Disentanglement with Futures, State, and Interaction

JATIN ARORA, Carnegie Mellon University, USA
STEFAN K. MULLER, Illinois Institute of Technology, USA
UMUT A. ACAR, Carnegie Mellon University, USA

Recent work has proposed a memory property for parallel programs, called disentanglement, and showed that it is pervasive in a variety of programs, written in different languages, ranging from C/C++ to Parallel ML, and showed that it can be exploited to improve the performance of parallel functional programs. All existing work on disentanglement, however, considers the "fork/join" model for parallelism and does not apply to "futures", the more powerful approach to parallelism. This is not surprising: fork/join parallel programs exhibit a reasonably strict dependency structure (e.g., series-parallel DAGs), which disentanglement exploits. In contrast, with futures, parallel computations become first-class values of the language, and thus can be created, and passed between functions calls or stored in memory, just like other ordinary values, resulting in complex dependency structures, especially in the presence of mutable state. For example, parallel programs with futures can have deadlocks, which is impossible with fork-join parallelism.

In this paper, we are interested in the theoretical question of whether disentanglement may be extended beyond fork/join parallelism, and specifically to futures. We consider a functional language with futures, Input/Output (I/O), and mutable state (references) and show that a broad range of programs written in this language are disentangled. We start by formalizing disentanglement for futures and proving that purely functional programs written in this language are disentangled. We then generalize this result in three directions. First, we consider state (effects) and prove that stateful programs are disentangled if they are race free. Second, we show that race freedom is sufficient but not a necessary condition and non-deterministic programs, e.g. those that use atomic read-modify-operations and some non-deterministic combinators, may also be disentangled. Third, we prove that disentangled task-parallel programs written with futures are free of deadlocks, which arise due to interactions between state and the rich dependencies that can be expressed with futures. Taken together, these results show that disentanglement generalizes to parallel programs with futures and, thus, the benefits of disentanglement may go well beyond fork-join parallelism.

CCS Concepts: • **Software and its engineering** → **Runtime environments**; **Parallel programming languages**; **Garbage collection**.

Additional Key Words and Phrases: Disentanglement

ACM Reference Format:
Jatin Arora, Stefan K. Muller, and Umut A. Acar. 2024. Disentanglement with Futures, State, and Interaction. *Proc. ACM Program. Lang.* 8, POPL, Article 53 (January 2024), 31 pages. https://doi.org/10.1145/3632895

1 INTRODUCTION

Functional programming offers important correctness benefits to parallel programmers, allowing them to side step the challenges of pesky data races by writing pure code or guiding them with powerful type systems so that they can use effects judiciously. Historically, these correctness benefits meant little, because parallel functional programs delivered poor performance in comparison to

Authors' addresses: Jatin Arora, jatina@andrew.cmu.edu, Carnegie Mellon University, Pittsburgh, USA; Stefan K. Muller, smuller2@iit.edu, Illinois Institute of Technology, Chicago, USA; Umut A. Acar, umut@cmu.edu, Carnegie Mellon University, Pittsburgh, USA.

ACM 2475-1421/2024/1-ART53
https://doi.org/10.1145/3632895

their procedural counterparts. Recent work on disentanglement has broken a new path towards bridging the correctness and performance gap by exploiting the "disentanglement hypothesis", the idea that most objects in a fork-join program are never touched by concurrently executing threads. Recent research has shown that when exploited to manage memory efficiently, the disentanglement hypothesis allows parallel functional programs to perform well, including at scale [Arora et al. 2021, 2023; Westrick et al. 2020].

Research on disentanglement originates with a key theoretical result that prove that all objects in a race-free program are disentangled [Westrick et al. 2020]. Later work extended this theory by showing that an object gets entangled only if it participates in determinacy races [Arora et al. 2023]. Because races typically lead to correctness bugs [Adve 2010; Allen and Padua 1987; Bocchino et al. 2011, 2009; Boehm 2011; Emrath et al. 1991; Mellor-Crummey 1991; Netzer and Miller 1992; Steele Jr. 1990], they are rare in parallel programs, leading to the hypothesis that most objects in a fork-join parallel programs are disentangled.

The work on disentanglement has delivered promising results but it has a significant limitation: it applies to only fork-join (nested) parallel programs. Although the fork-join model is reasonably expressive, it cannot express parallel programs with more complex dependencies and parallelism techniques such as pipelining [Blelloch and Reid-Miller 1999; Singer et al. 2019a; Spoonhower et al. 2009]. More generally fork-join parallelism does not help with asynchronous, interactive applications [Acar et al. 2016; Muller et al. 2017, 2018, 2019; Singer et al. 2020b]. Recognizing these limitations of fork-join parallelism, many modern programming languages and frameworks support a more powerful form of parallelism called futures. Invented in the 1970s [Baker and Hewitt 1977], futures allow you to create a parallel task and demand the result from the task at a later time when needed (hence the name "future"). Unlike fork-join which is merely a control-flow construct, futures are first-class values, making parallelism a "first-class citizen" of the programming language, which in turn contributes to their expressive power: futures can express data-dependent parallelism, pipelining, asynchrony, and interaction. Perhaps unsurprisingly, languages such as Concurrent Haskell [Marlow 2011; Peyton Jones et al. 2008], Habanero Java [Imam and Sarkar 2014], Parallel ML (Manticore) [Fluet et al. 2008, 2011; Ohori et al. 2018; Spoonhower et al. 2009], PriML [Muller et al. 2020; Muller and Acar 2016; Muller et al. 2017, 2019; Singer et al. 2020b], OCaml [Dolan et al. 2018a; LWT 2022], Rust [Rust Team 2019], and TPL (a .NET library) [Leijen et al. 2009], have all embraced futures as a means of supporting more expressive parallelism and interaction. This raises the question of whether disentanglement can be generalized to futures, and if so, how broadly—can the many interesting applications of futures be disentangled.

In this paper, we generalize disentanglement to include futures. We consider a functional language that supports futures, I/O (Input/Output), and (mutable) state in the form of references. The language combines futures, which are first-class threads, with state and I/O, supporting thus the expression of interactive and asynchronous programs. We show that a broad range programs written in the language are disentangled. The language's expressiveness also raises questions of safety, such as deadlock-freedom, between (concurrent) futures.

To establish disentanglement for this language, we start by restricting our attention to a pure functional fragment with I/O, and prove that programs written in this fragment are disentangled. We then generalize this result by considering the full language with mutable state and prove that determinacy race free ([Netzer and Miller 1992]) parallel programs with futures are disentangled. Because races typically cause incorrect behavior at scale (e.g., [Adve 2010; Boehm 2011]), they are frequently classified as a correctness bug. Additionally, motivated by hardware support for atomic read-modify-write operations, we generalize this result by showing that certain classes of races, which involve races caused by atomic operations on machine words, do not harm disentanglement.

These results therefore establish that a reasonably broad class of parallel programs written with futures are disentangled.

Because they make parallelism a “first class citizen” of a programming language, futures have great expressive power, especially when combined with mutable state. This great power has a cost: using futures and state, the programmers can create circular dependencies, which lead to deadlock. For example, two futures can discover each other through shared mutable state and attempt to synchronize with each other, leading to deadlock [Cogumbreiro et al. 2017; Voss et al. 2019]. In this paper, we show that all disentangled programs are deadlock-free.

These results are not straightforward extensions of prior work on disentanglement for fork-join programs due to the complex dependency structure introduced by futures. Unlike, fork-join programs, whose dependencies can be represented as series-parallel DAGs, the synchronization patterns in programs with futures are more flexible. A series-parallel DAG forbids synchronization between concurrently executing computations and allows us to infer disentanglement directly from the DAG. But with futures, a computation may synchronize with a future essentially any time, creating a dependency structure which is dynamic and data dependent. To tackle this challenge, we track the memory objects owned by each future (via allocations) by organizing memory as a tree defined by spawned futures. We then treat synchronization with a concurrent future as a "one-way dependency" that reorganizes the memory tree to reflect the new dependency structure. This rewriting allows us to define and reason about disentanglement for futures.

The specific contributions of this paper include the following.

- A trace-based definition of disentanglement that accounts for futures.
- A core calculus for programs with futures, mutable state, and I/O for generating traces that can then be used to check for disentanglement.
- A proof that pure functional and purely interactive programs with futures are disentangled.
- A proof that race free programs with futures and mutable state satisfy disentanglement.
- Formal definition of weak races and proof that weakly race free programs are disentangled.
- Proof that disentangled programs ensure an important safety property: deadlock-freedom.
- Applications demonstrating the breadth of disentanglement using a variety of techniques including pipelined parallelism, interaction, and asynchrony.

2 DISENTANGLEMENT WITH FUTURES

In this section, we define disentanglement for programs of a language that supports futures, I/O (input/output), and mutable references. We state disentanglement using the computation tree, a tree that captures the control flow dependencies between the program threads and their memory actions such as allocations, reads, and writes. The computation tree is generated by the language semantics at each step of program evaluation. At a high level, we say that a computation tree satisfies disentanglement if the allocation actions of concurrent threads are oblivious to each other and a program evaluation satisfies disentanglement if the computation tree satisfies disentanglement at each step of the evaluation.

The language semantics models parallelism by interleaving the evaluation of futures and their continuations. During the parallel evaluation of a future and its continuation, the semantics represents their actions as parallel in the computation tree. However, once the future finishes its evaluation, the semantics applies a join transformation on the tree. This transformation rewrites the computation tree to sequence the future’s actions with the continuation’s actions, capturing the idea that after the future has completed, its actions no longer need to be considered concurrent to the actions of the continuation. As we show in subsequent sections, the join transformation enables us to reason about disentanglement for futures.

$$
\begin{array}{rlll}
\textit{Future names} & a, b & & \\
\textit{Memory Locations} & \ell & & \\
\textit{Types} & \tau & ::= & \texttt{bool} \mid \texttt{nat} \mid \tau \rightarrow \tau \mid \tau\ \texttt{fut} \\
\textit{Storables} & s & ::= & \texttt{true} \mid \texttt{false} \mid n \mid \texttt{fun}\ f\ x\ \texttt{is}\ e \mid \texttt{fcell}[a] \mid \texttt{ref}\ v \\
\textit{Values} & v & ::= & \ell \\
\textit{Memory} & \mu & \in & \textit{Locations} \rightharpoonup \textit{Storables} \\
\textit{Expressions} & e & ::= & v \mid s \mid x \mid e\ e \mid \texttt{fut}(e) \mid \texttt{fpoll}(e) \mid \texttt{get}(e) \mid \\
 & & & \texttt{ref}\ e \mid\ !\,e \mid e_1 := e_2 \mid \texttt{in\_nat}() \mid \texttt{out\_nat}(e) \\
\textit{Future Map} & \Delta & ::= & \emptyset \mid \Delta\ [a \blacktriangleright e] \mid \Delta\ [a \triangleright v] \\
\textit{Action Trace} & t & ::= & \bullet \mid \mathbf{A}\ell \Leftarrow s \mid \mathbf{R}\ell \Rightarrow s \mid \mathbf{U}\ell \Leftarrow s \mid \mathbf{F}\ell \Rightarrow v \mid t \oplus t \\
\textit{Computation Trees} & T & ::= & [t] \mid t \oplus (T \otimes_a T) \mid t \oplus_a T
\end{array}
$$

Fig. 1. Syntax of λ^U

2.1 Syntax

Our language contains constructs for functions, references, futures, and support for input/output operations. Figure 1 presents the syntax of the language.

Types. The types include booleans, natural numbers, function types and the type τ `fut` for futures which evaluate an expression of type τ.

Storables and memory locations. To define disentanglement and precisely account for the actions on memory, the language distinguishes between ***storables*** and ***locations***. Storables include numbers, named recursive functions, and future cells. The language steps storables to locations and uses a memory store μ to map location to storables. A storable at a location may refer/point to other locations. We use $\mathsf{Loc}(s)$ to denote the locations referred to by the storable s. We represent locations with variables like ℓ, use $\mu(\ell)$ to denote the storable at location ℓ, and use $\mu[\ell \hookrightarrow s]$ to denote the allocation of location ℓ in the memory store μ (with the implicit requirement that $\ell \notin \mathrm{dom}(\mu)$). Locations are the only irreducible form of the language. In our dynamics, we use this distinction between storables and locations to track all the program allocations.

Expressions. The expressions include the usual constructs for functions and references. The expression $\texttt{fut}(e)$ spawns a future to evaluate expression e. The language dynamics gives each future a name like a, b and other similar variables. For each future, the language allocates a future cell, which can be used by other threads to either 1) block on the future with the expression `get`, which returns the future's value when it finishes, or 2) poll the future with the expression `fpoll`, which returns true or false depending on whether the future has terminated. We denote the future cell for future a as $\texttt{fcell}[a]$. The expressions $\texttt{in\_nat}()$ and $\texttt{out\_nat}(e)$ support input and output operations for natural numbers. The language models an input as a non-deterministic step to a number and an output as a deterministic step that reads the argument and returns. This model captures the memory effects associated with these operations, which is sufficient for our goal of defining and reasoning about disentanglement.

2.2 Computation Trees

The computation tree records the memory actions taken during evaluation and organizes them according to their control flow dependencies. Each node of the tree represents a memory ***action*** taken by the program, which may be one of the following:

- $\mathbf{A}\ell \Leftarrow s$ is the allocation of location ℓ initialized with storable s.
- $\mathbf{R}\ell \Rightarrow s$ is a memory lookup (read) at location ℓ which returns storable s.
- $\mathbf{U}\ell \Leftarrow s$ is an update (write) which stores storable s at a mutable location ℓ.

- $\mathbf{F}\ell \Rightarrow v$ is a *synchronization* with the future whose future cell is at location ℓ which returns v.

The edges of the tree represent sequential ordering between the actions. For simplicity, we fuse sequentially taken memory actions into a single node of the tree and call it an action trace. An ***action trace*** contains a (possibly empty) series of actions composed by the operator $\oplus$, where the connective $\oplus$ emphasizes that actions within a trace are taken sequentially. Figure 1 shows the syntax of action traces and computation trees. We denote action traces with a lowercase variable like t and computation trees with an upper case variable like T.

When the evaluation starts, the computation tree only contains a single node. When a thread spawns a future, we add two leaves to the tree, one for storing the actions of the future and the second for storing the actions of the thread after the spawn. After a thread spawns a future, we refer to it as the ***continuation*** of that future.

A computation tree of the form $[t]$ is a leaf and represents a sequential evaluation that performs actions in trace t. A computation tree of the form $t \oplus (T_1 \otimes_a T_2)$ represents an evaluation that performs the actions in trace t before spawning a future named a. The tensor $\otimes_a$ is called the ***spawn point*** of future a. The spawn point denotes that the actions of the future are in subtree T_1, actions of the continuation are in subtree T_2, and the respective actions are taken in parallel. A computation tree of the form $t \oplus_a T$ represents an evaluation that spawned a future named a, but the future has finished and "joined" with its continuation. The operator $\oplus_a$ is called the ***join point*** of future a. We describe the join operation in Section 2.4.

Small example. Figure 2 shows two computation trees of an evaluation where a thread *main* spawns a future a, which in turn spawns future b, and then thread *main* synchronizes with the future a to retrieve its result (location ℓ''). The figure shows two trees but we return to the right side tree later in the section. In the left tree, each box denotes an action trace and the edges between boxes denote the edges of the tree. The figure labels each box with the thread that performs its actions. After the *main* thread spawns the future a, it allocates the future cell $\texttt{fcell}[a]$ at location ℓ (see $\mathbf{A}\ell \Leftarrow \texttt{fcell}[a]$); the thread can now use location ℓ to synchronize with the future. The future a spawns future b and similarly allocates a cell for it at location ℓ' (see $\mathbf{A}\ell' \Leftarrow \texttt{fcell}[b]$). The future a then allocates some storable s at location ℓ'' (see $\mathbf{A}\ell'' \Leftarrow s$), which is the return value of the future. Its continuation (thread *main*) synchronizes with it and receives location ℓ'' (see $\mathbf{F}\ell \Rightarrow \ell''$).

2.3 Disentanglement

At a high level, disentanglement restricts concurrent threads from accessing each other's allocations. In the context of futures, disentanglement implies that a continuation is prohibited from accessing a future's allocations as long as the future is executing. However, if the continuation synchronizes with the future, disentanglement lifts the restrictions and allows the continuation to freely access the future's allocations. This is because a synchronization between the continuation and the future returns only after the future has terminated, rendering them non-concurrent.

We define disentanglement using the computation tree. The computation tree arranges the memory actions of program threads according to their control flow dependencies, i.e., it orders sequentially dependent memory actions in an ancestor-descendant relationship and keeps concurrent memory actions unrelated. A computation tree satisfies disentanglement when every action in the tree only mentions locations that are allocated by the ancestor actions of that action. An evaluation satisfies disentanglement if its computation tree maintains disentanglement at each step.

We formalize disentanglement for a tree with an inductive process using the judgement $A \vdash T\ de$. The judgement's context A stores the set of locations allocated by the ancestor actions of tree T. The judgement checks that every location mentioned by an action of tree T is either present in

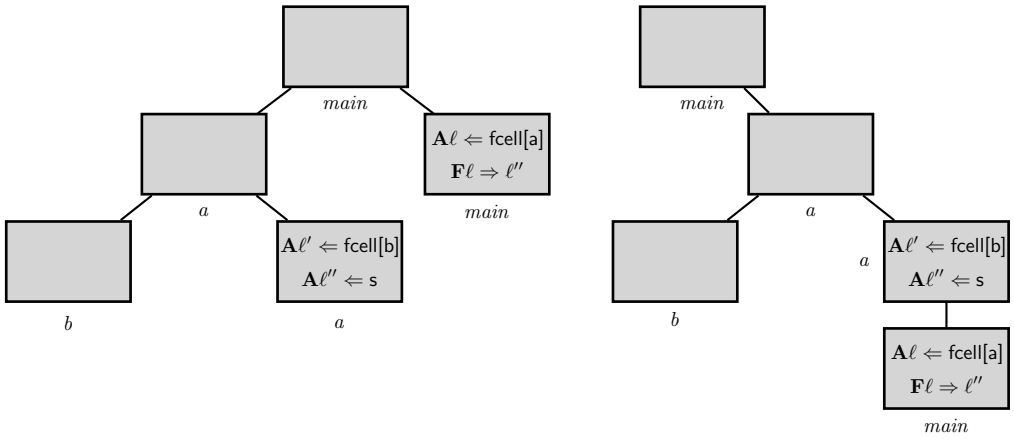

Fig. 2. Two computation trees representing an evaluation where a thread *main* spawns a future named a, which in turn spawns future b, and then the main thread synchronizes with future a to retrieve its result (location ℓ''). We denote each node of the tree with a box containing a possibly empty action trace. The labels on the boxes denote the thread that performed the actions. The left and right trees show the tree structure without and with the join transformation. Without the join transformation, the left tree (mis-)characterizes the computation as entangled, as it represents the allocation ℓ'' of future a to be concurrent to the synchronized access of location ℓ'' by thread *main*. With the join transformation, the right tree correctly characterizes the computation to be disentangled as the allocation action is an ancestor of the synchronization action.

$\boxed{A \vdash t\ de}$

$$\frac{}{A \vdash \bullet\ de} \qquad \frac{\mathsf{Loc}(s) \subseteq A}{A \vdash (\mathbf{A}\ell \Leftarrow s)\ de} \qquad \frac{\ell \in A \qquad \mathsf{Loc}(s) \subseteq A}{A \vdash (\mathbf{R}\ell \Rightarrow s)\ de} \qquad \frac{\ell \in A \qquad \mathsf{Loc}(v) \subseteq A}{A \vdash (\mathbf{F}\ell \Rightarrow v)\ de} \qquad \frac{\ell \in A \qquad \mathsf{Loc}(s) \subseteq A}{A \vdash (\mathbf{U}\ell \Leftarrow s)\ de}$$

$$\frac{A \vdash t_1\ de \qquad A \cup \mathsf{A}(t_1) \vdash t_2\ de}{A \vdash t_1 \oplus t_2\ de}$$

$\boxed{A \vdash T\ de}$

$$\frac{A \vdash t\ de}{A \vdash [t]\ de} \qquad \frac{A \vdash t\ de \qquad A \cup \mathsf{A}(t) \vdash T_1\ de \qquad A \cup \mathsf{A}(t) \vdash T_2\ de}{A \vdash t \oplus (T_1 \otimes_a T_2)\ de} \qquad \frac{A \vdash t\ de \qquad A \cup \mathsf{A}(t) \vdash T\ de}{A \vdash t \oplus_a T\ de}$$

Fig. 3. The figure defines the judgements $A \vdash T\ de$ and $A \vdash t\ de$, which formalize disentanglement for a tree T and a node t respectively. The context A contains locations allocated by the ancestor actions of the tree/node.

the context A, or is allocated by some ancestor action in tree T. For a full tree T, if the judgement $\emptyset \vdash T\ de$ holds (i.e., with the empty context) then the tree T satisfies disentanglement.

Figure 3 defines the rules for the judgement $A \vdash T\ de$ for the tree and judgement $A \vdash t\ de$ for a node t of the tree. When the tree is of the form $[t]$, the judgement checks the node t which is an action trace. If the trace t is of the form $t_1 \oplus t_2$, then its rule checks the trace t_1 and subsequently checks the trace t_2 after extending the context A with the allocations in trace t_1. This is because actions in trace t_1 are ancestors to actions in trace t_2.

$$\frac{T_1 = [t_1]}{\bowtie (a, T_1, T_2) = t_1 \oplus_a T_2}\ \textsc{Leaf} \qquad \frac{T_1 = t_1 \oplus_b T_1'}{\bowtie (a, T_1, T_2) = t_1 \oplus_b \bowtie (a, T_1', T_2)}\ \textsc{Join Point}$$

$$\frac{T_1 = t_1 \oplus (T_1' \otimes_b T_1'')}{\bowtie (a, T_1, T_2) = t_1 \oplus (T_1' \otimes_b \bowtie (a, T_1'', T_2))}\ \textsc{Spawn Point}$$

Fig. 4. Function Join

The rule for the allocation action $\mathbf{A}\ell \Leftarrow s$ checks that all locations in storable s are in the set A. The rule for the read action $\mathbf{R}\ell \Rightarrow s$ checks that both the location ℓ and locations in storable s are in set A. Similarly, the rule for the update action inspects both the location and the new storable. The rule for the synchronization action ($\mathbf{F}\ell \Rightarrow v$) checks the location ℓ and the locations in value v.

The rule for the form $t \oplus (T_1 \otimes_a T_2)$ checks that the trace t is disentangled ($A \vdash t\ de$) and inspects the subtrees T_1 and T_2 after extending the context A with locations allocated by the trace t (denoted as $A(t)$). This is because actions of trace t are ancestors of actions in subtrees T_1 and T_2. Importantly, the rule does not include the locations allocated by tree T_2 to check tree T_1 and vice-versa. This is because their actions are not in an ancestor-descendant relationship.

When the tree is of the form $t \oplus_a T$, the judgement checks the trace t and the tree T. In a tree of this form, actions of trace t are ancestors to actions in tree T. Thus, the rule for this form checks the tree T after extending the context A with allocations of the trace t.

2.4 Joins

After a future terminates, its continuation can access its allocations without violating disentanglement. Our semantics represents this in the computation tree by transforming the tree after a future terminates. The semantics rearranges the tree such that memory actions of the future become ancestors of its continuation's actions, and they appear sequentially ordered. The semantics performs this ***join transformation*** at an evaluation step called ***join***. The join step is only a tool for reasoning about disentanglement and, in no way, affects the actual parallelism of the program.

Example. To illustrate the join transformation, we draw two trees in Figure 2. The two trees represent an evaluation in which a thread *main* spawns future a, which subsequently spawns future b, and then thread *main* synchronizes with the future to retrieve its result. The left tree does not incorporate the join transformation whereas the right tree does. In the left tree, the synchronization action by thread *main* ($\mathbf{F}\ell \Rightarrow \ell''$) and the allocation action ($\mathbf{A}\ell'' \Leftarrow s$) by future a are positioned concurrently in the tree, i.e., they are not in an ancestor-descendant relationship. Consequently, since the allocation of location ℓ'' is not an ancestor of the synchronization action, the left tree mistakes the evaluation as violating disentanglement. In contrast, the right tree satisfies disentanglement because of the join transformation. By sequencing the future's actions with those of the continuation, the join transformation ensures that the allocation action ($\mathbf{A}\ell'' \Leftarrow s$) for location ℓ'' becomes an ancestor of the synchronization action ($\mathbf{F}\ell \Rightarrow \ell''$).

By sequencing the future's actions before the continuation's actions, the join transformation represents that once a future finishes, it is no longer concurrent with the continuation. However, it is essential to note that any futures spawned by the completed future may still be executing and remain concurrent with the continuation. As a result, it is crucial for the join transformation to not sequence their actions with the continuation.

Join Function. Figure 4 shows the join transformation with the function $\bowtie$. The function takes arguments a, T_1, T_2, where a is the future that finished and trees T_1 and T_2 are the children of the

spawn point of future a, i.e., tree T_1 is the tree containing the future's actions, and tree T_2 is the tree containing the continuation's actions. To perform the join, the function recurses down the tree T_1, following the actions of the future until it reaches the leaf. This leaf marks the end of the future's actions because it has finished. Then, the function sticks tree T_2 as a descendant of that leaf, making all of the future's actions ancestors of the continuation's actions. The function does not change the relationship between any actions corresponding to other threads in trees T_1 and T_2.

In the leaf case, when tree T_1 is a leaf of form $[t_1]$, the function returns $t_1 \oplus_a T_2$, where the operator $\oplus_a$ marks the join point of future a. For tree T_1 of the form $t_1 \oplus_b T_1'$, which represents the join point of future b, the function returns the tree $t_1 \oplus_b \bowtie (a, T_1', T_2)$. This resulting tree maintains the relationship between trace t_1 and tree T_1', and also incorporates the join of tree T_2 with tree T_1'. For tree T_1 of the form $t_1 \oplus (T_1' \otimes_b T_1'')$, which represents the spawn point of future b, the function recurses on subtree T_1'' and leaves subtree T_1' unchanged. This is because the tree T_1' contains the actions of future b and the actions of the future a are in tree T_1''. By recursing down tree T_1'', the function makes all the actions of the future a ancestors to the actions of tree T_2. We give another example to illustrate this function in the Appendix.

Fork/Join. We note that our join here is similar to a "join" in a fork-join computation but there are some important differences. The join in fork-join is two-way because two sibling tasks finish their execution and join with each other. It requires that all tasks nested within the joining tasks also terminate before the join can proceed. As a result, join effectively eliminates all concurrency and parallelism within its scope. On the other hand, in the case of futures, the join is one-way because a future finishes and joins into its continuation. Furthermore, tasks spawned by the joining future can escape its scope. This key difference allows for concurrency and parallelism even after the join, as the spawned tasks can execute independently of the future and its continuation.

We can indeed observe these differences by considering the join function. For fork-join, a corresponding join function, say function J, is $\mathsf{J}([t_1], [t_2]) = [t_1 \oplus t_2]$. This function is relatively simple because both its arguments are guaranteed to be leaves in the computation tree. Any tasks nested within their scope have finished. In contrast, the join function for futures operates on trees. This is because the continuation has not finished and the future, even though itself has finished, may have spawned other futures which are still executing. This ability of futures to spawn futures which continue to execute concurrently beyond the joining point introduces additional challenges for reasoning about concurrency and proving disentanglement.

2.5 Language Semantics

Our operational semantics steps a program state consisting of four components: (i) a future map Δ tracking the evaluation of futures, (ii) a memory store μ mapping locations to storables, (iii) a computation tree T, and (iv) an expression e. We write a program state as $(\Delta\,;\mu\,;T\,;e)$. Figure 5 shows the rules for the semantics.

Allocations and functions. The allocation rule Alloc extends the memory μ with location ℓ mapped to storable s and records it in the leaf $[t]$ as the allocation action $\mathbf{A}\ell \Leftarrow s$. Rules AppSL and AppSR for function application step the function and the argument respectively. The application rule App applies the function to the argument. It substitutes recursive mentions of the function in its body e by location ℓ, and substitutes the variable x by the argument v.

References. Rules RefS, BangS, UpdSL, and UpdSR evaluate their corresponding subexpressions. The rule Bang corresponds to dereferencing a mutable location ℓ and looks up the location ℓ in memory store μ and returns the stored value v. The rule records this in the leaf $[t]$ as the read action $\mathbf{R}\ell \Rightarrow \texttt{ref}\ v$. The rule Upd corresponds to a destructive update and updates the memory location ℓ to refer to value v. The rule records this in the leaf $[t]$ as the update action $\mathbf{U}\ell \Leftarrow \texttt{ref}\ v$.

$$\frac{\ell \notin \mathrm{dom}(\mu)}{\Delta\,;\,\mu\,;\,[t]\,;\,s \to \Delta\,;\,\mu[\ell \hookrightarrow s]\,;\,[t \oplus (\mathbf{A}\ell \Leftarrow s)]\,;\,\ell}\ \textsc{Alloc}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e_1 \to \Delta'\,;\,\mu'\,;\,T'\,;\,e_1{}'}{\Delta\,;\,\mu\,;\,T\,;\,(e_1\ e_2) \to \Delta'\,;\,\mu'\,;\,T'\,;\,(e_1'\ e_2)}\ \textsc{AppSL} \qquad \frac{\Delta\,;\,\mu\,;\,T\,;\,e_2 \to \Delta'\,;\,\mu'\,;\,T'\,;\,e_2{}'}{\Delta\,;\,\mu\,;\,T\,;\,(\ell_1\ e_2) \to \Delta'\,;\,\mu'\,;\,T'\,;\,(\ell_1\ e_2')}\ \textsc{AppSR}$$

$$\frac{\mu(\ell) = \mathtt{fun}\ f\ x\ \mathtt{is}\ e}{\Delta\,;\,\mu\,;\,[t]\,;\,(\ell\ v) \to \Delta\,;\,\mu\,;\,[t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fun}\ f\ x\ \mathtt{is}\ e)]\,;\,[\ell, v\,/\,f, x]e}\ \textsc{App}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e_1 \to \Delta\,;\,\mu'\,;\,T'\,;\,e_1{}'}{\Delta\,;\,\mu\,;\,T\,;\,(e_1 := e_2) \to \Delta\,;\,\mu'\,;\,T'\,;\,(e_1' := e_2)}\ \textsc{UpdSL} \qquad \frac{\Delta\,;\,\mu\,;\,T\,;\,e_2 \to \Delta\,;\,\mu'\,;\,T'\,;\,e_2{}'}{\Delta\,;\,\mu\,;\,T\,;\,(\ell_1 := e_2) \to \Delta\,;\,\mu'\,;\,T'\,;\,(\ell_1 := e_2')}\ \textsc{UpdSR}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e \to \Delta\,;\,\mu'\,;\,T'\,;\,e'}{\Delta\,;\,\mu\,;\,T\,;\,(\mathtt{ref}\ e) \to \Delta\,;\,\mu'\,;\,T'\,;\,(\mathtt{ref}\ e')}\ \textsc{RefS}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e \to \Delta\,;\,\mu'\,;\,T'\,;\,e'}{\Delta\,;\,\mu\,;\,T\,;\,(!\,e) \to \Delta\,;\,\mu'\,;\,T'\,;\,(!\,e')}\ \textsc{BangS} \qquad \frac{\mu(\ell) = \mathtt{ref}\ v}{\Delta\,;\,\mu\,;\,[t]\,;\,(!\,\ell) \to \Delta\,;\,\mu\,;\,[t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{ref}\ v)]\,;\,v}\ \textsc{Bang}$$

$$\frac{}{\Delta\,;\,\mu_0[\ell \hookrightarrow s]\,;\,[t]\,;\,(\ell := v) \to \Delta\,;\,\mu_0[\ell \hookrightarrow \mathtt{ref}\ v]\,;\,[t \oplus (\mathbf{U}\ell \Leftarrow \mathtt{ref}\ v)]\,;\,v}\ \textsc{Upd}$$

$$\frac{(a\ \mathit{fresh}) \qquad \Delta' = \Delta[a \blacktriangleright e]}{\Delta\,;\,\mu\,;\,[t]\,;\,\mathtt{fut}(e) \to \Delta'\,;\,\mu\,;\,t \oplus ([\bullet] \otimes_a [\bullet])\,;\,\mathtt{fcell}[a]}\ \textsc{FSpawn}$$

$$\frac{\Delta(a) \blacktriangleright e_1 \qquad \Delta\,;\,\mu\,;\,T_1\,;\,e_1 \to \Delta'\,;\,\mu'\,;\,T_1'\,;\,e_1' \qquad (\Delta' = \Delta_s'[a \blacktriangleright e_1])}{\Delta\,;\,\mu\,;\,t \oplus (T_1 \otimes_a T_2)\,;\,e_2 \to \Delta_s'[a \blacktriangleright e_1']\,;\,\mu'\,;\,t \oplus (T_1' \otimes_a T_2)\,;\,e_2}\ \textsc{FutS}$$

$$\frac{\Delta\,;\,\mu\,;\,T_2\,;\,e_2 \to \Delta'\,;\,\mu'\,;\,T_2'\,;\,e_2'}{\Delta\,;\,\mu\,;\,t \oplus (T_1 \otimes_a T_2)\,;\,e_2 \to \Delta'\,;\,\mu'\,;\,t \oplus (T_1 \otimes_a T_2')\,;\,e_2'}\ \textsc{ContS}$$

$$\frac{\Delta = \Delta_s[a \blacktriangleright v] \qquad \bowtie (a, T_1, T_2) = T \qquad \Delta' = \Delta_s[a \triangleright v]}{\Delta\,;\,\mu\,;\,t \oplus (T_1 \otimes_a T_2)\,;\,e_2 \to \Delta'\,;\,\mu\,;\,t \oplus T\,;\,e_2}\ \textsc{FJoin}$$

$$\frac{\mu(\ell) = \mathtt{fcell}[a] \qquad \Delta(a) \triangleright v}{\Delta\,;\,\mu\,;\,[t]\,;\,\mathtt{fpoll}(\ell) \to \Delta\,;\,\mu\,;\,[t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a])]\,;\,\mathtt{true}}\ \textsc{PollT}$$

$$\frac{\mu(\ell) = \mathtt{fcell}[a] \qquad \Delta(a) \blacktriangleright e}{\Delta\,;\,\mu\,;\,[t]\,;\,\mathtt{fpoll}(\ell) \to \Delta\,;\,\mu\,;\,[t \oplus (\mathbf{R}\ell \Rightarrow \mathtt{fcell}[a])]\,;\,\mathtt{false}}\ \textsc{PollF}$$

$$\frac{\Delta\,;\,\mu\,;\,T\,;\,e \to \Delta'\,;\,\mu'\,;\,T'\,;\,e'}{\Delta\,;\,\mu\,;\,T\,;\,\mathtt{get}(e) \to \Delta'\,;\,\mu'\,;\,T'\,;\,\mathtt{get}(e')}\ \textsc{GetS} \qquad \frac{\mu(\ell) = \mathtt{fcell}[a] \qquad \Delta(a) \triangleright v}{\Delta\,;\,\mu\,;\,[t]\,;\,\mathtt{get}(\ell) \to \Delta\,;\,\mu\,;\,[t \oplus (\mathbf{F}\ell \Rightarrow v)]\,;\,v}\ \textsc{Get}$$

$$\frac{n :\ \mathsf{int}}{\Delta\,;\,\mu\,;\,[t]\,;\,\mathtt{in\_nat}() \to \Delta\,;\,\mu\,;\,[t]\,;\,n}\ \textsc{Input} \qquad \frac{\mu(\ell) = n}{\Delta\,;\,\mu\,;\,[t]\,;\,\mathtt{out\_nat}(\ell) \to \Delta\,;\,\mu\,;\,[t \oplus \mathbf{R}\ell \Rightarrow n]\,;\,()}\ \textsc{Output}$$

Fig. 5. Dynamics of λ^U

Futures. The rule FSpawn spawns a future. It steps the expression $\texttt{fut}(e)$ to the future cell $\texttt{fcell}[a]$, where a is an unused/fresh name. Each future's evaluation is tracked in the ***future map***, denoted Δ, which stores all future names and their expressions. We write $\Delta[a \blacktriangleright e]$ to extend map Δ with future a and use $\Delta(a) \blacktriangleright e$ to denote that a is mapped to expression e. The rule FSpawn extends the future map with the new future and also adds two empty leaves to the computation tree. The resulting tree is of the form $t \oplus ([\bullet] \otimes_a [\bullet])$, where the symbol $\bullet$ denotes the empty trace. The rule composes the leaves with the operator $\otimes_a$, marking the spawn point of future a. The left and right leaf will store the subsequent actions of the future and the continuation respectively.

The rules FutS and ContS step the program state if the computation tree is of the form $t \oplus (T_1 \otimes_a T_2)$. The rule FutS looks up the future name a and expression e_1 in future map Δ, and steps the expression e_1 with the left subtree T_1. The rule has a premise, the condition $\Delta' = \Delta'_s[a \blacktriangleright e_1]$ which guarantees that stepping e_1 does not change the future map for future a, i.e., $\Delta(a) = \Delta'(a)$. The rule has the premise because this rule is responsible for tracking the evaluation of future a. For the resulting state, the rule FutS maps the future a to expression e'_1. The rule ContS steps the continuation e_2 with the right subtree T_2. These rules can be interleaved non-deterministically to model parallel evaluation.

Once a future is fully evaluated to a value, the rule FJoin joins it with its continuation. The rule performs the join transformation on the computation tree, as described earlier in Section 2.4, and also updates the future map to mark that the future has joined. In the future map, we use an unshaded triangle to denote joined futures. The rule changes the map from $\Delta_s[a \blacktriangleright v]$ to $\Delta_s[a \triangleright v]$.

Polling and synchronization. The rules Poll, PollT, and PollF describe the semantics of polling. The rule Poll steps its argument subexpression. If the future being polled has joined, then the rule PollT steps $\texttt{fpoll}(\ell)$ to $\texttt{true}$; otherwise, the rule PollF steps $\texttt{fpoll}(\ell)$ to $\texttt{false}$. Since both the rules look up location ℓ in the memory store μ, they insert the read action $\mathbf{R}\ell \Rightarrow \texttt{fcell}[a]$ to the computation tree. Note that polling is a non-blocking primitive, as the expression $\texttt{fpoll}$ always steps immediately.

Unlike the expression $\texttt{fpoll}(e)$, the expression $\texttt{get}(e)$ blocks until the future completes and then returns its the value. The rule GetS steps the argument expression e to a location ℓ. Then, once the future referred by location ℓ has joined, the rule Get retrieves the value of from the future map and returns it. The rule records this synchronization in the computation tree with the action $\mathbf{F}\ell \Rightarrow v$, where v is the return value of the future. Notice that the rule Get has the condition $\Delta_s[a \triangleright v]$ in the premise, asserting that it blocks until the future has joined.

Input/Output. The rule Input steps the expression $\texttt{in\_nat}()$ to a non-deterministic natural number n. The natural number n will then be allocated in the memory store by the rule Alloc. The rule's non-determinism models the effect of the input on evaluation and the allocation captures the effect of the input on memory and disentanglement. The rule output takes a location ℓ, which stores a natural number n, and steps the location to a unit value. The rule extends the computation tree with the read action $\mathbf{R}\ell \Rightarrow n$. This step models an output to an environment and the read action captures the effect of the output on disentanglement. For brevity, we do not model I/O on other types but the language can be extended to support them.

3 DISENTANGLEMENT IN PURE AND PURELY INTERACTIVE PROGRAMS

In this section, we focus on a subset of our language which includes futures and I/O. We call the subset language λ^P, where P indicates that the language is "pure" or "purely interactive", i.e., the language supports futures and I/O, but does not allow mutable references. Because the language disallows mutation, its programs are automatically devoid of races, making it an excellent medium for expressing parallel programs. The language supports futures which can express sophisticated algorithms and techniques such as pipelining. When coupled with the ability to interact with the

```
type α tree =
  Empty
| Node of α * (α tree) fut * (α tree) fut

merge :: α tree → α tree → α tree
fun  merge t1 t2 =
  case (t1, t2) of
    (Empty, _) → t2
  | (_, Empty) → t1
  | (Node (v, l, g), _) →
      let s = split v t2
          ll = fut merge (get l) (#1 s)
          gg = fut merge (get g) (#2 s)
      in
        Node (v, ll, gg)
```

```
split :: α → α tree
         → ((α tree) fut * (α tree) fut)
fun split k t =
  case t of
    Empty → (fut Empty, fut Empty)
  | Node (v, l, g) →
      if k < v then
        let s = fut split k (get l)
        in
          (fut #1 s,
           fut Node (v, #2 s, g))
      else
        let s = fut split k (get g)
        in
          (fut Node (v, l, #1 s),
           fut #2 s)
```

Fig. 6. Pipelined merge with futures. We define the function `#1 x = get (fst x)` and `#2 x = get (snd x)`, where `fst` and `snd` project out the first and the second component of a pair.

environment, futures can elegantly express asynchronous and interactive applications such as a web server. In this section, we show that all such algorithms and applications satisfy disentanglement by showing that all programs of language λ^P are disentangled.

Before delving into the technical details, we begin with a classic example of pipelining with futures. As another example, we present a web server written in this language in Section 6.1.

3.1 Pipelining with Futures

Pipelining is a fundamental technique in the design of parallel algorithms that can meaningfully reduce the parallel depth (span). For example, pipelining was used by Paul et al. to improve parallel operations on balanced trees [Paul et al. 1983] and by Cole to give a $O(\lg n)$ span parallel mergesort algorithm [Cole 1988]. Implementing pipelined algorithms, however, is quite challenging, because the programmer has to carefully manage the rather complex, producer-consumer-like data dependencies between parallel computations. Blelloch and Reid-Miller [Blelloch and Reid-Miller 1999] showed that pipelined algorithms can be expressed at quite a high level by using functional programming extended with futures.

Figure 6 shows the code for a pipelined tree merge from Blelloch and Reid-Miller [Blelloch and Reid-Miller 1999], adapted to our language [1]. The `tree` datatype is a binary search tree whose branches are of future type. The `merge` function returns the non-empty tree if one of the trees is empty. In the case where both trees are non-empty, the function splits the second tree by using the key at the root of the first tree and recursively merges the two "halves" from the two trees. These recursive merges run inside futures, allowing them to proceed in parallel. Because the function `split` also returns the recursive portion of its result inside a future, the recursive calls to `merge` can run in a pipelined fashion with the `split`. This is possible because each node of the tree is wrapped inside future and is demanded by the `get` expression as needed.

Given two balanced trees of depth $O(\lg n)$, the parallel `merge` runs in $O(\lg n)$ span or parallel time. With fork-join parallelism, however, the best parallel merge runs in $O(\lg^2(n))$ span.

3.2 Disentanglement for Futures and Interaction

We show here that evaluation of a program in language λ^P always satisfies disentanglement. To evaluate a program, we consider the stepping relation ($\rightarrow_P$), which contains a subset of the rules we

[1] The example in Blelloch and Reid-Miller [Blelloch and Reid-Miller 1999] uses a non-strict semantics for forcing futures whereas our futures are strict.

$\boxed{K\,;\,A \vdash_{\Delta,\mu} T\,;\,e\ \mathit{defut}}$

$$\frac{\mathsf{Loc}(e) \subseteq A \qquad \mathsf{Fut}(e,\mu) \subseteq K \qquad \forall \ell \in A.\ \mathsf{Loc}(\mu(\ell)) \subseteq A}{K\,;\,A \vdash_{\Delta,\mu} [\bullet]\,;\,e\ \mathit{defut}}\ (5.1) \qquad \frac{A \vdash t\ \mathit{de} \qquad K\,;\,A \cup \mathsf{A}(t) \vdash_{\Delta,\mu} [\bullet]\,;\,e\ \mathit{defut}}{K\,;\,A \vdash_{\Delta,\mu} [t]\,;\,e\ \mathit{defut}}\ (5.2)$$

$$\frac{A \vdash t\ \mathit{de} \quad \Delta(a) \rhd v \quad (a \notin K) \qquad \begin{array}{c} K\,;\,A \cup \mathsf{A}(t) \vdash_{\Delta,\mu} [\bullet]\,;\,v\ \mathit{defut} \\ K \cup \{a\}\,;\,A \cup \mathsf{A}(t) \vdash_{\Delta,\mu} T\,;\,e\ \mathit{defut} \end{array}}{K\,;\,A \vdash_{\Delta,\mu} t \oplus_a T\,;\,e\ \mathit{defut}}\ (5.3)$$

$$\frac{A \vdash t\ \mathit{de} \quad \Delta(a) \blacktriangleright e_1 \quad (a \notin K) \quad \begin{array}{c} K\,;\,A \cup \mathsf{A}(t) \vdash_{\Delta,\mu} T_1\,;\,e_1\ \mathit{defut} \\ K \cup \{a\}\,;\,A \cup \mathsf{A}(t) \vdash_{\Delta,\mu} T_2\,;\,e_2\ \mathit{defut} \end{array}}{K\,;\,A \vdash_{\Delta,\mu} t \oplus (T_1 \otimes_a T_2)\,;\,e_2\ \mathit{defut}}\ (5.4)$$

Fig. 7. Rules for the invariant *defut*

define in our full language (Section 2.5). The subscript *"P"* denotes that the relation only considers rules for the constructs in the language λ^P, which contains futures and I/O but no mutable references. To prove the result formally, we consider the computation tree produced by the language semantics and show that it is disentangled at each step, i.e.,

Theorem 3.1 (Pure and purely interactive futures are disentangled). *For any* $\emptyset\,;\,\emptyset\,;\,[\bullet]\,;\,e_0 \longmapsto_P^* \Delta\,;\,\mu\,;\,T\,;\,e$, *where* $\mathsf{Loc}(e_0) = \emptyset$, *we have* $\emptyset \vdash T\ \mathit{de}$.

The theorem states that starting from the initial state $(\emptyset\,;\,\emptyset\,;\,[\bullet]\,;\,e_0)$, if the program takes an arbitrary number of steps, then the tree of the resulting state satisfies disentanglement. Because the theorem leaves the number of steps to be arbitrary, it implies that the computation tree of each state satisfies disentanglement, proving that the evaluation satisfies disentanglement.

To prove the theorem, we observe that in a pure program, the only way a thread can share an allocation with another thread is by synchronizing with it as a future. Such a synchronization returns allocations, potentially including handles to other futures because futures are themselves memory allocations (they are first class). Thus, to establish disentanglement, we must prove that a thread never gets a handle to a future that is spawned concurrently. We formalize this by defining two properties, namely ***defut*** and ***ok*** and prove them as invariants of the computation. For a program state $(\Delta\,;\,\mu\,;\,T\,;\,e)$, we define the properties with an inductive process on the tree T, using the judgements $K\,;\,A \vdash_{\Delta,\mu} T\,;\,e\ \mathit{defut}$ and $A\,;\,\Delta' \vdash_{\mu'} K\ \mathit{ok}$ respectively. The set K contains future names and the set A contains memory locations. For the full tree, the sets K and A are empty, but for internal subtrees, the sets contain the futures and locations that actions of a subtree may mention without violating disentanglement. We discuss the properties in detail and prove them by induction.

Property defut. The *defut* property implies disentanglement and in addition imposes restrictions on the memory locations mentioned by the expressions of various program threads. Given a subtree T and an expression e, the judgement $\mathit{defut}\ (K\,;\,A \vdash_{\Delta,\mu} T\,;\,e\ \mathit{defut})$ enforces that (i) the tree T satisfies disentanglement w.r.t. the set A, i.e., $A \vdash T\ \mathit{de}$, (ii) the expression e only mentions futures present in set K and locations present in set A, in addition to futures spawned and locations allocated within the subtree T, and (iii) all subtrees of tree T satisfy the judgement *defut*.

Figure 7 show the rules for the judgement *defut*. The Rule (5.1) applies to an empty leaf of the form $[\bullet]$. The rule checks that all locations mentioned by expression e are in the set A. The rule uses the function $\mathsf{Fut}(e,\mu)$, which returns all futures mentioned by expression e, and asserts that

they are in the set K [2]. Crucially, the rule also enforces that the set A is closed under the memory pointer relation, i.e., a location in A only points to locations which are also in A (highlighted in purple). Since all locations mentioned by the expression e are in set A, the purple premise ensures that that accessing a location mentioned in expression e does not lead to a location outside A. The Rule (5.2) considers the case when the tree is a leaf, i.e., of the form $[t]$. It checks that trace t is disentangled and defers to the Rule (5.1) after extending the set A with the allocations in trace t.

The Rule (5.4) applies to a tree of the form $t \oplus (T_1 \otimes_a T_2)$. Recall that this tree represents the spawning of future a after actions in trace t, with tree T_1 recording the actions of the future, and tree T_2 recording the actions of the continuation. The rule checks the tree T_2 with expression e_2 after extending the set A to $A \cup \mathsf{A}(t)$ and set K to $K \cup \{a\}$. These extensions represent that the tree T_2 and expression e_2 can mention locations allocated by the trace t and also access the future a. For the tree T_1, the rule uses future map Δ to retrieve the future's expression e_1 and checks $K\,;A \cup \mathsf{A}(t) \vdash_{\Delta,\mu} T_1\,;e_1$ *defut*. The rule extends the set A with the allocations in tree t, but unlike for tree T_2, the rule does not extend the set K with future a. By excluding future a from set K, the rule prohibits the future from mentioning and accessing itself.

The Rule (5.3) shows the conditions for future a after it has joined with its continuation e. The tree in this case is of the form $t \oplus_a T$, where the operator $\oplus_a$ marks the join point of future a. Because the future has joined, its is mapped to a value v in the future map and the rule checks the value similar to Rule (5.1).

Property ok. The judgement *ok* ($A\,;\Delta \vdash_\mu K$ *ok*) guarantees that the value of every terminated future in set K only refers to futures and locations within sets K and A. With this judgement, we can show that if a thread performs a synchronization action on a future to retrieve a value, then the value only contains locations and futures within sets K and A. We can define it formally as follows.

$$\frac{K \subseteq \mathrm{dom}(\Delta) \qquad \forall a \in K.\ \Delta(a) \triangleright v \Rightarrow \mathsf{Loc}(v) \subseteq A \wedge \mathsf{Fut}(v,\mu) \subseteq K}{A\,;\Delta \vdash_\mu K\ ok}$$

As a sanity check, the property *ok* ensures that all futures in K are in the future map Δ. The property then checks the values of terminated futures; recall that futures that have terminated are mapped with an unshaded triangle in the future map Δ. For each terminated future, the property checks that the return value v of the future only refers to locations and futures in sets A and K respectively. The property uses the function $\mathsf{Loc}(v)$ that returns all the locations mentioned in value v and asserts that all such locations are in the set A. The property uses the function $\mathsf{Fut}(v,\mu)$, which returns all futures referred by value v and asserts that they are in the set K. We leave the definitions of these functions to the Appendix B.

We show the following lemma which is the inductive step for the main theorem (Theorem 3.1).

Lemma 3.2. *For any sets A and K, and step $\Delta\,;\mu\,;T\,;e \to_P \Delta'\,;\mu'\,;T'\,;e'$, if $A\,;\Delta \vdash_\mu K$ ok and $K\,;A \vdash_{\Delta,\mu} T\,;e$ defut then $A\,;\Delta' \vdash_{\mu'} K$ ok and $K\,;A \vdash_{\Delta',\mu'} T'\,;e'$ defut.*

The lemma says: if a state satisfying the properties *defut* and *ok* takes a step, then the resulting state also satisfies both the properties. We prove the lemma itself by induction on the stepping relation: $\Delta\,;\mu\,;T\,;e \to_P \Delta'\,;\mu'\,;T'\,;e'$.

Proof. We cover the step Get of the operational semantics here and leave others to the Appendix.
Case Get.

[2]We define the function $\mathsf{Fut}(e,\mu)$ formally in the Appendix B

$$\frac{\mu(\ell) = \texttt{fcell}[a] \qquad \Delta(a) \rhd v}{\Delta\,;\mu\,;[t]\,;\texttt{get}(\ell) \longrightarrow \Delta\,;\mu\,;[t \oplus (\mathbf{F}\ell \Rightarrow v)]\,;v}\ \textsc{Get}$$

The rule Get synchronizes with a terminated future and returns its value. The initial state $(\Delta;\mu;T;e)$ for this rule is $(\Delta\,;\mu\,;[t]\,;\texttt{get}(\ell))$. The state after the step $(\Delta'\,;\mu'\,;T'\,;e')$ is $(\Delta\,;\mu\,;[t \oplus (\mathbf{F}\ell \Rightarrow v)])$. The step requires that the location ℓ points to future a ($\mu(\ell) = \texttt{fcell}[a]$) and that the future a has joined ($\Delta(a) \rhd v$). Thus, we have $e = \texttt{get}(\ell)$, $T = [t]$, $\mu(\ell) = \texttt{fcell}[a]$, $\Delta(a) \rhd v$, $\Delta' = \Delta$, $\mu' = \mu$, $T' = [t \oplus (\mathbf{F}\ell \Rightarrow v)]$, and $e' = v$.

To prove the lemma, we need to prove the *defut* and *ok* properties for the state after the step. We can also assume that the properties hold for the state before the step, i.e., we assume $K\,;A \vdash_{\Delta,\mu} [t]\,;\texttt{get}(\ell)$ *defut* and $A\,;\Delta \vdash_{\mu} K$ *ok*. The *ok* judgement $A\,;\Delta' \vdash_{\mu'} K$ *ok* follows directly from $A\,;\Delta \vdash_{\mu} K$ *ok* because $\mu' = \mu$ and $\Delta' = \Delta$.

To prove property *defut*, we need to show $K\,;A \vdash_{\Delta',\mu'} T'\,;e'$ *defut* which is equivalent $K\,;A \vdash_{\Delta,\mu} [t \oplus (\mathbf{F}\ell \Rightarrow v)]\,;v$ *defut*. We can show this using the (Rule (5.2)) and prove its premise as follows:

- $A \vdash t \oplus (\mathbf{F}\ell \Rightarrow v)$ *de*. To prove the judgement $A \vdash t \oplus (\mathbf{F}\ell \Rightarrow v)$ *de*, it suffices to show $A \vdash t$ *de* and $A \cup \mathsf{A}(t) \vdash (\mathbf{F}\ell \Rightarrow v)$ *de*. We can show the first part, $A \vdash t$ *de*, by inversion on $K\,;A \vdash_{\Delta,\mu} [t]\,;\texttt{get}(\ell)$ *defut*. To prove $A \cup \mathsf{A}(t) \vdash (\mathbf{F}\ell \Rightarrow v)$ *de*, we need to show $\mathsf{Loc}(v) \subseteq A \cup \mathsf{A}(t)$ and $\ell \in A \cup \mathsf{A}(t)$. We can show the first part, $\mathsf{Loc}(v) \subseteq A \cup \mathsf{A}(t)$, by inversion on $A\,;\Delta \vdash_{\mu} K$ *ok* and $\Delta(a) \rhd v$. The second part, $\ell \in A \cup \mathsf{A}(t)$, follows from inversion on $K\,;A \vdash_{\Delta,\mu} [\bullet]\,;\texttt{get}(\ell)$ *defut*, which gives us $\mathsf{Loc}(\texttt{get}(\ell)) \subseteq A \cup \mathsf{A}(t)$, which means that $\ell \in A \cup \mathsf{A}(t)$.
- $K\,;A \cup \mathsf{A}(t \oplus (\mathbf{F}\ell \Rightarrow v)) \vdash_{\Delta,\mu} [\bullet]\,;v$ *defut*. This follows from $A \cup \mathsf{A}(t \oplus (\mathbf{F}\ell \Rightarrow v)) = A \cup \mathsf{A}(t)$ and $K\,;A \cup \mathsf{A}(t) \vdash_{\Delta,\mu} [\bullet]\,;v$ *defut* which we shows as follows:
  - $\mathsf{Loc}(v) \subseteq A \cup \mathsf{A}(t)$, by inversion on $A\,;\Delta \vdash_{\mu} K$ *ok*
  - $\forall \ell \in (A \cup \mathsf{A}(t))$. $\mathsf{Loc}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, by inversion on $K\,;A \vdash_{\Delta,\mu} [\bullet]\,;\texttt{get}(\ell)$ *defut*
  - $\mathsf{Fut}(v,\mu) \subseteq K$, by inversion on $K\,;A \vdash_{\Delta,\mu} [\bullet]\,;\texttt{get}(\ell)$ *defut*

□

4 STATEFUL PROGRAMS WITH DISENTANGLEMENT

Because it allows for in-place updates, mutable state improves the efficiency and scalability of many parallel algorithms. When used in conjunction with futures, state further allows expressing parallelism hidden behind complex dependencies effortlessly, for example, by allowing futures to be stored inside stateful data structures such as references and arrays. In this section, we show that determinacy race free programs with futures and state satisfy disentanglement. This result broadens disentanglement to encompass a huge variety of programs because determinacy race freedom is a generally accepted safety property for parallel programs.

Some programs, however, use races to improve performance and control them carefully so as to not harm correctness. With the introduction of atomic hardware instructions such as "compare-and-swap", one can safely perform atomic updates, which has further increased the prevalence of this practice. For example, many graph algorithms today use "compare-and-swap" instructions to mark the vertices of a graph that they visit to prevent multiple, possibly concurrent visits, to the same vertex. To account for these programs, we introduce the cas ("compare-and-swap") primitive to our language. and establish that weakly race-free programs, i.e., programs which may exhibit races on "unboxed values" such as integers, are still disentangled.

We start this section with an example that uses futures and state to express parallelism and asynchrony in a PDF viewer. The application is devoid of races and is therefore disentangled.

```
type pdf = {raw_data : bytes array,
  num_pages : int,
  page_offsets : int → int}
val render : pdf * int → page
val display : page → unit
val getClick : unit → int

fun viewer (p : pdf) =
let
  page_arr = tabulate (#num_pages p) NONE
  fun loop () =
    let pnum = getClick () in
      case page_arr[pnum] of
        NONE =>
          pg = render (p, pnum);
          display (pg);
          page_arr[pnum] ← SOME (fut (pg));
          fill_page_arr (pnum - δ, pnum + δ) page_arr p
        SOME f =>
          display (get f)
in
  loop ()
```

```
fun fill_page_arr (l, r) page_arr p =
  let
    fun fill i =
      case page_arr[i] of
        NONE => page_arr[i] ←
          fut (render (p, i))
      | SOME _ => ()
  in
    foreach (l, r) (fn i => fill (i))
```

Fig. 8. PDF viewer with disentanglement

4.1 PDF Viewer with Disentanglement

We consider a PDF viewer that accepts a page number from the user and displays the corresponding page of the PDF on the screen. Before displaying a requested page, the viewer must first render the page, i.e., it must interpret the bytes in the PDF and generate a visual representation, like a pixel array, which it can then display. Our viewer renders pages in an optimized manner, as it not only renders the pages as requested by the user, but also anticipates the user's navigation actions and renders adjacent pages in the background. It spawns futures to perform this proactive rendering and stores the futures in an array indexed by the appropriate page numbers. In this way, futures and array implement a proactive memoization/caching mechanism, allowing the viewer to display rendered pages efficiently and without delay.

Figure 8 shows the code for the viewer. The function `viewer` allocates the "page array", a future option array indexed by the page numbers of the PDF. Initially all elements of the array are `NONE`. The function then runs the `loop` function, which repeatedly awaits for the user to request a page number by calling the function `getClick`. To complete the user request, the function `loop` looks up the page number in the array and if it finds `NONE`, the loop prepares the page by calling function `render`, displays the page to the user, and writes it to the page array at this page number. After completing the user request, the function `loop` proactively renders δ number of pages near the current page by calling the function `fill_page_arr`, which spawns futures to render the given range of pages and writes the corresponding futures in the page array. Note that function `fill_page_arr` takes constant time because it returns immediately after spawning futures that render pages in the background. This allows the loop to be ready promptly for the next user request.

The code uses the stateful array to track futures and also to memoize the already prepared pages. Note that the array is accessed exclusively by a single thread, the thread that runs the functions `viewer`, `loop`, and `fill_page_arr`. The futures spawned by the thread never read or write to the array. Thus, the code does not exhibit determinacy races because all the writes in the code are visible only to the writer itself. The next subsection proves that determinacy race freedom implies disentanglement, thereby establishing that the PDF viewer is also disentangled.

$$\frac{}{F \vdash \bullet \; drf} \quad \frac{\ell \notin F}{F \vdash (\mathbf{A}\ell \Leftarrow s) \; drf} \quad \frac{\ell \notin F}{F \vdash (\mathbf{F}\ell \Rightarrow v) \; drf} \quad \frac{\ell \notin F}{F \vdash (\mathbf{U}\ell \Leftarrow s) \; drf} \quad \frac{\ell \notin F}{F \vdash (\mathbf{R}\ell \Rightarrow s) \; drf}$$

$$\frac{F \vdash t_1 \; drf \qquad F \vdash t_2 \; drf}{F \vdash t_1 \oplus t_2 \; drf}$$

$$\frac{F \vdash t \; drf}{F \vdash [t] \; drf} \quad \frac{F \vdash t \; drf \qquad F \vdash T \; drf}{F \vdash t \oplus_a T \; drf} \quad \frac{F \vdash t \; drf \qquad F \cup \mathsf{AW}(T_2) \vdash T_1 \; drf \qquad F \cup \mathsf{AW}(T_1) \vdash T_2 \; drf}{F \vdash t \oplus (T_1 \otimes_a T_2) \; drf}$$

Fig. 9. The figure defines the judgement $F \vdash T \; drf$, where F is a set of locations that actions of T must not mention. The function AW takes a tree and returns the set of locations allocated/updated by it.

4.2 Determinacy Race Freedom implies Disentanglement

Determinacy Races. A ***determinacy race*** occurs when two concurrent threads access the same memory location, and one of those accesses modifies the location [Netzer and Miller 1992]. ***Determinacy race freedom*** is the program property that guarantees that every execution of the program is free of determinacy races. We say that a computation with no determinacy races is determinacy race free. For brevity, we write race and race freedom to mean determinacy race and determinacy race freedom.

We define race-freedom formally using the computation tree. The computation tree organizes the memory actions of program threads based on their control flow dependencies. It orders sequential memory actions in an ancestor-descendant relationship and keeps concurrent memory actions unrelated. The memory actions in the tree include modifying actions such as allocation ($\mathbf{A}\ell \Leftarrow s$) and update ($\mathbf{U}\ell \Leftarrow s$) and non-modifying actions such as read ($\mathbf{R}\ell \Rightarrow s$) and sync ($\mathbf{F}\ell \Rightarrow v$). A computation tree satisfies race freedom if no modifying action on a location is concurrent to another action, modifying or otherwise, on that location.

Figure 9 defines race freedom for a computation tree with an inductive process using the judgement $F \vdash T \; drf$. The judgement's context F represents a set of "forbidden locations" that are modified by threads concurrent to those represented in tree T. The judgement $F \vdash T \; drf$ ensures that no action of tree T operates on a location in the set F.

Let's look at the rules for the judgement. The rule for the tree $t \oplus (T_1 \otimes_a T_2)$ shows how the forbidden set prohibits races between the concurrent trees T_1 and T_2. When checking tree T_1, the rule extends the forbidden set with locations that are modified by tree T_2. Specifically, let $\mathsf{AW}(T)$ represents the set of locations modified by tree T. Then the rule checks tree T_1 with the forbidden set $F \cup \mathsf{AW}(T_2)$, which ensures that actions of tree T_1 are forbidden from locations modified by tree T_2. The rule checks tree T_2 similarly.

The rule for the read action $\mathbf{R}\ell \Rightarrow s$ checks that location ℓ is not in the set F, checking that no concurrent action modified location ℓ. The rules for other actions also check the respective locations. These rules uncover an interesting perspective on the distinction between race freedom and disentanglement. While race freedom only restricts where an action occurs, disentanglement goes a step further and also restricts what an action can store or retrieve. For example, the rule for checking disentanglement of a read action $\mathbf{R}\ell \Rightarrow s$ checks both the location ℓ and the locations in storable s (see Figure 3), whereas the rule for race freedom only checks the location ℓ, leaving the contents of storable s unrestricted. From this perspective, it is perhaps surprising that disentanglement applies to a broader class of programs (because it is implied by race freedom).

DRF implies DE. We prove taking an arbitrary number of steps from an initial state, if the computation tree satisfies the *drf* property at each step, then it satisfies the *de* property after the final step. Thus, if the *drf* property holds for every step, the *de* property does as well.

$$\boxed{K\,;A\,;F \vdash_\mu \Delta\,;T\,;e\ \mathit{drfde}}$$

$$\frac{\mathsf{Loc}(e) \subseteq A \quad \mathsf{Fut}(e,\mu) \subseteq K \qquad \forall \ell \in A \setminus F.\ \mathsf{Loc}(\mu(\ell)) \subseteq A \qquad \forall \ell \in A.\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K}{K\,;A\,;F \vdash_\mu \Delta\,;[\bullet]\,;e\ \mathit{drfde}}$$

$$\frac{F \vdash t\ \mathit{drf} \qquad A \vdash t\ \mathit{de} \qquad K\,;A \cup \mathsf{A}(t)\,;F \vdash_\mu \Delta\,;[\bullet]\,;e\ \mathit{drfde}}{K\,;A\,;F \vdash_\mu \Delta\,;[t]\,;e\ \mathit{drfde}}$$

$$\frac{F \vdash t\ \mathit{drf} \qquad A \vdash t\ \mathit{de} \qquad \Delta(a) \rhd v \qquad \begin{array}{c} K\,;A \cup \mathsf{A}(t)\,;F \vdash_\mu \Delta\,;[\bullet]\,;v\ \mathit{drfde} \\ K \cup \{a\}\,;A \cup \mathsf{A}(t)\,;F \vdash_\mu \Delta\,;T\,;e\ \mathit{drfde} \end{array}}{K\,;A\,;F \vdash_\mu \Delta\,;t \oplus_a T\,;e\ \mathit{drfde}}$$

$$\frac{F \vdash t\ \mathit{drf} \qquad A \vdash t\ \mathit{de} \qquad \Delta(a) \blacktriangleright e_1 \qquad \begin{array}{c} K\,;A \cup \mathsf{A}(t)\,;F \cup \mathsf{AW}(T_2) \vdash_\mu \Delta\,;T_1\,;e_1\ \mathit{drfde} \\ K \cup \{a\}\,;A \cup \mathsf{A}(t)\,;F \cup \mathsf{AW}(T_1) \vdash_\mu \Delta\,;T_2\,;e_2\ \mathit{drfde} \end{array}}{K\,;A\,;F \vdash_\mu \Delta\,;t \oplus (T_1 \otimes_a T_2)\,;e_2\ \mathit{drfde}}$$

Fig. 10. Strengthening of disentanglement and race freedom with invariants on futures and memory

Theorem 4.1 (DRF $\Rightarrow$ DE). *For any* $\emptyset\,;\emptyset\,;[\bullet]\,;e_0 \to^n \Delta\,;\mu\,;T_n\,;e_n$ *where* $\mathsf{Loc}(e_0) = \emptyset$*, if every intermediate tree* T_i *in* $\{T_1 \ldots T_n\}$ *satisfies* $\emptyset \vdash T_i$ *drf, then* $\emptyset \vdash T_n$ *de.*

To prove the theorem, we observe that determinacy race freedom forbids concurrent threads to read each other's memory updates, and thus prevents them from communicating their allocations among each other. This, in turn, prevents threads from sharing spawned futures and violating disentanglement. We prove the theorem by induction on the number of steps. The inductive hypothesis enforces two properties on the program state: the property *drfde* which implies both disentanglement and race freedom and the property *ok* which restricts the structure of futures. For a state $\Delta\,;\mu\,;T\,;e$, we write the *drfde* property as the judgement $K\,;A\,;F \vdash_\mu \Delta\,;T\,;e$ *drfde* and the *ok* property as the judgement $A\,;\Delta \vdash_\mu K$ *ok*. The sets A, K, and F are empty for the full state, but for sub states of various threads in the program, the sets encode memory information relevant to disentanglement and race freedom. The set A is a set of locations and set K is a set of futures, both of which a thread can access without violating disentanglement. The set F is the set of forbidden locations that a thread should not access or else the thread violates race freedom.

$$\frac{K \subseteq \mathrm{dom}(\Delta) \qquad \forall a \in K.\ \Delta(a) \rhd v \Rightarrow \mathsf{Loc}(v) \subseteq A \wedge \mathsf{Fut}(v,\mu) \subseteq K}{A\,;\Delta \vdash_\mu K\ \mathit{ok}}$$

The *ok* property is identical to the one used in the Section 3 and roughly guarantees that any future in set K which has terminated can only return locations in set A and futures in set K.

Figure 10 shows the rules for the judgement $K\,;A\,;F \vdash_\mu \Delta\,;T\,;e$ *drfde*. The rules encode both disentanglement and race freedom by creating the sets A and F in the same way as the definitions of judgement *de* and *drf* respectively. The rules additionally maintain the set K of futures, which contains futures whose spawn/join points are ancestors of tree T; recall that operator $\otimes_a$ denotes the spawn point of future a and operator $\oplus_a$ denotes the join point. Whenever a rule sees a spawn/join point, it adds the appropriate future to the set K in its premise.

The rule for the empty tree ($K\,;A\,;F \vdash_\mu \Delta\,;[\bullet]\,;e$ *drfde*) contains additional constraints imposed by the property *drfde* (see the first rule in Figure 10). The rule ensures that 1) expression e only mentions locations in set A, 2) expression e only mentions futures in set F, 3) all locations that are in set A but not in set F, i.e. in set difference $A \setminus F$, point to locations in set A, and 4) if a location in set A refers to a future, then the future is in set K. The four properties guarantee that no matter what the expression e does in the next step, it will not access a location outside set A or access a

future outside set K, assuming the step does not access a location in set F (which is implied by the race freedom assumption). These constraints guarantee disentanglement. We prove in the following lemma that with race freedom, the properties *drfde* and *ok* are invariants of the computation.

Lemma 4.2. *For any* $\Delta\,;\mu\,;T\,;e \to \Delta'\,;\mu'\,;T'\,;e'$ *if* $K\,;A\,;F \vdash_\mu \Delta\,;T\,;e$ *drfde,* $A\,;\Delta \vdash_\mu K$ *ok, and* $K \vdash T'$ *drf, then* $K\,;A\,;F \vdash_{\mu'} \Delta'\,;T'\,;e'$ *drfde and* $A\,;\Delta' \vdash_{\mu'} K$ *ok.*

The lemma states that if the *drfde* and the *ok* properties hold for a sub state $\Delta\,;\mu\,;T\,;e$ and if state takes a step to state $\Delta'\,;\mu'\,;T'\,;e'$, then the *drfde* and the *ok* properties holds for the resulting state assuming its tree T' satisfies race freedom, i.e., $F \vdash T'$ *drf*.

We can use the lemma to show that the full program state always satisfies disentanglement. For the full state, the sets K, A, and F are empty. From the lemma, we have that if a state that satisfies *drfde* takes a step such that the resulting state satisfies the *drf* property then the resulting state also satisfies the *drfde* property. Because the initial state satisfies the property *drfde* and we assume that all states satisfy the *drf* property, we have that all states satisfy the *drfde* property. Since the *drfde* property implies the *de* property, we have that race freedom implies disentanglement.

We prove the lemma by induction on the stepping relation $\Delta\,;\mu\,;T\,;e \to \Delta'\,;\mu'\,;T'\,;e'$. We cover the case for reading a reference (Rule Bang of Figure 5) here, and leave others to the Appendix.

Proof. Case Bang. This step reads a reference's value from memory. The expression e is of the form $!\,\ell$ and it steps to location ℓ', such that $\mu(\ell) = \texttt{ref}\ \ell'$. The tree T is a leaf of the form $[t]$; this step extends it with the read action and creates the tree $T' = [t \oplus (\mathbf{R}\ell \Rightarrow \texttt{ref}\ \ell')]$. The future map and the memory remain unchanged, i.e., $\Delta' = \Delta$ and $\mu' = \mu$. Thus, for the resulting state, the condition $ok{-}A\,;\Delta' \vdash_{\mu'} K$ *ok*, follows directly from $A\,;\Delta \vdash_\mu K$ *ok*.

Because the step is assumed to be race-free, we know $F \vdash T'$ *drf*. Recall that the *drf* property makes sure that no action tree T' mentions a location in F. Because the read action $\mathbf{R}\ell \Rightarrow \texttt{ref}\ \ell'$ is in tree T', we know $\ell \notin F$. By applying inversion on $K\,;A\,;F \vdash_\mu \Delta\,;[t]\,;!\,\ell$ *drfde*, we get: $F \vdash t$ *drf*, $A \vdash t$ *de*, and $K\,;A \cup \mathsf{A}(t)\,;F \vdash_\mu \Delta\,;[\bullet]\,;!\,\ell$ *drfde*. By applying inversion on $K\,;A \cup \mathsf{A}(t)\,;F \vdash_\mu \Delta\,;[\bullet]\,;!\,\ell$ *drfde*, we have: $\ell \in A \cup \mathsf{A}(t)$, $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathsf{Loc}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, $\mathsf{Fut}(e, \mu) \subseteq K$, and $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow a \in K$.

The judgement $K\,;A\,;F \vdash_\mu \Delta\,;T'\,;e'$ *drfde* follows from applying Rule (8.2) with the following:

- $F \vdash T'$ *drf*, assumed.
- $A \vdash T'$ *de*. Note that T' is a leaf of the form $[t \oplus (\mathbf{R}\ell \Rightarrow \texttt{ref}\ \ell')]$. We know trace t is disentangled from $A \vdash T$ *de* and $T = [t]$. To prove, that the read action $(\mathbf{R}\ell \Rightarrow \texttt{ref}\ \ell')$ is disentangled, we need to show $\ell \in A$ and $\ell' \in A$ (both established above).
- $K\,;A \cup \mathsf{A}(t \oplus (\mathbf{R}\ell \Rightarrow \texttt{ref}\ \ell'))\,;F \vdash_\mu \Delta\,;[\bullet]\,;\ell'$ *drfde*. There is no allocation in this step, i.e., $A \cup \mathsf{A}(t \oplus (\mathbf{R}\ell \Rightarrow \texttt{ref}\ \ell')) = A \cup \mathsf{A}(t)$. We prove $K\,;A \cup \mathsf{A}(t)\,;F \vdash_\mu \Delta\,;[\bullet]\,;\ell'$ *drfde* using Rule (8.1) with the following:
  - $\mathsf{Loc}(\ell') \subseteq A \cup \mathsf{A}(t)$, established above
  - $\forall \ell \in (A \cup \mathsf{A}(t)) \setminus F.\ \mathsf{Loc}(\mu(\ell)) \subseteq A \cup \mathsf{A}(t)$, established above.
  - $\mathsf{Fut}(\ell', \mu) \subseteq K$. If ℓ' refers to a future cell, i.e. $\mu(\ell') = \texttt{fcell}[a]$ for some a, then we know that $a \in K$ because: $\ell' \in A \cup \mathsf{A}(t)$ and $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow a \in K$. Thus, in this case, $\mathsf{Fut}(\ell', \mu) = \{a\} \subseteq K$. Otherwise, $\mathsf{Fut}(\ell', \mu) = \emptyset$, which is trivially a subset of K.
  - $\forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \texttt{fcell}[a] \Rightarrow a \in K$, established above

□

4.3 Weakly Race-Free Programs are Disentangled

We consider a weaker notion of race freedom that allows "weak races" on small, word-size, values of *primitive* types, e.g., `int`, and `bool`. The distinguishing property of values of these primitive

```
1  val claimed = array n false
2
3  fun search (g: graph) (start: node) (goal: node) =
4    (start = goal) ||
5    (let children = filter_map
6         (fun i →
7            if cas (claimed[i], false, true) then
8              Some (future (search g i goal))
9            else
10             None)
11        (neighbors g start)
12     in
13       fold (fun found child → found || get child) false children)
```

Fig. 11. Parallel graph search using futures

types is that they do not require explicit allocation at runtime. The motivation for allowing such races comes from the practice of parallel programming, where many algorithms and their implementations [Kumar et al. 2017] use weak races for efficiency reasons.

Example. As an example of a class of algorithms that use weak races for improved efficiency and scalability, consider the graph-search example given in Figure 11. This function `search` takes a graph `g` and a `start` vertex and searches for a `goal` vertex. If the goal node hasn't been reached, the function attempts to claim its neighbors by using an atomic compare and swap against a bit vector `claimed`. The atomic operation `cas(r, exp, tar)` compares the expected value `exp` to the value stored at `r`; if they are equal, then it stores the target value `tar` and returns `true`, otherwise it returns `false`. If `cas` returns false, another thread has claimed the neighbor node. For each neighbor the current thread claims, the search creates a future to search the neighbor; each such neighbor can be thought of as child in the search tree defined by claimed neighbors of all vertices. The search then iterates over the list of children futures and waits for each to complete using the `get` operation.

Because of the compare-and-swap operation, this graph search algorithm is not determinacy race free and therefore violates a broadly accepted safety condition. Yet, such algorithms are common and many other graph algorithms use some variant of the same technique to avoid visiting vertices multiple times redundantly. There are several reasons for the use of such a race. First, the race is relatively simple to reason about, because it involves a trusted atomic operation implemented in hardware. Second, compare-and-swap operations are efficient in practice. Third, the alternative to the race is to implement a relatively sophisticated parallel or concurrent set data structure.

This example is disentangled even though it is not determinacy race free. Intuitively, the example remains disentangled because the objects involved in the race are all small value of primitive type `bool`. Because such values do not share allocations between concurrent tasks, they do not cause entanglement. In the rest of this subsection, we formalize this intuition and prove that races on primitive values do not violate disentanglement.

Language extension. We extend our language to distinguish primitive values from other values. We define the syntactic class ***small values***, denoted u, containing values of types bool and int, which the language no longer allocates in the memory store μ. For other types, the language has storables which the language allocates in the memory store μ by stepping them to memory locations. Thus, a value v in the language is either a location or a small value. We also extend the language with expression $\mathrm{cas}(e_1, e_2, e_3)$ that atomically performs a compare and swap. We leave much of its dynamics to the Appendix C, but show a key rule here, which corresponds to the case

where the cas operation succeeds. The dynamics of other expressions does not change.

$$\frac{\mu = \mu_0[\ell \hookrightarrow \texttt{ref}\ v_1'] \qquad v_1 = v_1' \qquad \mu' = \mu_0[\ell \hookrightarrow \texttt{ref}\ v_2]}{\Delta\ ;\ \mu_0[\ell \hookrightarrow s]\ ;\ [t]\ ;\ \texttt{cas}(\ell, v_1, v_2) \rightarrow \Delta\ ;\ \mu'\ ;\ [t \oplus (\mathbf{R}\ell \Rightarrow \texttt{ref}\ v_1') \oplus (\mathbf{U}\ell \Leftarrow \texttt{ref}\ v_2)];\texttt{true}}\ \textbf{CasS}$$

To step the operation $\texttt{cas}(\ell, v_1, v_2)$, the dynamics checks the value stored at the mutable reference ℓ. In the above step, the operation succeeds because the value v_1' stored at reference ℓ matches the first argument v_1 of the cas expression. This step, thus, modifies the memory reference ℓ to store v_2 and also updates the computation tree with read and write actions. The step returns the boolean `true` indicating that the operation succeeded.

Weak race freedom. For computation trees of our extended language, we define a property called *wrf* which stands for weak race freedom. We define the property as a judgment $F \vdash T\ wrf$, which is similar to the judgement $F \vdash T\ drf$ for determinacy race freedom, as it checks that no action in tree T mentions any location in the set F. Their rules are the same except for the following rule:

$$\frac{F \vdash t\ wrf \qquad F \cup \mathsf{A}(T_2) \cup \mathsf{LW}(T_2) \vdash T_1\ wrf \qquad F \cup \mathsf{A}(T_1) \cup \mathsf{LW}(T_1) \vdash T_2\ wrf}{F \vdash t \oplus (T_1 \otimes_a T_2)\ wrf}$$

This rule checks the tree $t \oplus (T_1 \otimes_a T_2)$ for the *wrf* property by checking the trace t and checking trees T_1 and T_2. When it checks tree T_1, it extends the set F with all the modifying actions of tree T_2, except those actions that read/write small values. The rule ignores actions like $\mathbf{R}\ell \Rightarrow \texttt{ref}\ u$ and $\mathbf{U}\ell \Leftarrow \texttt{ref}\ u$, thereby allowing races on small values. Specifically, the rule extends the set F with locations that are allocated by tree T_2, i.e., $\mathsf{A}(T_2)$ and also some of the locations that are modified by tree T_2, i.e., $\mathsf{LW}(T_2)$, ignoring those that operate on references of small values.

WRF ⇒ DE. Even though mutable references containing small values get special status when it comes to weak race freedom, they are still allocated in the memory store μ and need to be accounted for disentanglement. We show the following theorem.

Theorem 4.3 (WRF ⇒ DE). *For any $\emptyset\ ;\ \emptyset\ ;\ [\bullet]\ ;\ e_0 \rightarrow^n \Delta\ ;\ \mu\ ;\ T_n\ ;\ e_n$ where $\mathsf{Loc}(e_0) = \emptyset$, if every intermediate tree T_i in $\{T_1 \ldots T_n\}$ satisfies $\emptyset \vdash T_i\ wrf$, then $\emptyset \vdash T_n\ de$.*

We prove that the *wrf* property implies the *de* property, meaning weak races do not break disentanglement. For space reasons, we leave the proof to the Appendix, but the approach is similar to the proof of *drf* implies *de* (Theorem 4.1).

5 DISENTANGLEMENT GUARANTEES DEADLOCK FREE FUTURES

Futures are a powerful mechanism for parallelism. They are more general than fork-join and, as illustrated in prior sections, they are an excellent medium for pipelining, dynamic programming, graph algorithms, and interactive applications. But this generality comes at a cost: programs with futures can deadlock. A program deadlocks when there is a circular dependency among futures and all of them wait for others to complete. This prevents futures from making progress.

Disentanglement guarantees that futures don't get stuck, i.e., a disentangled execution can never deadlock. By disallowing concurrent threads from sharing their allocations, disentanglement also prevents sharing of futures. Disentanglement imposes a partial order on the "knowledge" of futures and prohibits cyclical dependencies. In the context of computation trees, disentanglement ensures that no two disjoint subtrees mention futures spawned in each other.

Assuming disentanglement, we make the following observation: a thread can synchronize with a future only if the future is spawned by an ancestor of the thread in the computation tree. This observation raises an interesting question because futures can return handles to other futures. Suppose a thread synchronizes with some future a, and receives a handle to another future, b, that

$\boxed{K\,;A \vdash_\mu \Delta\,;T\,;e\ kf}$

$$\frac{\mathsf{Fut}(e,\mu) \subseteq K \qquad \forall \ell \in A \cup \mathsf{A}(t).\ \mu(\ell) = \mathtt{fcell}[a] \Rightarrow a \in K}{K\,;A \vdash_\mu \Delta\,;[t]\,;e\ kf}$$

$$\frac{a \notin K \quad \Delta(a) \rhd v \quad K\,;A \cup \mathsf{A}(t) \vdash_\mu \Delta\,;[\bullet]\,;v\ kf \quad K \cup \{a\}\,;A \cup \mathsf{A}(t) \vdash_\mu \Delta\,;T\,;e\ kf}{K\,;A \vdash_\mu \Delta\,;t \oplus_a T\,;e\ kf}$$

$$\frac{a \notin K \quad \Delta(a) \blacktriangleright e_1 \quad K\,;A \cup \mathsf{A}(t) \vdash_\mu \Delta\,;T_1\,;e_1\ kf \quad K \cup \{a\}\,;A \cup \mathsf{A}(t) \vdash_\mu \Delta\,;T_2\,;e_2\ kf}{K\,;A \vdash_\mu \Delta\,;t \oplus (T_1 \otimes_a T_2)\,;e_2\ kf}$$

Fig. 12. The judgement *kf* checks that a future is only known to the descendants of its spawn node ($\otimes$) or join node ($\oplus$). K and A represent the set of known futures and allocations respectively.

was spawned by a. Now that the thread has a handle, it can synchronize with future b, which was concurrently spawned, and at first glance, it may appear that this breaks the aforementioned observation. However, there is a subtle point to consider. When the thread accesses future a and its return value, the future a has already joined, indicating that future a and the spawn point of future b are ancestors of the thread in the computation tree, even though the future b was spawned concurrently to the thread (see the right side tree in Figure 2). We prove the following theorem.

Theorem 5.1 (No Deadlock). *If* $\Delta_0\,;\mu_0\,;T_0\,;e_0 \to^n \Delta_n\,;\mu_n\,;T_n\,;e_n$ *and* $\forall i \le n.\ \emptyset \vdash T_i$ *de, then either* e_n *is a value and* $\forall a \in dom(\Delta_n), \exists v.\ \Delta_n(a) \rhd v$, *or* $\Delta_n\,;\mu_n\,;T_n\,;e_n \to \Delta_{n+1}\,;\mu_{n+1}\,;T_{n+1}\,;e_{n+1}$.

The theorem states that starting from the initial program state ($\Delta_0\,;\mu_0\,;T_0\,;e_0$), if we take an arbitrary number of steps that satisfy disentanglement, then the resulting state is either final, i.e., its expression and all its futures have been evaluated, or it can step. Thus, if an evaluation satisfies disentanglement, it never encounters a deadlock. We prove the theorem using progress and preservation techniques and leave its details to the Appendix. Here, we prove the key invariant.

Property kf. The key invariant that allows us to show the theorem is the property *kf*, which ensures that a thread can synchronize with a future only if the future is spawned by an ancestor of the thread in the computation tree. Given a program state ($\Delta\,;\mu\,;T\,;e$), we formalize the property with an inductive process (on the tree T) using the judgement $K\,;A \vdash_\mu \Delta\,;T\,;e\ kf$. The judgement checks that futures in subtree T only refer to locations in set A and futures in set K. The sets A and K correspond to the locations and futures that are created by the ancestors of subtree T.

Figure 12 shows the rules for the judgement. In the base case, when the tree is a leaf $[t]$, the judgement asserts that all the futures mentioned in expression e (represented as $\mathsf{Fut}(\mu, e)$) are in the set K. The judgement also checks that every location in set A only mentions futures in set K, ensuring that if an expression reads a location in set A, it does not discover a future outside set K. If the tree is of the form $t \oplus (T_1 \otimes_a T_2)$, the judgement checks subtree T_1 along with the expression e_1 of future a and checks subtree T_2 along with expression e_2 of the continuation. The rule extends the set K for checking the continuation but keeps it the same for checking the future (see Figure 12). This ensures that future a and any other futures spawned by future a can not access future a. This prevents cyclical dependencies and deadlocks.

DE $\Rightarrow$ *KF.* We prove that if a state satisfies the *kf* property and takes a step resulting in a state that satisfies the *de* property, then the resulting state also satisfies the *kf* property.

Lemma 5.2. *For any* $\Delta\,;\mu\,;T\,;e \to \Delta'\,;\mu'\,;T'\,;e'$ *if* $K\,;A \vdash_\mu \Delta\,;T\,;e\ kf$ *and* $A \vdash T'$ *de, then* $K\,;A \vdash_{\mu'} \Delta'\,;T'\,;e'\ kf$

We prove the lemma by induction on the stepping relation. It uses ideas similar to the proofs presented earlier in the paper and we leave the formal proof to the Appendix.

6 APPLICATIONS

Because it combines futures, state, and I/O, our language enables us to express a broad range of applications in a disentangled fashion. For example, in Section 3, we used our language to express a parallel tree merge algorithm and show that futures express pipelining in a succinctly and efficiently. In Section 4.1, we presented a PDF viewer, demonstrating the language's ability to handle asynchrony by utilizing futures and state to proactively compute and cache results, improving responsiveness. In Section 4.3, we showed a graph search algorithm, illustrating that the language can implement non-deterministic algorithms that use atomic read-modify-write operations on machine words. All of these programs satisfy disentanglement.

However, establishing disentanglement for such programs is not easy. Disentanglement is a low-level property of memory allocations that requires the programmer to reason about the program threads and their allocation behavior, which is particularly difficult in a high-level language that hides allocations from the programmer. Our main theorems can facilitate this reasoning because they establish determinacy race freedom as a sufficient condition for disentanglement. Because race freedom is a broadly accepted correctness condition for parallel programs, researchers have devised many methods for reasoning about race freedom and tools for detecting races [Feng and Leiserson 1997; Mellor-Crummey 1991; Xu et al. 2020]. Furthermore, in our context, determinacy race freedom is relatively easier to reason about because our language explicitly delineates mutable effects. In this section, we provide two examples to illustrate this. The first example is a web server involving interaction, and the second example is a dynamic programming algorithm leveraging data-dependent parallelism.

6.1 Web Server

Our web server listens for clients on its socket and spawns futures to handle their requests. Each future services exactly one client and as it does so, the future tracks the relevant information from their requests and aggregates it into a log object. Our current example simplifies the log to only include the name of the client and request count, but in practice, the log could contain many different statistics that we omit. When a client terminates the connection, the corresponding future produces a log of its interaction with the client and completes its evaluation.

The server synchronizes with the futures to obtain their logs. However, in order to respond to incoming clients efficiently, the server never blocks or waits on a future. The server achieves this by regularly polling the futures it spawned, filtering out the ones that have terminated, and synchronizing only with terminated futures. By only synchronizing with the terminated futures, the server ensures that it gets the logs immediately without creating any interruption.

Figure 13 shows the code for the server. The function `server` initializes a socket and proceeds to listen for incoming clients by calling the function `listen`. Subsequently, the server calls the function `loop`. The function `loop` accepts new clients, spawns futures to handle their requests, and collects and processes logs. The function `loop` maintains the spawned futures in a set named `clients`. Each time it accepts a client, the loop spawns a future to (concurrently) execute the function `process`, which services the requests of the client and returns the log.

Before spawning a future for a new client, the loop checks on the other clients by calling the function `handle_clients`. The function `handle_clients` takes the set of futures, filters those that have finished servicing their clients, and aggregates their logs. To filter out completed futures, the function uses the non-blocking primitive `fpoll`, which returns `true` for terminated futures (see line 14). Subsequently, the function uses the primitive `get` to obtain the logs. Note that each

```
type socket
type log = {name : string, requests: int}
start_socket : unit → socket
listen : socket → unit
accept : socket → socket option
process : socket → log

fun server () =
let
  server_sock = start_socket ()
  val _ = listen(server_sock)
  fun handle_clients (clients : log fut set) =
    let
      completed = filter (fn c => fpoll c) clients
      logs = map (completed, fn c => get c)
    in
      (logs, Set.diff (clients, completed))

  fun loop (clients  : log fut set) =
    let
      (logs, remaining_clients) = handle_clients (clients)
      (* Process logs as desired *)
    in
      case accept (server_sock) of
        NONE => loop (remaining_clients) (* No new clients *)
      | SOME client_sock =>
        let val f = fut (process (client_sock))
        in loop (Set.add (remaining_clients, f))
in
  loop (Set.empty ())
```

```
type request
recv : socket → request option
service : request → unit
name : socket → string

fun process (c) =
  let fun loop (req, cnt) =
    case req of
      NONE =>
      {name = name (c),
        count = cnt}
    | SOME req =>
      service (req);
      loop (recv (c), cnt + 1)
  in
    loop (recv (c), 0)
```

Fig. 13. A server with pollable futures and disentanglement

use of `get` returns immediately because the function only calls them on terminated futures. After aggregating the logs, the function removes the terminated futures from the set of futures, and returns the logs and the running futures back to the loop.

The function `loop` processes the logs it receives. We leave the log processing abstract as it varies with the use case. Then, it proceeds to process a new client, adds it to the set of clients, and repeats.

Overall the application uses the ability to store futures in a set and to poll them for managing clients without ever blocking. The application does not use any mutable effects and is purely interactive. Therefore, from the result that purely interactive programs are disentangled (Theorem 3.1), we get that application satisfies disentanglement. This is interesting because the server is interactive and involves communication between threads, and performs I/O, but it remains disentangled. The key point is that the server thread obtains a log only after the corresponding future has terminated, which, as we show in this paper, does not violate disentanglement.

6.2 Futures and References for Dynamic Programming

We consider a dynamic programming algorithm that tabulates an $M \times N$ matrix by computing a cell's value from the value of its neighboring cells in the row above the cell. Specifically, it computes the value at a cell (i, j) by applying an abstract function `f` to the values at cells $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$. The algorithm exemplifies a common pattern and has various applications, such as seam carving and sequence alignment [Avidan and Shamir 2007; Singer et al. 2019b]. In seam carving, for instance, the function `f` takes the minimum of the neighbor's values and adds a constant factor to compute the value of a cell.

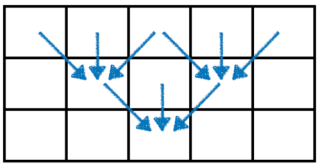

Fig. 14. An illustration of data-dependent parallelism in a DP matrix: two paths can proceed in parallel regardless of all the other elements in their respective rows.

```
1  fun f: α * α * α → α
2  val id : α
3  val arr = array2D (M, N, NONE)
4
5  fun lookup (i, j) =
6      if i < 0 || j < 0 || j ≥ M then id (* Out-of-bounds reads return id *)
7      else
8        case arr[i][j] of
9        | SOME r → get r
10       | NONE → raise Impossible
11
12 fun compute_cell (i, j) =
13    f (lookup (i - 1, j - 1), lookup (i - 1, j), lookup (i - 1, j + 1))
14
15 (*Initialize the array row-by-row in parallel*)
16 val _ = seq_fill M
17   (fun i → par_fill N
18     (fun j → arr[i][j] ← SOME (future (compute_cell (i, j)))))
19 val result = lookup (M - 1, N - 1)
```

Fig. 15. Dynamic programming with futures, state, and disentanglement

To tabulate the matrix, we could implement an algorithm that proceeds in a row-by-row manner and fills each row in parallel (because cells of a row do not depend on each other). This algorithm, however, does not exploit parallelism across the rows. In particular, once three consecutive cells of row i are computed, the middle cell in the next row $i + 1$ can be computed without waiting for the rest of the cells of row i. Because such "vertical" parallelism depends on the data flow, it is impossible to express with fork-join parallelism, but is naturally expressible by using a combination of futures and state. Figure 14 illustrates the flow of data and parallelism present in this application.

Figure 15 shows an algorithm where futures unleash the data dependent parallelism across rows. The algorithm represents each cell of the matrix with a future, which waits for the neighbors to complete and then computes the cell's value by using the function `compute_cell`. The algorithm starts by initializing each cell of a mutable $M \times N$ array `arr` with `NONE`. It then fills the array with futures. To do so, the algorithm proceeds in a row-by-row manner and writes the futures of a row in parallel, ensuring that a future is spawned only after the futures it depends on have been spawned. Each future executes the function `compute_cell` which synchronizes with the neighboring cells by calling the function `lookup`, a function that handles boundary conditions around the edges of the matrix, waits for a cell to finish using expression `get`, and returns its value.

The algorithm satisfies disentanglement but this is not easy to establish by reasoning about the the memory allocations of the program. One potential concern arises from the fact that futures read handles to other futures from the array, and since these handles themselves are allocated concurrently, reading them could create entanglement. However, we can see that each future only

reads those indices of the shared array which are tabulated before the future is spawned. Thus, no future witnesses the concurrent updates of the array (see line 10) and the code satisfies determinacy race freedom. Using the result that race freedom implies disentanglement (Theorem 4.1), we get that the code satisfies disentanglement.

7 RELATED WORK

From Fork-Join Parallelism to Futures. Fork-join parallelism has proved to be an effective model for many parallel computations [Blumofe et al. 1996; Frigo et al. 1998; Lea 2000]. Fork-join parallelism, however, is only effective for a set of computations that, especially viewed from the angle of increasing hardware parallelism, is increasingly limited. For example, it is difficult, if not impossible, to use fork-join parallelism to express parallel tasks that execute asynchronously until a data-driven condition, e.g., based on input from the user, is satisfied. Muller et al.'s work shows that futures can provide this kind of asynchrony, which is pervasive in interactive applications [Muller et al. 2020; Muller and Acar 2016; Muller et al. 2017, 2023, 2019; Singer et al. 2020b]. This line of work observed that, when combining asynchronous interaction and fine-grained compute-heavy parallelism, it is necessary to assign higher *priorities* in the scheduler to the interactive threads to maintain responsiveness. Priorities are orthogonal to the theory of disentanglement we explore in this paper, and so we did not add them to this theory to keep the focus on disentanglement.

Even if we restrict ourselves to computational applications, many of which can in principle be expressed in fork-join parallelism, techniques such as pipelining cannot be expressed in fork-join [Blelloch and Reid-Miller 1999]. The primary reason for this is that fork-join parallelism is only able to capture parallelism exposed by the control dependencies of the computation, whereas futures can capture parallelism via data dependencies [Acar et al. 2016]. A secondary reason is that with futures, parallelism may be treated as a "first-class" value in a computation, e.g., futures may be stored in ordinary data structures (we used this property in several examples in this paper)—this is not possible with fork-join parallelism.

Researchers have therefore converged on using futures for increasing the expressiveness of parallel languages. Futures were invented in the 1970s [Baker and Hewitt 1977] and were brought to their current form by Halstead in the 1980s [Halstead 1985]. Today, many concurrent and parallel programming systems support futures, including Cilk-F/L [Singer et al. 2020a, 2019a], I-Cilk [Muller et al. 2020], Concurrent Haskell [Hammond 2011; Li et al. 2007; Marlow and Peyton Jones 2011; Peyton Jones et al. 2008], Habanero Java [Imam and Sarkar 2014], Parallel ML [Acar et al. 2020; Arora et al. 2021, 2023; Fluet et al. 2008, 2011; Guatto et al. 2018; Ohori et al. 2018; Raghunathan et al. 2016; Sivaramakrishnan et al. 2014; Spoonhower 2009a; Westrick et al. 2020], OCaml [Dolan et al. 2018a; LWT 2022], Rust [Rust Team 2019], and TPL (a .NET library) [Leijen et al. 2009].

Futures can also be challenging to manage in the run-time system of a programming language. For example, data locality of parallel programs with futures can significantly worsen when they execute in parallel [Acar et al. 2002] and restricting their expressiveness can improve the data locality [Herlihy and Liu 2014; Spoonhower et al. 2009].

Race Freedom. As we established in this paper, disentanglement is implied by freedom from determinacy races [Feng and Leiserson 1997; Netzer and Miller 1992]. Determinacy races, also called general races, cause non-determinism and are considered bugs for programs that are intended to be deterministic [Netzer and Miller 1992]. Absence of determinacy races guarantees determinism: in every execution, the executed instructions and their execution order are the same. Determinacy races are different from *data races*, which only occur when a critical section of the code is not executed atomically. Unlike data races, determinacy races are quite conservative and can include accesses that produce deterministic outcomes. For example, atomic fetch-and-add operations by concurrent

threads do not cause a data race because each increment is performed atomically. However, such operations cause a determinacy race because the execution order is nondeterministic.

We do not propose a way to check for disentanglement in this paper, but there has been work on checking and managing entanglement in fork-join programs [Arora et al. 2023; Westrick et al. 2022]. There is a lot of work checking for race freedom and bounding the impact of races, partly because data races usually cause incorrect behavior [Adve 2010; Boehm 2011; Dolan et al. 2018b]. Many algorithms for race detection in fork-join parallel programs have been proposed [Bender et al. 2004; Cheng et al. 1998; Feng and Leiserson 1997; Fineman 2005; Mellor-Crummey 1991; Raman et al. 2010, 2012; Utterback et al. 2016; Xu et al. 2020]. More recent work considers race detection for futures [Xu et al. 2020]. Because our theorems establish race freedom as a sufficient condition for disentanglement, we can leverage all of this work to check for disentanglement.

There has also been significant research on race detection for more general concurrent programs [Flanagan and Freund 2009; Kini et al. 2017; O'Callahan and Choi 2003; Savage et al. 1997; Smaragdakis et al. 2012; Yu et al. 2005]. Such programs differ from task-parallel programs, because they use coarse-grained threads and synchronize in an unstructured manner, using locks and other synchronization primitives. The two classes of programs therefore typically require different approaches for reasons of efficiency, soundness (ability to correctly detect races), and completeness (ability to detect all races). In the paper, we have also proved that a weaker form of race freedom, which allows for determinacy races on primitive data types, does not break disentanglement. We are not aware of prior work that studied such "weak" races, but this could be of interest, because many practical parallel programs today employ such races to improve efficiency.

Disentanglement. The motivation for this work comes from recent results on efficient task parallelism in functional languages [Arora et al. 2021, 2023; Guatto et al. 2018; Raghunathan et al. 2016; Westrick et al. 2022, 2020]. The goal of all this work is to develop provably and practically efficient and scalable parallel memory management techniques for nested-parallel functional languages by taking advantage of disentanglement, a memory property that was initially discovered for fork-join pure functional programs [Raghunathan et al. 2016] , was later extended to account for mutable state [Guatto et al. 2018; Westrick et al. 2020]. Our definition of disentanglement is consistent with these works, i.e., if we restrict our language to fork-join, the class of programs that satisfy our definition of disentanglement is the same as previous works. Our disentanglement definition and the corresponding results are strictly more general because futures are more expressive than fork-join (we can implement fork join constructs using futures) [Spoonhower 2009b].

More recently, disentanglement was proposed as a continuous object-level property with the hypothesis that most objects in all programs are disentangled. The hypothesis is supported by a theoretical observation that objects become entangled only when they participate in a race and a practical observation that races are rare in parallel programs. Using the hypothesis, the work culminates in language called MPL that implements a provably and practically efficient memory manager for fork-join programs. [Arora et al. 2021, 2023]. We believe our techniques are compatible with the memory manager used in MPL. First, because our results establish race free programs to be disentangled, we expect that the disentanglement hypothesis holds for parallel programs with futures. This is important because the memory manager in MPL is designed to exploit disentanglement and its efficiency relies on the observation that entanglement is rare. Second, our semantics shows that the heap trees in MPL can be generalized for futures, as the semantics produces a computation tree for each step of the execution; each node (action trace) of the computation tree would be a heap in the MPL runtime. However, one engineering challenge is extending the specialized scheduling infrastructure of MPL to support futures, which have a more complex dependency structure than fork-join.

Deadlock. The expressiveness of futures can make them harder to use safely. One important concern is deadlock: with futures, it is possible to create cyclic dependencies in a computation that prevent the computation from making progress. Cogumbreiro et al. identify this problem and formulate two properties called "known joins" and "transitive joins" that can be enforced by restricting the expressive power of joins [Cogumbreiro et al. 2017; Voss et al. 2019] to prevent deadlock. They achieve this by enforcing a discipline on the use of futures, i.e., they forbid tasks from synchronizing with certain other futures. Loosely speaking, the known joins property states that task A is allowed to sync with future B only if B is spawned by one of the ancestors of A. The transitive joins work relaxes the known joins restriction by adding transitivity, i.e., if task A can synchronize with B and B can synchronize with C, then A is allowed to synchronize with C. The known joins property is naturally satisfied in disentangled executions—in a disentangled execution, a task only knows about those locations/futures that are allocated/spawned by its ancestors (thus, as we have already proven separately in this paper, disentanglement implies deadlock freedom). Prior work has shown that determinacy race freedom implies known joins. We showed that determinacy race freedom implies disentanglement and also introduce weak race freedom, not considered in the above work, and showed that this more practical notion implies disentanglement.

8 CONCLUSION

In the past several years, researchers have discovered that parallel functional programs, even effectful ones, exhibit a memory property, called disentanglement. This property requires that concurrent threads remain oblivious to each other's allocations. Initially, disentanglement was motivated by efficiency and performance concern in parallel functional programming languages. But recent work shows that it is also a property of independent interest, because it is exhibited by a variety of parallel programs, even those written in low-level parallel programming languages such as C/C++ [Arora et al. 2023; Wilkins et al. 2023]. All of this prior work on disentanglement assumes fork-join (nested) parallel programs and take advantage of their structured dependencies, which can be represented by "series-parallel" directed acyclic graphs, which allow only serial and parallel composition.

In this paper, we show that a broad range of parallel programs that use futures are also disentangled under certain conditions, even as they perform I/O and use mutable state. These results were surprising to us—we did not expect that the complex dependency structure of parallel programs with futures, which allows asynchronous and data-dependent dependencies between threads/tasks, could be "tamed" to establish disentanglement. We observe that futures rely on one-way synchronization, where synchronization occurs from the future being read to the reader, in contrast to the two-way synchronization as in fork-join. Using this observation, we develop techniques to "comb" the complex dependency structure and ensure disentanglement. We show that futures, when combined with I/O and mutation, expand the applicability of disentanglement-based approaches to interactive programs. In this more general setting with I/O and mutation, futures effectively provide an implementation technique for "first-class" threads, which can then be used to implement a variety of concurrency patterns, raising thus the possibility of deadlock. As a final result, we prove that disentanglement ensures deadlock-freedom.

ACKNOWLEDGMENTS

This research was supported by the following grants NSF (CCF-1901381, CCF-2115104,CCF-2119352, CCF-2107241) and by a gift from Intel.

REFERENCES

Umut A. Acar, Jatin Arora, Matthew Fluet, Ram Raghunathan, Sam Westrick, and Rohan Yadav. 2020. MPL: A High-Performance Compiler for Parallel ML. https://github.com/MPLLang/mpl.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theory of Computing Systems* 35, 3 (2002), 321–347.

Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016. Dag-calculus: A Calculus for Parallel Computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. 18–32.

Sarita V. Adve. 2010. Data races are evil with no exceptions: technical perspective. *Commun. ACM* 53, 11 (2010), 84.

T. R. Allen and D. A. Padua. 1987. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the 1987 International Conference on Parallel Processing*. 721–727.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space Efficient Parallel Functional Programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. Efficient Parallel Functional Programming with Effects. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1558–1583. https://doi.org/10.1145/3591284

Shai Avidan and Ariel Shamir. 2007. Seam Carving for Content-Aware Image Resizing. In *ACM SIGGRAPH 2007 Papers* (San Diego, California) *(SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 10–es. https://doi.org/10.1145/1275808.1276390

Henry G. Baker and Carl E. Hewitt. 1977. *The Incremental Garbage Collection of Processes.* AI memo 454. MIT Press.

Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*. 133–144.

Guy Blelloch and Margaret Reid-Miller. 1999. Pipelining with Futures. *Theory of Computing Systems* 32, 3 (1999), 213–239.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55 – 69.

Robert L. Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *ACM POPL*.

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (Orlando, Florida, USA) *(OOPSLA '09)*. 97–116.

Hans-Juergen Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *3rd USENIX Workshop on Hot Topics in Parallelism, HotPar'11, Berkeley, CA, USA, May 26-27, 2011*.

Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*.

Tiago Cogumbreiro, Rishi Surendran, Francisco Martins, Vivek Sarkar, Vasco T. Vasconcelos, and Max Grossman. 2017. Deadlock avoidance in parallel programs with futures: why parallel tasks should not wait for strangers. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 103:1–103:26.

Richard Cole. 1988. Parallel merge sort. *SIAM J. Comput.* 17, 4 (1988), 770–785.

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2018a. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 98–117.

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018b. Bounding Data Races in Space and Time. *SIGPLAN Not.* 53, 4 (jun 2018), 242–255. https://doi.org/10.1145/3296979.3192421

Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. 1991. Event Synchronization Analysis for Debugging Parallel Programs. In *Supercomputing '91*. 580–588.

Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1–11.

Jeremy T. Fineman. 2005. *Provably Good Race Detection That Runs in Parallel.* Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. *SIGPLAN Not.* 44, 6 (June 2009), 121–133. https://doi.org/10.1145/1543135.1542490

Matthew Fluet, Mike Rainey, and John Reppy. 2008. A scheduling framework for general-purpose parallel languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (2011), 1–40.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

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*. 81–93.

Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7 (1985), 501–538.

Kevin Hammond. 2011. Why Parallel Functional Programming Matters: Panel Statement. In *Reliable Software Technologies - Ada-Europe 2011 - 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings*. 201–205.

Maurice Herlihy and Zhiyu Liu. 2014. Well-structured Futures and Cache Locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. ACM, New York, NY, USA, 155–166.

Shams Mahmood Imam and Vivek Sarkar. 2014. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*. 75–86.

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 157–170.

Ananya Kumar, Guy E. Blelloch, and Robert Harper. 2017. Parallel functional arrays. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 706–718.

Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. 36–43.

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. 227–242.

Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*. 107–118.

LWT. 2022. LWT OCaml. GitHub. https://github.com/ocsigen/lwt

Simon Marlow. 2011. Parallel and Concurrent Programming in Haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7241)*, Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer (Eds.). Springer, 339–401.

Simon Marlow and Simon L. Peyton Jones. 2011. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, 21–32.

John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing'91*. 24–33.

Stefan Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 71–82.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2017. Responsive Parallel Computation: Bridging Competitive and Cooperative Threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 677–692.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Types and Cost Models for Responsive Parallelism. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '18)*.

Stefan K. Muller, Kyle Singer, Devyn Terra Keeney, Andrew Neth, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2023. Responsive Parallelism with Synchronization. *Proc. ACM Program. Lang.* 7, PLDI (2023), 712–735.

Stefan K. Muller, Sam Westrick, and Umut A. Acar. 2019. Fairness in Responsive Parallelism. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2019)*.

Robert H. B. Netzer and Barton P. Miller. 1992. What are Race Conditions? *ACM Letters on Programming Languages and Systems* 1, 1 (March 1992), 74–88.

Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, Rudolf

Eigenmann and Martin C. Rinard (Eds.). ACM, 167–178.

Atsushi Ohori, Kenjiro Taura, and Katsuhiro Ueno. 2018. Making SML# a General-purpose High-performance Language. Unpublished Manuscript.

Wolfgang Paul, Uzi Vishkin, and Hubert Wagener. 1983. Parallel dictionaries on 2–3 trees. In *International Colloquium on Automata, Languages, and Programming*. Springer, 597–609.

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*. 383–414.

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. ACM, New York, NY, USA, 392–406.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer Berlin / Heidelberg, 368–383.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 531–542.

Rust Team. 2019. Rust Language. https://www.rust-lang.org/

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*.

Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. 2020a. Scheduling I/O Latency-Hiding Futures in Task-Parallel Platforms. In *1st Symposium on Algorithmic Principles of Computer Systems, APOCS 2020, Salt Lake City, UT, USA, January 8, 2020*, Bruce M. Maggs (Ed.). SIAM, 147–161. https://doi.org/10.1137/1.9781611976021.11

Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2020b. Priority Scheduling for Interactive Applications. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, Christian Scheideler and Michael Spear (Eds.). 465–477.

Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019a. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/3293883.3295735

Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019b. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 257–271. https://doi.org/10.1145/3293883.3295735

K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming* FirstView (6 2014), 1–62.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 387–400.

Daniel Spoonhower. 2009a. *Scheduling Deterministic Parallel Programs*. Ph. D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf

Daniel Spoonhower. 2009b. *Scheduling Deterministic Parallel Programs*. Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA.

Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. 2009. Beyond Nested Parallelism: Tight Bounds on Work-stealing Overheads for Parallel Futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. ACM, New York, NY, USA, 91–100.

Guy L. Steele Jr. 1990. Making Asynchronous Parallelism Safe for the World. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 218–231.

Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 83–94.

Caleb Voss, Tiago Cogumbreiro, and Vivek Sarkar. 2019. Transitive joins: a sound and efficient online deadlock-avoidance policy. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). 378–390.

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection With Near-Zero Cost. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2022)*.

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

Michael Wilkins, Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Armenio Deiana, Simone Campanoni, Umut A. Acar, Peter Dinda, and Nikos Hardavellas. 2023. WARDen: Specializing Cache Coherence for High-Level Parallel Languages. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO 2023)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3579990.3580013

Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. 2020. Parallel determinacy race detection for futures. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 217–231. https://doi.org/10.1145/3332466.3374536

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, Andrew Herbert and Kenneth P. Birman (Eds.). ACM, 221–234.

Received 2023-07-11; accepted 2023-11-07