e}$$

$$\textsc{Length}\quad \frac{\ell \mapsto_p \vec{w}}{\mathsf{wp}\,\langle t,\ \mathsf{length}\,\ell\rangle\,\{\lambda\_i.\ \ulcorner i = |\vec{w}| \urcorner \ *\ \ell \mapsto_p \vec{w}\}}$$

$$\textsc{CallPrim}\quad \frac{\ulcorner v_1 \bowtie v_2 \xrightarrow{\mathsf{pure}} v \urcorner}{\mathsf{wp}\,\langle t,\ v_1 \bowtie v_2\rangle\,\{\lambda\_v'.\ \ulcorner v' = v \urcorner\}}$$

$$\textsc{Bind}\quad \frac{\mathsf{wp}\,\langle t,\ e\rangle\,\{\lambda\, t'\, v.\ \mathsf{wp}\,\langle t',\ K[v]\rangle\,\{\Psi\}\}}{\mathsf{wp}\,\langle t,\ K[e]\rangle\,\{\Psi\}}$$

$$\textsc{LetVal}\quad \frac{\mathsf{wp}\,\langle t,\ [v/x]e\rangle\,\{\Psi\}}{\mathsf{wp}\,\langle t,\ \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle\,\{\Psi\}}$$

$$\textsc{Call}\quad \frac{\ulcorner |\vec{x}| = |\vec{w}| \urcorner \qquad \mathsf{Func}\,v\,f\,\vec{x}\,e \qquad \mathsf{wp}\,\langle t,\ [v/f][\vec{w}/\vec{x}]e\rangle\,\{\Psi\}}{\mathsf{wp}\,\langle t,\ v\,\vec{w}\rangle\,\{\Psi\}}$$

$$\textsc{IfTrue}\quad \frac{\mathsf{wp}\,\langle t,\ e_1\rangle\,\{\Psi\}}{\mathsf{wp}\,\langle t,\ \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2\rangle\,\{\Psi\}}$$

$$\textsc{IfFalse}\quad \frac{\mathsf{wp}\,\langle t,\ e_2\rangle\,\{\Psi\}}{\mathsf{wp}\,\langle t,\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2\rangle\,\{\Psi\}}$$

$$\textsc{CASSucc}\quad \frac{\ulcorner 0 \le i < |\vec{w}| \ \wedge\ \vec{w}(i) = v_0 \ \wedge\ v_0 = v \urcorner \qquad \ell \mapsto \vec{w} \qquad v_0 \circleddash t}{\mathsf{wp}\,\langle t,\ \mathsf{CAS}\,\ell\,i\,v\,v'\rangle\,\{\lambda\_b.\ \ulcorner b = \mathsf{true} \urcorner \ *\ \ell \mapsto [i := v']\vec{w}\}}$$

$$\textsc{CASFail}\quad \frac{\ulcorner 0 \le i < |\vec{w}| \ \wedge\ \vec{w}(i) = v_0 \ \wedge\ v_0 \ne v \urcorner \qquad \ell \mapsto_p \vec{w} \qquad v_0 \circleddash t}{\mathsf{wp}\,\langle t,\ \mathsf{CAS}\,\ell\,i\,v\,v'\rangle\,\{\lambda\_b.\ \ulcorner b = \mathsf{false} \urcorner \ *\ \ell \mapsto_p \vec{w}\}}$$

$$\textsc{Store}\quad \frac{\ulcorner 0 \le i < |\vec{w}| \urcorner \qquad \ell \mapsto \vec{w}}{\mathsf{wp}\,\langle t,\ \ell[i] \leftarrow v\rangle\,\{\lambda\_\_.\ \ell \mapsto [i := v]\vec{w}\}}$$

$$\textsc{Par}\quad \frac{\begin{array}{c}\forall t_1\, t_2.\ \ t \preccurlyeq t_1 \ *\ t \preccurlyeq t_2 \Rrightarrow \exists \Psi_1\, \Psi_2.\ \ \mathsf{wp}\,\langle t_1,\ e_1\rangle\,\{\Psi_1\} \ *\ \mathsf{wp}\,\langle t_2,\ e_2\rangle\,\{\Psi_2\}\ * \\ (\forall t_1'\, v_1\, t_2'\, v_2\, t'\, \ell.\ \ \Psi_1\, t_1'\, v_1 \ *\ \Psi_2\, t_2'\, v_2 \ *\ t_1' \preccurlyeq t' \ *\ t_2' \preccurlyeq t' \ *\ \ell \mapsto [v_1; v_2] \mathrel{-\!\!*} \Psi\, t'\, \ell)\end{array}}{\mathsf{wp}\,\langle t,\ e_1 \,||\, e_2\rangle\,\{\Psi\}}$$

Fig. 7. Syntax-directed rules of DisLog

The Value rule asserts that if the symbolical evaluation of an expression ended at timestamp t, yielding a value v, then the postcondition $\Psi\, t\, v$ should hold. The Load rule extends the standard separation logic rule to prevent entanglement. The loaded value v must have been allocated before the current timestamp t, via the $v \circleddash t$ assertion in the precondition. The CASSucc and CASFail rules are similarly extended: they prevent entanglement by requiring that the scrutinized value was allocated before the current timestamp.

The Closure and TopLevel rules produce an assertion $\mathsf{Func}\, v\, f\, \vec{x}\, e$ certifying that calling v as a function will not cause entanglement. Obtaining this assertion for closures may be surprising at first, but is warranted by the following facts: *(i)* all the timestamps of locations captured by the closure are guaranteed to precede the closure's allocation timestamp t (by rule MementoPre), and *(ii)* closures are immutable objects and, as such, cannot themselves create entanglement [Westrick et al. 2022]. Phrased differently, the locations of the environment are allocated before the closure itself, and thanks to immutability, this fact never changes. The Func predicate is persistent. The Call rule allows calling a function, given the Func predicate. Proving that the environment was allocated before the current timestamp amounts to proving that the closure's location itself was allocated before the current timestamp, which is true since the closure's location is already part of the expression (§4.4).

The Bind rule gives meaning to the notion of the "current timestamp" of an expression. Operationally, the evaluation of a term $K[e]$ at timestamp t reduces the sub-expression e until it reaches a value v and an end timestamp t'. Then, the whole term $K[v]$ starts reducing at the new timestamp t'. The Bind rule paraphrases this operational behavior. The rule asserts that the user first has to reason about the sub-expression e at the same current timestamp. The user has then to reason about the filled term $K[v]$ at a current timestamp t', under the precondition that the sub-expression e reduced to a value v at end timestamp t'.

Reasoning About a Parallel Tuple. A pivotal rule of DisLog is Par. Let's first derive a naive version ParWeak of this rule below before focusing on the ultimate rule given in Fig. 7.

$$\frac{\forall\, t_1\, t_2. \qquad t \preccurlyeq t_1 \twoheadrightarrow \mathsf{wp}\,\langle t_1,\, e_1\rangle\,\{\Psi_1\} \qquad t \preccurlyeq t_2 \twoheadrightarrow \mathsf{wp}\,\langle t_2,\, e_2\rangle\,\{\Psi_2\}}{\mathsf{wp}\,\langle t,\, e_1 \,||\, e_2\rangle \left\{\lambda\, t'\, \ell.\ \begin{array}{l} \exists t_1'\ v_1\ t_2'\ v_2.\ \Psi_1\, t_1'\, v_1 * \Psi_2\, t_2'\, v_2 \\ t_1' \preccurlyeq t' * t_2' \preccurlyeq t' * \ell \mapsto [v_1; v_2] \end{array}\right\}}\ \textsc{ParWeak}$$

This rule allows reasoning about a parallel tuple $e_1 \,||\, e_2$ at current timestamp t. The premise universally quantifies over two fresh timestamps t_1 and t_2, which are used for e_1 and e_2, respectively. We focus on the left-hand side of the tuple as the right-hand side side is handled similarly. The user should verify e_1 with the postcondition Ψ_1, under the hypothesis $t \preccurlyeq t_1$ witnessing that t precedes t_1. This information allows the user to safely acquire any location that was safe to acquire from t. Indeed, if the user has an assertion $\ell \circledcirc t$, they can use the ClockMono rule to obtain an assertion $\ell \circledcirc t_1$.

We emphasize that the above rule ensures that the two fresh timestamps t_1 and t_2 are *unrelated*. Hence, an assertion $\ell \circledcirc t_1$ *cannot* be converted to an assertion $\ell \circledcirc t_2$. This would indeed be unsafe, as a location allocated by the left task could be acquired by the right one, creating entanglement.

After the join point, the postcondition of the ParWeak rule asserts that e_1 reduced to a value v_1 at end timestamp t_1', and that e_2 reduced to a value v_2 at end timestamp t_2' . The postcondition also produces witnesses that t_1' and t_2' precede the (new) current timestamp t'. Thanks to these two assertions any locations allocated by either of the two tasks are now accessible by any task at timestamp t'' such that $t' \preccurlyeq t''$. Finally, the postcondition asserts that the parallel tuple itself reduced to a location ℓ, pointing to the two resulting values v_1 and v_2.

Unfortunately, the ParWeak rule is tedious to use in practice. It fails to support a common pattern underlying our proof rules, which would allow the postconditions Ψ_1 and Ψ_2 of the newly forked tasks to depend on the tasks' timestamps t_1 and t_2. This dependence is rendered impossible by the universal quantification of the timestamps t_1 and t_2. Our final rule Par presented in Fig. 7 facilitates the wished-for pattern. The premise of the Par rule quantifies universally over the two timestamps, and *after* the quantification, allows the user to choose two existentially quantified postconditions Ψ_1 and Ψ_2 that can depend on the two timestamps. Moreover, the user is free to choose the postconditions after a potential ghost update; for example, to allocate an invariant that depends on both t_1 and t_2. The user should then verify the two parts of the parallel tuple with their respective timestamps, as in the ParWeak rule. Finally, the user has to show that the resulting values and timestamps entail the postcondition Ψ. This is formally expressed in the second line of the precondition of the rule.

4.4 Soundness

Finally, we devote our attention to stating and proving soundness of DisLog. For our disentanglement logic to be sound it has to hold that the reduction of a program verified using the rules of DisLog leads to a disentangled program state. Our semantics is phrased in terms of a transition system that gets stuck if entanglement is encountered, ensured by the highlighted premises for head reductions in Fig. 4. Soundness of our logic thus must entail that verified programs cannot get stuck.

Since we use a small-step semantics in a parallel world, the definition of "not getting stuck" needs careful wording. In particular, it is not enough to say that "the configuration can take a step". Indeed, one step of DisLang corresponds to a step of *one* task, whereas we want to ensure that *every*

$$\textsc{RedSched}\ \frac{S\,/\,T\,/\,e \xrightarrow{\text{sched}} S'\,/\,T'\,/\,e'}{\mathsf{red}\ S\ T\ e} \qquad \textsc{RedCtx}\ \frac{\mathsf{red}\ S\ T\ e}{\mathsf{red}\ S\ T\ (K[e])} \qquad \textsc{RedPar}\ \frac{(e_1 \notin \mathcal{V} \lor e_2 \notin \mathcal{V}) \quad (e_1 \notin \mathcal{V} \implies \mathsf{red}\ S\ T_1\ e_1) \quad (e_2 \notin \mathcal{V} \implies \mathsf{red}\ S\ T_2\ e_2)}{\mathsf{red}\ S\ (T_1 \otimes T_2)\ (e_1 \mid\mid e_2)}$$

Fig. 8. Reducibility of a configuration

$$\begin{aligned}
\mathsf{wp}\ \langle t,\ e\rangle\ \{\Psi\} &\triangleq \mathsf{wpg}\ \langle t,\ e\rangle\ \{\Psi\} \\
\mathsf{wpg}\ \langle T,\ e\rangle\ \{\Psi\} &\triangleq \\
&\big(\ulcorner e \in \mathcal{V} \urcorner \ast (\forall S.\ \mathsf{interp}\ S\ T\ e \Rrightarrow \ulcorner T \in \mathcal{T} \urcorner \ast \mathsf{interp}\ S\ T\ e \ast \Psi\ T\ e)\big) \\
&\lor \big(\ulcorner e \notin \mathcal{V} \urcorner \ast (\forall S.\ \mathsf{interp}\ S\ T\ e \Rrightarrow \\
&\qquad \ulcorner \mathsf{red}\ S\ T\ e \urcorner \ast \forall S'\ T'\ e'.\ \ulcorner S\,/\,T\,/\,e \xrightarrow{\text{step}} S'\,/\,T'\,/\,e' \urcorner \Rrightarrow \triangleright \Rrightarrow \mathsf{interp}\ S'\ T'\ e' \ast \mathsf{wpg}\ \langle T',\ e'\rangle\ \{\Psi\})\big)
\end{aligned}$$

Fig. 9. Definition of the weakest precondition modalities

task can take a proper step. We formalize this intuition with the notion of *reducibility*, captured by the judgment red $S\ T\ e$ and presented in Fig. 8. The RedSched rule asserts that a configuration that can make a scheduling step (either a head step, a fork, or a join) is reducible. The RedCtx rule asserts that the reducibility of a configuration facing an expression under an evaluation context amounts to the reducibility of this very expression. The RedPar rule asserts that a configuration facing a node of the task tree and a parallel tuple is reducible if at least one side of the pair is not a value (otherwise, a join should be possible), and each side that is not a value is reducible.

An expression e is *safe* if $(\emptyset, \emptyset, \emptyset)\,/\,t\,/\,e \xrightarrow{\text{step}}{}^{*} S'\,/\,T'\,/\,e'$ implies that either the configuration $S'\,/\,T'\,/\,e'$ is reducible, or that e' is a value and T' a single leaf. Our soundness theorem asserts that if an expression e can be verified using DisLog, then it is safe.

Theorem 4.1 (Soundness of DisLog). *If* $\mathsf{wp}\ \langle t,\ e\rangle\ \{\lambda\_\_.\ \ulcorner True \urcorner\}$ *holds, then* e *is safe.*

As the semantics of DisLang cannot progress when entanglement is detected, the soundness theorem asserts that e cannot reach an entangled state. The formal proof Theorem 4.1 can be found in our Coq formalization [Anon. 2023]. We detail below the main definitions and invariants.

Definition of the Weakest Precondition. The formal definition of the wp $\langle t,\ e\rangle\ \{\Psi\}$ assertion can be found in Fig. 9. The crux of our approach is to define the wp modality with respect to a more general—hidden from the user—weakest precondition modality that we refer to as wpg modality. The wpg modality is parameterized not with a single timestamp, but a whole task tree: we found this generalization crucial for the various proofs to succeed. Nevertheless, reasoning always takes place at the leaves. Hence, we can hide the details of the task tree from the user.

The definition of the wpg modality appears also in Fig. 9 and follows the traditional Iris recipe [Jung et al. 2018, §6]. As usual, the WP is defined as a guarded fixpoint, and makes use of a *state interpretation predicate* (or *central invariant*), written interp, which relates the ghost state and the physical state. The definition of wpg cases on whether the expression is a value or not. If the expression is a value, we can access the state interpretation, and deduce that the task tree consists of a single leaf and the postcondition. Otherwise, if the expression is not a value, the wpg modality asserts that the configuration is reducible, and that for any possible step, the state interpretation must continue to hold, as well as the WP of the reduced program. Apart from its mention of timestamps, our WP distinguishes itself from the standard Iris WP by making the state interpretation available in the value case, and making use of our custom red judgment.

$$\textsc{RDeLeaf}\quad \frac{\forall \ell.\ \ell \in locs(e) \implies \alpha(\ell) \preccurlyeq_G t}{\mathsf{rootsde}\ \alpha\ G\ t\ e} \qquad\qquad \textsc{RDePar}\quad \frac{\mathsf{rootsde}\ \alpha\ G\ T_1\ e_1 \qquad \mathsf{rootsde}\ \alpha\ G\ T_2\ e_2}{\mathsf{rootsde}\ \alpha\ G\ (T_1 \otimes T_2)\ (e_1 \mid\mid e_2)}$$

$$\textsc{RDeCtx}\quad \frac{\mathsf{rootsde}\ \alpha\ G\ T\ e \qquad \forall \ell\, t.\ \ell \in locs(K)\ \wedge\ t \in \mathsf{leaves}(T) \implies \alpha(\ell) \preccurlyeq_G t}{\mathsf{rootsde}\ \alpha\ G\ T\ (K[e])}$$

$$\mathsf{interp}\ (\sigma, \alpha, G)\ T\ e \ \triangleq\ \begin{array}{l} \ulcorner \mathsf{dom}(\sigma) = \mathsf{dom}(\alpha)\ \wedge\ \mathsf{rootsde}\ \alpha\ G\ T\ e \urcorner\ * \\ \boxed{\bullet G}^{\gamma}\ *\ \mathsf{Heap}\ \sigma\ *\ \mathop{\Huge *}_{(\ell,t)\in\alpha} (\mathsf{meta}\ \ell\ t) \end{array}$$

$$\mathsf{edge}\ t\ t' \ \triangleq\ \boxed{\circ\{(t,t')\}}^{\gamma} \qquad t \preccurlyeq t' \ \triangleq\ \mathsf{rtc\ edge}\ t\ t' \qquad \ell \circleddash t \ \triangleq\ \exists t_0.\ \mathsf{meta}\ \ell\ t_0\ *\ t_0 \preccurlyeq t$$

$$\mathsf{Func}\ v\ f\ \vec{x}\ e \ \triangleq\ \ulcorner v = \hat{\mu} f.\lambda\vec{x}.e \urcorner \ \vee\ \left(\ell \mapsto_{\square} (\mu f.\lambda\vec{x}.e)\ *\ \exists t.\ \mathsf{meta}\ \ell\ t\ *\ \mathop{\Huge *}_{\ell' \in locs(e)} (\ell' \circleddash t)\right)$$

Fig. 10. Definition of the state interpretation predicate and of base assertions

The State Interpretation Predicate. Our WP maintains a state interpretation predicate between each reduction step, which is defined in Fig. 10. We review its definition next.

We first focus on the roots disentanglement judgment rootsde $\alpha\ G\ T\ e$. This judgment asserts each task of the expression e only uses locations allocated before its associated timestamp. This judgment allows stating the MementoPre rule. If the task tree consists of a single leaf t, the RDeLeaf rule requires that the locations of e were allocated before t. The RDePar rule requires that both sides of a parallel tuple satisfy the rootsde judgment. In the case of an evaluation context, RDeCtx requires that the judgment holds for the expression under the context, and that the locations occurring in the evaluation context itself are allocated before all the leaves of the task tree.

The state interpretation predicate also gives meaning to the ghost state from the physical state. We first briefly explain the construction of ghost state in Iris abstractly, before detailing the part of interp that concerns ghost state. In Iris, ghost state is defined in terms of so-called *cameras* (CMRA) which can be thought as "step-indexed partial commutative monoids" [Jung et al. 2018], detailing a resource algebra. Iris provides predefined notions of resource algebras. For example, the resource algebra $Auth(M)$ describes the *authoritative resource algebra* over the resources M. This resource algebra gives access to $\bullet a$, the *authoritative* ownership of a, and $\circ b$, the *fragmentary* ownership of b. Together, these two assertions entail that there exists an element c of the algebra such that $a = b \cdot c$. The resource algebra $Set(M)$ describes the *set resource algebra*, where the composition of resources is described by set union. For our state interpretation predicate, we define a ghost cell γ which we equip with the resource algebra $Auth(Set(\mathcal{T} \times \mathcal{T}))$. The ghost cell γ stores the computation graph and gives meaning to the precedence assertion.

Iris moreover provides a generic construction to define points-to assertions via the gen_heap library [Iris Development Team 2023]. This library defines a certain piece of ghost state, defines an assertion Heap σ that ties a store σ to this ghost state, and defines the points-to assertion $\ell \mapsto_p \vec{w}$ in terms of this ghost state. Moreover, the gen_heap library allows associating persistent information to locations via a mechanism of *meta* assertions. In our case, we associate to each location ℓ the timestamp t of the task that allocated it, and write meta $\ell\ t$. The main property of this assertion is that, from the knowledge meta $\ell\ t$ and meta $\ell\ t'$, we can deduce that $t = t'$.

We are now able to review the details of the definition of our state interpretation predicate shown in the lower part of Fig. 10. First, it asserts that the domain of the store is the same as the allocation map, and the roots disentanglement judgment. The state interpretation also asserts the ghost authoritative ownership of the computation graph $\boxed{\bullet G}^{\gamma}$ and the ownership of the store via the Heap σ assertion. Moreover, the state interpretation asserts, for every mapping from a location ℓ to a timestamp t in the allocation map α, that the persistent knowledge meta $\ell\ t$ was set.

We define an edge between t and t' as a ghost fragmentary ownership of the singleton $\boxed{\circ\{(t,t')\}}^{\gamma}$. The conjunction of $\boxed{\bullet G}^{\gamma}$ and $\boxed{\circ\{(t,t')\}}^{\gamma}$ allows to deduce that $(t,t') \in G$. We define the precedence assertion $t \preccurlyeq t'$ as the reflexive-transitive-closure (rtc) over the edge predicate. The clock assertion $\ell \circleddash t$ is defined as a paraphrase of its informal definition. The location ℓ was allocated before timestamp t if there exists a timestamp t_0 such that ℓ was allocated at t_0, and t_0 precedes t.

The representation predicate of a λ-abstraction Func $v\ f\ \vec{x}\ e$ is a disjunction: either v is a top-level function, or a heap-allocated closure. In that case, we use a discarded fraction for the points-to assertion, as the closure is immutable. The predicate also asserts the existence of a timestamp t at which the closure was allocated, and that every location of its environment (the locations occurring in e) was allocated before t. We make use of this knowledge to verify the Call rule of Fig. 7. The closure's location is allocated before the current timestamp (thanks to the rootsde judgment), but since the locations of the environment were allocated before the allocation time of the closure itself, they are also allocated before the current timestamp, and hence safe to acquire.

5 A HIGH-LEVEL LOGIC: DISLOG+

In this section, we introduce DisLog+, an almost standard concurrent separation logic allowing to prove disentanglement for a large class of programs.

5.1 Don't Poke the Bear

Disentanglement is preserved by restricting reads: when a task acquires a location, the programmer must ensure that this location was allocated by a preceding task. However, numerous programs "don't poke the bear", that is, are disentangled because they do not read hazardous shared locations.

Determinacy-race-free programs are an example of such cautious programs. A *determinacy race* [Feng and Leiserson 1999], which we will call a *race* from now on, occurs when two concurrent tasks access the same location, and at least one of these accesses is a write. As noticed by Westrick et al. [2020], race-free programs are trivially disentangled: shared locations cannot be written to from different tasks, which prevent the communication of freshly allocated data between tasks. Westrick et al. [2020] moreover noticed that there exist some races that are also trivially disentangled. Races that fall into this category are: *(i)* write-write races, because a write does not acquire a value, and *(ii)* read-write races on data that was allocated before the beginning of the parallel phase, because tasks are allowed to communicate previously-allocated data.

What is the common denominator of all these cautious programs? Rather than categorically restricting reads, they demand a more nuanced consideration of writes. More precisely, these programs ensure that when a task writes a value to a location, this value is safe to acquire for any task that can access the said location. Race-free programs prevent hazardous concurrent reads, because a task must have unique ownership of a location to write to it. Write-write races do not restrict writes, as there is no concurrent read, and read-writes races are permitted as long as the written value was allocated before the beginning of the parallel phase.

For this large class of programs that don't poke the bear, we provide an almost traditional separation logic called DisLog+. By "traditional", we mean that the weakest precondition of DisLog+ takes a form which does not mention timestamps (§5.2). Moreover, its syntax-directed reasoning rules do not mention clock nor precedence assertions: they are the standard reasoning rules of concurrent separation logic (§5.3). By "almost", we stress that DisLog+ restricts the use of ghost state to prevent races, but is otherwise a standard separation logic. DisLog+ is hence ideally suited to reason about race-free programs. To cater to the benign races identified above, we extend DisLog+ with *(i)*

$$\begin{array}{rcl@{\qquad}rcl} vProp & \triangleq & \mathcal{T} \xrightarrow{mon} iProp & P \vdash_{vProp} P' & \triangleq & \forall t.\, P\,t \vdash_{iProp} P'\,t \\ P * P' & \triangleq & \lambda t.\, P\,t * P'\,t & \lceil \Phi \rceil & \triangleq & \lambda\_.\, \Phi \\ P \mathrel{-\!\!*} P' & \triangleq & \lambda t.\, \forall t'.\, t \preccurlyeq t' \mathrel{-\!\!*} P\,t' \mathrel{-\!\!*} P'\,t' & P@t & \triangleq & P\,t \\ \forall x.\, P & \triangleq & \lambda t.\, \forall x.\, P\,t & \ell \circledcirc now & \triangleq & \lambda t.\, \ell \circledcirc t \\ \exists x.\, P & \triangleq & \lambda t.\, \exists x.\, P\,t & \ell \mapsto_p \vec{w} & \triangleq & \lceil \ell \mapsto_p \vec{w} \rceil * \vec{w} \circledcirc now \end{array}$$

$$\mathsf{wpm}\, e\, \{Q\} \triangleq \lambda t.\, \forall t.\, t \preccurlyeq t' \mathrel{-\!\!*} \mathsf{wp}\, \langle t', e\rangle\, \{\lambda t''\, v.\, (Q\, v)@t''\}$$

Fig. 11. DisLog+ separation logic and assertions

write-only assertions to reason about write-write races (§5.4) and *(ii)* three rules to reason about read-write races on previously allocated data that we refer to as the philosopher's lemmas (§5.5).

5.2 Monotonicity to the Rescue

Our development of DisLog+ was triggered by two observations about race-free programs: *(i)* race-free programs ensure that, when a task accesses a location, any value referenced by the location is safe for the task to acquire, and *(ii)* this property is *monotonic* with respect to the precedence pre-order. Indeed, if all the pointed to values are safe to acquire for a given task, then these values are also safe to acquire for any of the task's descendants in the computation graph.

Taking inspiration from program logics that target weak-memory models [Kaiser et al. 2017; Mével et al. 2020], we define a whole new separation logic, in which every assertion is parameterized by a timestamp, called the *ambient* timestamp, and is monotonic with respect to the precedence pre-order. Fig. 11 presents the formal definitions of these assertions, written P and of type $vProp$.

The user can always *project* a $vProp$ assertion P to a particular timestamp t in $iProp$ via the construction $P@t$. Conversely, the lifting construction $\lceil \Phi \rceil$ allows to lift an $iProp$ assertion Φ into $vProp$. Hence, the whole ghost-state theory of $iProp$ is available in $vProp$. When the context allows it, we write Φ instead of $\lceil \Phi \rceil$ for the lifting of an $iProp$ assertion into $vProp$. The entailment of $vProp$ (written $\vdash_{vProp}$) is defined using the entailment of $iProp$ (written $\vdash_{iProp}$). The definition ensures that an entailment $P \vdash_{vProp} P'$ is valid if and only if, for any timestamp t the projection $P@t$ of the premise entails the projection $P'@t$ of the conclusion.

Fig. 11 also defines the assertions relative to DisLog+. The $\ell \circledcirc now$ assertion asserts that ℓ was allocated before the ambient timestamp. This is a persistent assertion, whose monotonicity is ensured by the ClockMono rule. Again, we overload this assertion to arbitrary values and collection of values. The crux of our approach resides in the definition of the points-to assertion. The assertion $\ell \mapsto_p \vec{w}$ is defined as the conjunction of the points-to assertion in $iProp$, written $\lceil \ell \mapsto_p v \rceil$, as well as the knowledge that every value pointed to by the location was allocated before the ambient timestamp, using the $\vec{w} \circledcirc now$ assertion. Hence, the points-to assertion of $vProp$ asserts that every load for this location is safe for this particular task, and any subsequent task!

The WP of DisLog+ takes the form $\mathsf{wpm}\, e\, \{\lambda v.\, P\}$, where the variable v denotes the resulting value of e and is bound in P. To abstract over the details of the postcondition, we write Q instead of $\lambda v.\, P$. The assertion $\mathsf{wpm}\, e\, \{Q\}$ asserts that e is safe to execute at any timestamp succeeding the ambient one, and that if e reaches a value v, then $Q\, v$ holds at the end timestamp (or at any subsequent timestamp, since Q is monotonic).

The user can freely go between DisLog+ and DisLog using the following Conversion rule.

$$\big(P \vdash_{vProp} \mathsf{wpm}\, e\, \{Q\}\big) \iff \big(\forall t.\;\; P@t \vdash_{iProp} \mathsf{wp}\, \langle t, e\rangle\, \{\lambda\, t'\, v.\, (Q\, v)@t'\}\big) \qquad \text{(Conversion)}$$

This rule needs a careful reading. It asserts that the precondition P entails $\mathsf{wpm}\, e\, \{Q\}$ in $vProp$ if and only if (in the meta-logic), for any timestamp, the projection of the precondition at this timestamp entails the WP of DisLog with the postcondition projected at the end timestamp. Notice

$$\textsc{Alloc+}\quad \text{wpm}\,(\text{alloc}\ n\ v)\ \{\lambda\,\ell.\,\ell \mapsto v^n\}$$

$$\textsc{Store+}\quad \frac{\ulcorner 0 \leq i < |\vec{w}| \urcorner \qquad \ell \mapsto \vec{w}}{\text{wpm}\,(\ell[i] \leftarrow v)\ \{\lambda\_.\,\ell \mapsto [i := v]\vec{w}\}}$$

$$\textsc{Load+}\quad \frac{\ulcorner 0 \leq i < |\vec{w}| \ \wedge\ \vec{w}(i) = v \urcorner \qquad \ell \mapsto_p \vec{w}}{\text{wpm}\,(\ell[i])\ \{\lambda\,v'.\,\ulcorner v' = v \urcorner \ * \ \ell \mapsto_p \vec{w}\}}$$

$$\textsc{Par+}\quad \frac{\text{wpm}\,e_1\,\{Q_1\} \qquad \text{wpm}\,e_2\,\{Q_2\}}{\text{wpm}\,(e_1 \,||\, e_2)\ \{\lambda\,\ell.\,\exists v_1\,v_2.\ \ell \mapsto [v_1; v_2] \ * \ Q_1\,v_1 \ * \ Q_2\,v_2\}}$$

Fig. 12. Selected rules of DisLog+

that the Conversion rule is an equivalence. While it is not surprising that a specification in DisLog+ is valid in DisLog (the former being more restrictive than the latter), the converse is also true: the user can use the full power of DisLog rules to verify a DisLog+ interface.

The soundness of DisLog+ is a direct corollary of the soundness of DisLog (Theorem 4.1), thanks to the Conversion rule.

Theorem 5.1 (Soundness of DisLog+). *If* $\text{wpm}\,e\,\{\lambda\_.\,\ulcorner True \urcorner\}$ *holds, then* e *is safe.*

5.3 Reasoning Rules of DisLog+

We showcase the most important reasoning rules of DisLog+ in Fig. 12, which are similar to the reasoning rules of the original concurrent separation logic with fractional permissions [Bornat et al. 2005]. These rules are expressed at the *vProp* level, where the horizontal bar stands for *vProp* entailment. In particular, the points-to assertion occurring in the rules is the one defined in Fig. 11, guaranteeing that any load will be safe. Nevertheless, all the rules of Fig. 12 are derived from the rules of DisLog (§4.3) using the Conversion rule.

The rules we present in Fig. 12 prevent races. Indeed, the only way to allow a race is by sharing a full points-to assertion between tasks, which is only possible via invariants [Jung et al. 2018, §2.2]. Because invariants can only be installed for assertions of type *iProp*, but points-to assertions in DisLog+ are of type *vProp*, DisLog+ rules out races by construction. We alluded to this restricted use of Iris ghost state by referring to DisLog+ as an "almost" standard separation logic (§5.1).

The Alloc+ rule produces a valid points-to assertion, that is, both the ownership information and the proof that the default value is safe to acquire at the ambient timestamp. Indeed, as the default value v occurs in the expression alloc $i\ v$, it was already acquired, hence already safe. This is reminiscent of the MementoPre rule. The Store+ rule is also standard and preserves the fact that any subsequent load will be safe. Indeed, the stored value v occurs in the expression $\ell[i] \leftarrow v$, and is hence safe to acquire. The Load+ rule is the standard rule of separation logic, as it internally rests on the fact that all the values pointed to by the location are safe to acquire. Finally, the Par+ rule heavily makes use of the monotonicity of *vProp* assertions. Indeed, the two postconditions Q_1 and Q_2 are valid for the end timestamp of the two forked tasks. Hence, they are also valid for the end timestamp of the parallel tuple, that succeeds them.

5.4 Write-Write Races are Disentangled: Fractional Write-Only Assertions

A write-write race occurs when two or more tasks race to write to a shared location, but neither of them (or any other task) ever reads from the said location. Write-write races are always disentangled, as a write does not acquire any location. However, from the point of view of functional correctness, write-write races are subtle: once the tasks join and the program reads the shared location, all the outcomes of the race should be taken into account.

To verify such races in more standard Iris settings, the user typically installs an invariant containing the points-to assertion, quantifies existentially over the pointed to value, and constrains it using ghost state. This existential quantification allows the user to change the pointed to value

WOStart
$$\ell \mapsto [v] \;\Rrightarrow\; \exists \delta.\ \mathsf{orig}^{\delta}\, v * \ell \Mapsto^{\delta}_{1} \emptyset$$

WOFrac
$$\ell \Mapsto^{\delta}_{(p_1+p_2)} (X_1 \cup X_2) \;\dashv\vdash\; \ell \Mapsto^{\delta}_{p_1} X_1 * \ell \Mapsto^{\delta}_{p_2} X_2$$

WOStore
$$\frac{\ell \Mapsto^{\delta}_{p} X}{\mathsf{wpm}\ (\ell[0] \leftarrow v)\ \{\lambda\_.\ \ell \Mapsto^{\delta}_{p} \{v\}\}}$$

WOCancel
$$\mathsf{orig}^{\delta}\, v * \ell \Mapsto^{\delta}_{1} \emptyset \;\Rrightarrow\; \ell \mapsto [v]$$

WOEnd
$$\frac{X \neq \emptyset}{\ell \Mapsto^{\delta}_{1} X \;\Rrightarrow\; \exists v.\ \ulcorner v \in X \urcorner * \ell \mapsto [v]}$$

Fig. 13. Fractional write-only points-to assertions

while preserving the invariant. In DisLog+, invariants are restricted to *iProp* assertions, and thus the user cannot store a points-to assertion inside an invariant. To allow the verification write-write races in DisLog+, we introduce the notion of a fractional *write-only assertion* presented in Fig. 13.

A write-only assertion takes the form $\ell \Mapsto^{\delta}_{p} X$, where δ is a list of ghost names, p a positive fraction less or equal to 1, and X a set of possible values. When $p = 1$, the set X contains all the values possibly written to ℓ. The write-only assertion comes with a companion assertion $\mathsf{orig}^{\delta}\, v$, which is persistent and describes the *original* value of the points-to assertion. The WOStart rule consumes a points-to assertion and produces an orig assertion and an empty write-only assertion. The WOFrac rule asserts that the write-only assertion is fractional: the user can always arbitrarily split and join it. The WOStore rule allows executing a store operation, overwriting the set of possible values. This rule only requires a fraction of the write-only assertion: a concurrent task could have another fraction and race to write.

Fig. 13 also includes two rules for getting back a points-to assertion from a write-only assertion. In both cases, the full fraction 1 must be given back. First, the WOCancel rule can be used if no write occurred, as witnessed by the empty write-only assertion. Then, the rule produces the original points-to assertion. Second, the WOEnd rule requires that at least one write occurred. If so, the rule produces a points-to assertion and a proof that the pointed to value is in the set of possible values.

The definition of write-only assertions appears in the supplementary material [Anon. 2023]. This definition makes use of standard ghost state and in particular a *cancellable invariant* [Jung et al. 2018, §7.1] in which a points-to assertion of DisLog is stored. Notably, the assertion $\ell \Mapsto^{\delta}_{p} X$ carries not only information on the contents of the cancellable invariant, but also a witness X ⌚ *now* that the set of possible values was allocated before the ambient timestamp. Hence, after an application of the WOEnd rule, we can reconstruct back a *vProp* points-to by canceling the invariant and exhibiting a witness that the pointed to value was allocated before the ambient timestamp.

While write-only assertions fit well in the context of *vProp*, we stress that they are not specific to it. Similar definitions can be proposed for regular points-to assertions at the *iProp* level, dropping assertions related to timestamps. In our discussion of write-only assertions, we focus on *references*, which in DisLang are represented by arrays of size 1. To target arbitrary arrays, we assume that our approach can be generalized to a more detailed interface with a per-offset write-only points-to assertion. This generalization should be purely mechanical.

5.5 Many Read-Write Races are Disentangled: Philosopher's Lemmas

Read-write races can create entanglement: a task could communicate an object it allocated to a concurrent task. Formally verifying that such races are safe often requires the user to use the full expressiveness of DisLog. However, some read-write races are trivially disentangled: if the communicated value is not allocated in the heap, or if the write swaps around previously allocated values. For such cases, Fig. 14 offers an interface of "philosopher's lemmas" in DisLog+.

This interface takes advantage of the fact that a *vProp* assertion can be viewed as the conjunction of a *subjective* part that depends on the ambient timestamp, and an *objective* part, that does not [Mével et al. 2020, §4.1]. As it does not depend on the ambient timestamp, the objective part

SplitSubjObj
$P \dashv\vdash \exists t.\, \uparrow t \ast P@t$

$\mathrm{group}\, L \triangleq \bigast_{(\ell,p,\vec{w})\in L} (\ell \mapsto_p \vec{w})$

GetClock
$$\frac{\mathrm{group}\, L}{\mathrm{group}\, L \ast (\mathrm{values}\, L) \circleddash\, now}$$

Objectivize
$$\frac{\mathrm{group}\, L \qquad (\mathrm{values}\, L) \circleddash\, t}{(\mathrm{group}\, L)@t}$$

Fig. 14. Philosopher's lemmas

can be projected into *iProp* and shared using an invariant. The SplitSubjObj rule asserts that an assertion P is equivalent to the conjunction of its subjective part $\uparrow t$ and its objective part $P@t$ where t is an existentially quantified timestamp. The assertion $\uparrow t$ is defined as $\lambda t'.\ t \preccurlyeq t'$ and formalizes that t precedes the ambient timestamp. This is a persistent assertion.

However, the existential quantification of the SplitSubjObj rule may be problematic. Indeed, the user may want to store an assertion of the form $P@t$ inside an invariant. However, opening the invariant, applying the rule from right-to-left and then from left-to-right will produce an assertion $P@t'$ for a new t', which prevent the user to close the invariant. Nevertheless, recall that our goal is to allow reasoning about read-write races on unboxed or previously allocated data, that is, a group of points-to assertions. A group L of points-to assertion is described by the group L assertion, where L is a list of tuples (location, fraction, list of values). The *values* of a list L are the set of the values stored in the points-to assertions. In this particular case, we provide an escape hatch by means of the Objectivize rule. We next illustrate how the reasoning takes place.

Just before reasoning about the parallel phase of the program, the user applies the GetClock rule and obtains an assertion $(\mathrm{values}\, L) \circleddash\, now$. Then, they use the SplitSubjObj rule and obtain a witness $\uparrow t$, a projected group $(\mathrm{group}\, L)@t$ and a projected clock $((\mathrm{values}\, L) \circleddash\, now)@t$. Unfolding the definitions, the latter assertion is equal to $(\mathrm{values}\, L) \circleddash\, t$.

The user can next allocate an invariant containing both the projected group and the clock assertion. The user then applies the Par+ rule, and gives the witness assertion $\uparrow t$ to both sub-tasks—recall that this assertion is persistent. While reasoning about the sub-tasks, for example just before reasoning about a store, the user opens the invariant, and gets the projected group and the clock assertion. The user then applies the SplitSubjObj rule from right-to-left to get a proper group, which they update with the Store+ rule to a group L'.

Finally, the time comes to close the invariant. The user cannot close the invariant by applying again the SplitSubjObj rule, as it will generate a new existentially quantified timestamp that they would not be able to communicate to other tasks. The Objectivize rule comes to save the day. To close the invariant, the goal is to provide an assertion $(\mathrm{group}\, L')@t$. The Objectivize rule asserts that this goal can be deduced from the assertion $(\mathrm{values}\, L') \circleddash\, t$. Thankfully, the user has at hand an assertion $(\mathrm{values}\, L) \circleddash\, t$. Hence, the goal is to prove that $locs(\mathrm{values}\, L) = locs(\mathrm{values}\, L')$. This equality holds if the program only swapped around values, or stored unboxed values.

6 EVALUATION

We showcase DisLog+ and DisLog and via a range of representative case studies.

We first focus on the `scratch` example of Sec. 2, and prove it correct in DisLog+ (§ 6.1). Next, parallel lookup in a lazy collection illustrates how to reason on a write-write race using write-only assertions (§ 6.2). We also verify an implementation of the parallel for loop construction.

Concurrent hashing and deduplication form our central case study. Deduplication consists of removing duplicates from a collection. This task can be efficiently done in parallel using concurrent hashing: each task tries to insert elements into a shared concurrent hash set, which by construction does not store duplicates. If the collection is fully allocated beforehand, we use a folklore hash set, and verify both the hash set interface and the duplication itself entirely within DisLog+ (§ 6.3), thanks to the philosopher's lemmas. In the case of a lazy collection, where elements may not be

$$\frac{\ulcorner \Phi \urcorner}{\mathsf{wpm}\,(\mathsf{new\_lock}\,[])\,\{\lambda\,\ell.\,\exists l\,u.\;\ell \mapsto [l;u] * \ulcorner \mathsf{lock}\,l\,u\,\Phi \urcorner\}}$$

$$\frac{\ulcorner \mathsf{lock}\,l\,u\,\Phi \urcorner}{\mathsf{wpm}\,(l\,[])\,\{\lambda\,b.\,\ulcorner b = \mathsf{true}\urcorner \twoheadrightarrow (\ulcorner \mathsf{locked}\,l \urcorner * \ulcorner \Phi \urcorner)\}} \qquad \frac{\ulcorner \mathsf{lock}\,l\,u\,\Phi \urcorner \quad \ulcorner \mathsf{locked}\,l \urcorner \quad \ulcorner \Phi \urcorner}{\mathsf{wpm}\,(u\,[])\,\{\lambda\,\_.\,\ulcorner \mathit{True} \urcorner\}}$$

Fig. 15. Case study: specification of a spin-lock

already allocated prior to the parallel phase, the previous approach cannot be directly used: naïvely applying the previous deduplication algorithm would result in entanglement, and so a different deduplication algorithm is needed. We address this issue by first partially removing duplicates in parallel with a more subtle hash set, then getting rid of the remaining duplicates by calling the previous deduplication function. Interestingly, the proof requires the full power of DisLog (§6.4).

In the case studies, we write a non-recursive function as $\lambda\vec{x}.e$, which is a sugar for $\mu\_.\lambda\vec{x}.e$. where _ denotes an anonymous binding. We add a hat and write $\hat{\lambda}$ to distinguish top-level functions.

6.1 The scratch Example

We first give an interface to a spin-lock. The verified code is a direct translation of the spin-lock presented earlier (§2.4), which is implemented as a pair of closures sharing a reference to a boolean named r, initialized to false. The first closure attempts a lock by doing a CAS on r from false to true. The second closure releases the lock by setting the reference r to false.

Our specification of locks in DisLog+ appears in Fig. 15 and is very similar to the standard specification of locks in high-order separation logic [Gotsman et al. 2007; Svendsen and Birkedal 2014]. In DisLog+, a lock must be restricted to protect an *iProp* assertion Φ, as a lock protects an invariant. The precondition of the specification of new_lock consumes Φ. The postcondition produces a location ℓ pointing to a pair of closures (l, u), as well as an *iProp* assertion lock $l\,u\,\Phi$, which is persistent and asserts that the l and u describe a valid lock. The specification of a call to the closure l requires a valid lock, and returns a boolean. If this boolean is true, then the lock was successfully locked: the user gains the protected assertion Φ as well an exclusive token locked l, witnessing that the lock is now locked. This token is required to call the closure u, as well as Φ, which has to be given back when the user wants to unlock the lock.

We conduct the proofs of the interface of Fig. 15 entirely within DisLog+. Indeed, we are in an extreme case: the shared reference points-to a boolean, which is unboxed. Hence, the points-to of DisLog+ is equivalent to the points-to of DisLog. Thus, we are able to define the lock $l\,u\,\Phi$ assertion using an invariant that stores directly the points-to assertion of the shared reference.

We make use of the interface of locks we just presented, as well as the philosopher's lemmas, and verify the following interface for the scratch example.

$$\frac{\mathop{\ast}_{i\in\{0,1\}} \quad \forall \ell.\; \ell \mapsto \mathit{defaultElem}^{\mathsf{N}} \twoheadrightarrow \mathsf{wpm}\,(\mathsf{doWork}\,[\ell])\,\{\lambda\,\_.\,Q_i \;*\; \exists\vec{w}.\;\ell \mapsto \vec{w}\}}{\mathsf{wpm}\,(\mathsf{scratch}\,[])\,\{\lambda\,\_.\,Q_0 \;*\; Q_1\}}$$

In our proof, the lock guards the proposition $(\mathit{shared} \mapsto \mathit{defaultElem}^{\mathsf{N}})@t_0$, where t_0 is the timestamp at which *shared* was allocated. Each task gets an assertion $\uparrow t_0$. Then, if a task acquires the lock, we are able to reconstruct a points-to assertion with the SplitSubjObj rule, call the doWork function, call the cleanScratchPad function, and close the invariant using the Objectivize rule.

6.2 Parallel Lookup in a Lazy Collection

The left part of Fig. 16 presents the code of a *parallel loop* parfor $[a; b; h]$, calling the function h for each index between a and b. The presented code is a direct translation of the implementation used

$$\begin{aligned}&\mathsf{parfor} \triangleq \hat{\mu}\, f.\lambda[a;b;h].\\&\quad \mathsf{if}\ (b-a) == 0\ \mathsf{then}\ ()\\&\quad \mathsf{else\ if}\ (b-a) == 1\ \mathsf{then}\ h\ [a]\\&\quad \mathsf{else\ let}\ mid\ =\ a + ((b-a)/2)\ \mathsf{in}\\&\qquad (f\ [a;mid;h])\ ||\ (f\ [mid;b;h])\end{aligned}
\qquad
\begin{aligned}&\mathsf{lookup} \triangleq \hat{\lambda}[k;n].\\&\quad \mathsf{let}\ r\ =\ \mathsf{alloc}\ 1\ ()\ \mathsf{in}\\&\quad \mathsf{let}\ h\ =\ \lambda[i].\ \mathsf{let}\ x\ =\ k\ [i]\ \mathsf{in}\\&\qquad \mathsf{if}\ x == ()\ \mathsf{then}\ ()\ \mathsf{else}\ r[0] \leftarrow x\ \mathsf{in}\\&\quad \mathsf{parfor}\ [0;n;h]\ ;;\ r[0]\end{aligned}$$

$$\frac{\mathop{\ast}_{i\in[a;b)}\ \mathsf{wpm}\ (k\ [i])\ \{\lambda_.\, Q\ i\}}{\mathsf{wpm}\ (\mathsf{parfor}\ [a;b;k])\ \{\lambda_.\ \mathop{\ast}_{i\in[a;b)}\ (Q\ i)\}}
\qquad
\frac{\mathop{\ast}_{i\in[0;n)}\ \mathsf{wpm}\ (k\ [i])\ \{Q\}}{\mathsf{wpm}\ (\mathsf{lookup}\ [k;n])\ \{\lambda\, v.\ \exists \vec{w}.\ \ulcorner \mathsf{found}\ n\ \vec{w}\ v \urcorner\ \ast\ \mathop{\ast}_{v\in\vec{w}}\ (Q\, v)\}}$$

Fig. 16. Case study: parallel lookup in a lazy collection

in the standard library of MPL [MPL Development Team 2022]. The specification of parfor appears below and should be unsurprising. The precondition requires, for every index i between a and b, that $h\ [i]$ is safe and satisfies a postcondition $Q\ i$. The postcondition of parfor $[a;b;h]$ produces the iterated conjunction of the postconditions.

The right part of Fig. 16 presents the code of the lookup $[k;n]$ function that looks up for a non-unit value in the lazy collection k up to index n. To do so, the function uses a reference r, and a closure h that takes an index i, produces the i-th index of the lazy collection, and writes it in r if it is non-unit. The closure h is then called in parallel for every index between 0 and n. This is a typical example of a write-write race: each call on h may write in r, but never read from it.

The specification of lookup $[k;n]$ appears at the bottom of Fig. 16. Its precondition requires that k is valid lazy collection: between indices 0 and n, k produces a value satisfying a predicate Q. The postcondition of the specification produces a value v and asserts the existence of $\vec{w}$, the collection itself. The found $n\ \vec{w}\ v$ judgment asserts $\vec{w}$ has size n and that either v is not the unit value and occurs in $\vec{w}$, or v is the unit value and every value in $\vec{w}$ is the unit value. It is defined as:

$$\mathsf{found}\ n\ \vec{w}\ v \quad \triangleq \quad |\vec{w}| = n \quad \wedge \quad (v \neq () \wedge v \in \vec{w})\ \vee\ (v = () \wedge \forall w.\ w \in \vec{w} \implies w = ())$$

The proof of our specification makes use of a write-only assertion (§ 5.4). Indeed, just after the allocation of r, we convert its points-to assertion into a write-only assertion, and give a fraction $\frac{1}{n}$ to each task. At the end, we do a case analysis on whether a non-unit value is in the lazy collection k, and convert the write-only assertion back to a normal points-to assertion accordingly.

6.3 Deduplication via Concurrent Hashing

In the next two sections, we suppose a user-chosen capacity C, which bounds the number of elements within hash sets. We also suppose a hash function from values to integers.

We present a folklore [VerifyThis 2022] concurrent, lock-free, fixed-capacity hash set using *open addressing* and *linear probing* to handle collision [Knuth 1998]. The code of our hash set appears in the left part of Fig. 17. The hash set consists of an array of size C. When created, the array is filled with a dummy element d, which cannot be inserted in the hash set as it denotes an empty slot. Inserting an element is done by the add $[s;d;x]$ function, where s is the hash set, d the dummy element and x the element being inserted. The function calls the *put* auxiliary closure, which tries to insert x at a given offset, originally the hash of x, using a CAS. If this offset is already taken by a distinct value, potentially due to a collision, the *put* closure tries the next offset. (We do not resize: the *put* function loops if the table is full.) The function elems $[s;d]$ function returns the elements of s: that is, all the element distinct from the dummy element d. Occurrences of d are removed via a call to a dedicated filter_compact function.

This hash set can be used in parallel to insert values. However, in order to preserve disentanglement, the user should only insert values that were allocated before the beginning of the parallel

$$\begin{aligned}
&\mathsf{init} \triangleq \hat{\lambda}[d].\ \mathsf{alloc}\ d\ C\\
&\mathsf{elems} \triangleq \hat{\lambda}[s;d].\\
&\quad \mathsf{filter_compact}\ [s;d]\\
&\mathsf{add} \triangleq \hat{\lambda}[s;d;x].\\
&\quad \mathsf{let}\ put = \mu f.\lambda[i].\\
&\qquad \mathsf{if}\ (\mathsf{CAS}\ \ell\ i\ d\ x \lor \ell[i] == x)\\
&\qquad \mathsf{then}\ ()\\
&\qquad \mathsf{else}\ f\ [(i+1)\ \mathsf{mod}\ C]\ \mathsf{in}\\
&\quad put\ [\mathsf{hash}\ [x]\ \mathsf{mod}\ C]
\end{aligned}$$

$$\frac{\ulcorner d \notin A \urcorner \qquad A \circledcirc now}{\mathsf{wpm}\ (\mathsf{init}\ [d])\ \{\lambda\, s.\ \mathsf{hset}\ s\ d\ A\ \emptyset\ 1\}}$$

$$\frac{\mathsf{hset}\ s\ d\ A\ X\ 1}{\mathsf{wpm}\ (\mathsf{elems}\ [s;d])\ \{\lambda\, \ell.\ \exists \vec{w}.\ \ell \mapsto \vec{w}\ *\ \ulcorner \mathsf{deduped}\ \vec{w}\ X \urcorner\}}$$

$$\frac{\ulcorner x \in A \urcorner \qquad \mathsf{hset}\ s\ d\ A\ X\ p}{\mathsf{wpm}\ (\mathsf{add}\ [s;d;x])\ \{\mathsf{hset}\ s\ d\ A\ (X \cup \{x\})\ p\}}$$

$$\frac{\ulcorner d \notin \vec{v} \urcorner \qquad \ell \mapsto_p \vec{v}}{\mathsf{wpm}\ (\mathsf{dedup}\ [d;\ell])\ \{\lambda\, \ell'.\ \exists \vec{w}.\ \ell' \mapsto \vec{w}\ *\ \ell \mapsto_p \vec{v}\ *\ \ulcorner \mathsf{deduped}\ \vec{w}\ \vec{v} \urcorner\}}$$

Fig. 17. Case study: deduplication of an array by concurrent hashing

phase [Westrick 2022]. Indeed, the *put* auxiliary function does a CAS operation on an *a priori* arbitrary index, which may have been filled by a concurrent task. Our interface hence restricts insertions to a set of values that were allocated before the hash set itself. The representation predicate of a hash set s is written $\mathsf{hset}\ s\ d\ A\ X\ p$, where d is the dummy element, A a set of values that were witnessed as allocated before the hash set, X a set of values that were inserted, and p a fraction between 0 and 1. This predicate can be split and joined, allowing for parallel use.

$$\mathsf{hset}\ s\ d\ A\ (X_1 \cup X_2)\ (p_1 + p_2) \quad \dashv\vdash \quad \mathsf{hset}\ s\ d\ A\ X_1\ p_1\ *\ \mathsf{hset}\ s\ d\ A\ X_2\ p_2$$

Such a predicate is created by the init $[d]$ function. Its precondition requires a witness that a set A of values were allocated before the current timestamp, via the $A \circledcirc now$ assertion. Such an assertion can be obtained with the GetClock rule. The precondition also requires that the dummy element d is not an element of A. The postcondition produces a valid empty hash set with fraction 1. The specification of add $[s;d;x]$ requires a valid hash set with an arbitrary fraction, and that the element being inserted is in the authorized set of values. The specification of elems $[s;d]$ consumes a hash set with fraction 1 with content X and produces an array ℓ with content $\vec{w}$. The $\mathsf{deduped}\ \vec{w}\ X$ assertion asserts that $\vec{w}$ contains no duplicate and has the same elements as X:

$$\mathsf{deduped}\ \vec{w}\ X \quad \triangleq \quad \mathsf{NoDup}\ \vec{w}\ \land\ (\forall v.\ v \in \vec{w} \iff v \in X)$$

The proofs of the interface presented in Fig. 17 rest on the philosopher's lemmas. Intuitively, the hset predicate involves a cancellable invariant, storing an assertion $(s \mapsto \vec{w})@t_0$ where t_0 is the timestamp at which the hash set was allocated, and values $\vec{w} \subseteq A$. The predicate also involves an assertion $\uparrow t_0$ as well as an assertion $A \circledcirc t_0$, allowing to use the Objectivize rule.

Fig. 17 also presents the specification of the dedup $[d;\ell]$ function. This function deduplicates the array ℓ using our concurrent hash set. The function first creates a hash set s. Then, the function allocates a closure which, given an offset i, inserts the element $\ell[i]$ inside s. Next, the function calls the closure in parallel for every index of the array. Finally, the function returns the elements of the hash set. The precondition requires that ℓ points to an array $\vec{v}$ and the existence of a dummy element d that is not in the array. The postcondition returns a fresh location ℓ' pointing to an array $\vec{w}$ that is a deduplicated version of $\vec{v}$. Making of our hash set, the proof is straightforward: each task gets an assertion $\mathsf{hset}\ s\ d\ \vec{v}\ \emptyset\ (1/|\vec{v}|)$, enabling them to insert their element. At the end, the fractions of the hset predicate are joined, and the specification of elems concludes the proof.

6.4 Deduplication of a Lazy Collection via Concurrent Hashing

We cannot reuse the hash set of the previous section to deduplicate a lazy collection in parallel: its elements might not be allocated before the parallel phase, which could lead to entanglement during

$$\frac{\ulcorner n \neq 0 \urcorner}{\mathsf{wp}\ \langle t,\ \mathsf{init}\ [d]\rangle\ \{\lambda\, t'\, s.\ \bigstar_{i\in[0;n)}\ (\mathsf{lhset}\ s\ d\ i\ \emptyset\ t')\}} \qquad \frac{\ulcorner x \neq d \urcorner \qquad \mathsf{lhset}\ s\ d\ i\ X\ t}{\mathsf{wp}\ \langle t,\ \mathsf{add}\ [s; d; i; x]\rangle\ \{\lambda\, t'\ \_.\ \mathsf{lhset}\ s\ d\ i\ (X \cup \{x\})\ t'\}}$$

$$\frac{\bigstar_{i\in[0;n)}\ \mathsf{wpm}\ (k\ [i])\ \{\lambda\, v.\ \ulcorner v \neq d \urcorner\ *\ Q\ v\}}{\mathsf{wpm}\ (\mathsf{dedup\_lazy}\ [d; k; n])\ \{\lambda\, \ell.\ \exists \vec{v}\, \vec{w}.\ \ulcorner |\vec{v}| = n \wedge \mathsf{deduped}\ \vec{w}\ \vec{v} \urcorner\ *\ \ell \mapsto \vec{w}\ *\ \bigstar_{v\in\vec{v}}\ (Q\ v)\}}$$

Fig. 18. Case study: deduplication of a lazy collection by concurrent hashing

parallel insertion into the hash set. To address this issue, we implement and verify a more subtle hash set that can store elements allocated by concurrent tasks while having a small number of duplicates. After the parallel phase, we use the previous dedup function to get rid of the remaining duplicates. The hash set consists of a pair of arrays of the same size C, and takes inspiration from the *lock stripping* technique [Herlihy and Shavit 2012]. The first array is similar to the hash set of the previous section. The second stores *task identifiers*, represented as unboxed integers. The content of each offset in this second array governs the access to the same offset in the first one: when a task tries to insert an element at an offset, it first has to write its identifier with a CAS in the second array at this very offset. If the offset contains a different identifier, the task is not allowed to inspect the related offset in the first array. By doing so, dangerous races occur on the second array storing only unboxed values, preserving disentanglement. The first counterpart is that duplicates may occur in the hash set. Their number is bounded by the number of tasks. The second counterpart is that read-write races occur on the first array, and the proof that they are disentangled relies on a subtle invariant: the full power of DisLog is needed.

We show in Fig. 18 the specifications of init and add, where n is the number of tasks that will use the hash set. They involve a representation predicate $\mathsf{lhset}\ s\ d\ i\ X\ t$, which asserts that the hash set s with dummy element d can be used with the task identifier i, contains elements in X and is valid at timestamp t. The add function must be called with the correct identifier. The key idea is that this representation predicate is monotonic with respect to the precedence pre-order. We derive similar specifications in DisLog+ using the Conversion rule, confining the timestamp-related reasoning.

Fig. 18 also presents the specification of our deduplication function $\mathsf{dedup\_lazy}\ [d; k; n]$. The precondition requires that the lazy collection k is safe between index 0 and n, that it returns a value that is not the dummy element, and that satisfies a given postcondition Q. The postcondition returns a location ℓ and guarantees the existence of two arrays $\vec{v}$ and $\vec{w}$ such that $\vec{v}$ contains n elements and $\vec{w}$ is a deduplicated version of $\vec{w}$. The postcondition then asserts that the returned location ℓ points to $\vec{w}$, and that for every value v in $\vec{v}$, the assertion $Q\ v$ holds.

7 MECHANIZATION

All our results are mechanized in the Coq proof assistant [Coq Development Team 2022] using Iris [Jung et al. 2018] and its dedicated Proof Mode [Krebbers et al. 2018]. Our mechanization is available in the supplementary material [Anon. 2023]. Rounding and excluding comments, the definition of the language takes 1200LOC, the proofs of the two logics and their soundness theorems, 4600LOC, and the verification of case studies 3700LOC. We provide tactics to be used while reasoning with the two logics, and automation to DisLog+ thanks to the Diaframe library [Mulder et al. 2022].

8 RELATED WORK

Disentanglement. There has been a variety of work on disentanglement [Arora et al. 2021, 2023; Guatto et al. 2018; Raghunathan et al. 2016; Westrick 2022; Westrick et al. 2022, 2020]. Much of this work focuses on dynamic techniques that exploit disentanglement for improved efficiency, especially for parallel memory management. In particular, Arora et al. [2021] developed a provably efficient memory manager for functional programs based on disentanglement, and Arora et al. [2023] extended this approach to support unrestricted effects by accounting for the cost of entanglement. These works rely on disentanglement for efficiency and scalability, and leave the task of reasoning about disentanglement to the programmer. The first formal definition for disentanglement was given by Westrick et al. [2020] using traces of memory operations, and Westrick et al. [2022] developed a semantics which detects entanglement during execution. Our semantics for disentanglement is similar in the sense that it becomes stuck when entanglement occurs. In this context, the logics developed in this paper statically verify that execution never becomes stuck.

Linearity and Concurrency. Our "plain vanilla" DisLog+ (i.e., without fractional write-only assertions and philosopher's lemmas) is related to reasoning approaches establishing race freedom by a linear treatment of resources. These approaches comprise type systems for the π-calculus [Igarashi and Kobayashi 2001, 2004] as well as session type systems [Balzer and Pfenning 2017; Caires et al. 2016; Jacobs et al. 2022; Lindley and Morris 2015; Toninho et al. 2013]. The latter are based on a Curry-Howard correspondence established between linear logic and the session-typed π-calculus [Caires and Pfenning 2010; Wadler 2012]. Most closely related to our work in terms of employed techniques is the work by Jacobs et al. [2022], which mechanizes safety of a session-typed language in Coq, where safety encompasses freedom of memory leaks and deadlocks. The authors introduce the notion of a connectivity graph, which is acyclic by construction due to linearity, and use concurrent separation logic to prove acyclicity-preservation graph transformations. Our work, in contrast, is not confined to a linear setting.

Separation Logics. Multiple Iris-based concurrent separation logic were developed [Bizjak et al. 2019; Chajed et al. 2021; Krogh-Jespersen et al. 2020]. Among them, logics targeting weak-memory models [Dang et al. 2020; Kaiser et al. 2017; Mével et al. 2020] inspired DisLog+. Indeed, they all build a high-level logic on top of a low-level logic using monotonicity arguments. In their case, assertions are monotonic predicates over the view of the memory: assertions remain valid even after observing additional memory events. In their case, the view ordering of the memory is a pure assertion. We generalize their approach to a pre-order within *iProp*. Moine et al. [2023] present a separation logic to reason about heap space for a sequential language with garbage collection. Their language is similar to the sequential subset of DisLang. In particular, they also make the difference between top-level functions and heap-allocated closures. Disentanglement is closely related to garbage collection: disentanglement ensures in particular that locations occurring in the program are always safe to read for a task-local garbage collector: this is reminiscent of the free variable rule [Felleisen and Hieb 1992]. Outside the Iris world, Fu et al. [2010] present a concurrent separation logic with temporal reasoning. Contrary to them, our notion of time only relates to the pre-order induced by the fork-join structure of the program.

9 CONCLUSION AND FUTURE WORK

Disentanglement is an important property for parallel performance, and prior work leaves the challenge of reasoning about disentanglement to the programmer. We address this challenge by presenting DisLog, the first program logic to formally verify that a program is disentangled. Additionally, we present DisLog+, which allows for mostly standard separation logic proofs and offers proofs of disentanglement "for free" for many programs. Using these logics, we prove disentanglement for a number of examples, including several lock-free data structures. Our experience with DisLog and DisLog+ is that the effort required to prove disentanglement is often small and can be confined to the daring parts of the program. In future work, we plan to develop a type system to automatically infer disentanglement where possible. We hope that a semantic type soundness approach [Timany et al. 2022] making use of DisLog could be used to prove such a system sound.

REFERENCES

Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016. Dag-calculus: a calculus for parallel computation. In *International Conference on Functional Programming (ICFP)*. 18–32. https://doi.org/10.1145/2951913.2951946

Anon. 2023. Supplementary Material. Submitted together with this paper.

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press. http://www.cambridge.org/9780521033114

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space-Efficient Parallel Functional Programming. *Proc. ACM Program. Lang.* 5, POPL, Article 18 (jan 2021), 33 pages. https://doi.org/10.1145/3434299

Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. Efficient Parallel Functional Programming with Effects. *Proc. ACM Program. Lang.* 7, PLDI, Article 170 (jun 2023), 26 pages. https://doi.org/10.1145/3591284

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 37:1–37:29. https://doi.org/10.1145/3110281

Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing Obligations in Higher-Order Concurrent Separation Logic. *Proc. ACM Program. Lang.* 3, POPL, Article 65 (jan 2019), 30 pages. https://doi.org/10.1145/3290378

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. http://www.cs.uwm.edu/~boyland/papers/permissions.pdf

Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science* 375, 1–3 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *21th International Conference onf Concurrency Theory (CONCUR) (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423. https://doi.org/10.1017/S0960129514000218

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *Symposium on Operating Systems Design and Implementation*. 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

Coq Development Team. 2022. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.7313584

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 34:1–34:29. https://hal.inria.fr/hal-02351793/

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* 103, 2 (1992), 235–271. https://www2.ccs.neu.edu/racket/pubs/tcs92-fh.pdf

Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326. https://doi.org/10.1007/s002240000120

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR 2010 - Concurrency Theory*, Paul Gastin and François Laroussinie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–402.

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science, Vol. 4807)*. Springer, 19–37. http://dx.doi.org/10.1007/978-3-540-76637-7_3

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 81–93. https://doi.org/10.1145/3178487.3178494

Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Atsushi Igarashi and Naoki Kobayashi. 2001. A Generic Type System for the Pi-calculus. In *8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 128–141. https://doi.org/10.1145/360204.360215

Atsushi Igarashi and Naoki Kobayashi. 2004. A Generic Type System for the Pi-calculus. *Theoretical Computer Science* 311, 1-3 (2004), 121–163. https://doi.org/10.1016/S0304-3975(03)00325-6

Iris Development Team. 2023. `iris.base_logic.lib.gen_heap`. https://plv.mpi-sws.org/coqdoc/iris/iris.base_logic.lib.gen_heap.html.

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Connectivity Graphs: a Method for Proving Deadlock Freedom Based on Separation Logic. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–33. https://doi.org/10.1145/3498662

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

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *European Conference on Object-Oriented Programming (ECOOP)*. 17:1–17:29. https://people.mpi-sws.org/~dreyer/papers/iris-weak/paper.pdf

Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP, Article 77 (jul 2018), 30 pages. https://doi.org/10.1145/3236772

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 12075)*. Springer, 336–365. https://iris-project.org/pdfs/2020-esop-aneris-final.pdf

Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Computer Journal* 6, 4 (Jan. 1964), 308–320.

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *24th European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 7, POPL, Article 25 (jan 2023), 30 pages. https://doi.org/10.1145/3571218

MPL Development Team. 2022. The MaPLe (MPL) compiler v0.3. https://github.com/MPLLang/mpl.

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 809–824. https://doi.org/10.1145/3519939.3523432

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP, Article 96 (June 2020), 29 pages. http://cambium.inria.fr/~fpottier/publis/mevel-jourdan-pottier-cosmo-2020.pdf

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 290–310.

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (May 2007), 271–307. http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/concurrency.pdf

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy E. Blelloch. 2016. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 392–406. https://doi.org/10.1145/2951913.2951935

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 8410)*. Springer, 149–168. http://cs.au.dk/~birke/papers/icap-conf.pdf

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness. (2022). https://iris-project.org/pdfs/2022-submitted-logical-type-soundness.pdf Unpublished.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *22nd European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 350–369. https://doi.org/10.1007/978-3-642-37036-6_20

VerifyThis. 2022. Challenge 3 - The World's Simplest Lock-Free Hash Set. https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Verify%20This/Challenges2022/verifyThis2022-challenge3.pdf

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*. 76–90. https://cs.au.dk/~birke/papers/2021-ms-queue-final.pdf

Philip Wadler. 2012. Propositions as Sessions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 273–286. https://doi.org/10.1145/2364527.2364568

Sam Westrick. 2022. *Efficient and Scalable Parallel Functional Programming Through Disentanglement*. Ph. D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~swestric/22/thesis.pdf

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection with Near-Zero Cost. *Proc. ACM Program. Lang.* 6, ICFP, Article 115 (aug 2022), 32 pages. https://doi.org/10.1145/3547646

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 47 (jan 2020), 32 pages. https://doi.org/10.1145/3371115