Disentanglement, Theory and Practice

Rohan Yadav

May 2019

Advised by

Umut Acar

Abstract

In this thesis, we investigate a notion of independence in parallel programs called *disentanglement*. Informally, disentanglement is the property that threads are oblivious to data allocated by concurrently executing threads. This property prevents certain types of concurrent access, but still permits types of races on shared data.

Disentanglement has immediate implications for efficient memory management strategies. For nested parallel programs, a hierarchical memory management scheme which reflects the structure of parallel tasks can offer numerous benefits with respect to data locality and garbage collection.

We formally describe and analyze disentanglement, and show all programs that are determinacy race free are disentangled. Determinacy race freedom is an important correctness property for parallel programs, and is well studied in literature. Additionally, we discuss how disentanglement can be used to architect efficient memory management techniques, specifically, a concurrent garbage collector with minimal interference between the mutator and collector.

1 Introduction

Advances in computing hardware have brought parallel computation into the mainstream. Writing good parallel programs continues to be a challenging task for programmers, mainly due to race conditions and concurrency bugs that can arise when using parallel imperative languages. These concurrency issues are difficult to diagnose and debug especially as the program scales to higher core counts. On the other hand, functional languages express parallelism implicitly, and are naturally race free due to either limited or no mutation. Functional languages have a clear semantics, and allow the programmer to express parallelism in their algorithm, as opposed to the implementation. Aggregate operations like map, tabulate, filter, reduce, etc. are powerful abstractions to express algorithms succinctly, and are well supported in a functional paradigm. Based on these ideas, many parallel functional languages have been implemented and studied [11, 25, 27, 36].

However, a traditional problem with functional languages is that they are not as efficient as imperative languages, especially in the parallel context. This gap arises from the nature of functional languages to avoid mutation and side effects, which leads to functional languages copying data structures instead of performing efficient updates to them. The increased copying that functional programs perform lead to heavily increased allocation rates and memory traffic, which affect performance on modern machines considerably. Functional programs are not restricted from using mutation though, as modern functional languages like OCaml and Standard ML support effects through reference types. Even when a parallel algorithm has a functional interface, uses of mutation internally, such as at small input sizes for recursive algorithms, can greatly improve the performance. However, parallel functional languages have difficulty supporting mutation efficiently due to toll on their memory management systems. Functional programs allocate memory at a fast rate, and supporting mutation at the same time can be costly to the runtime system [7, 8, 18, 19]. Additionally, effects can break the independence assumptions that memory managers make for functional languages, causing such operations to be expensive [26, 30].

Instead of supporting general effects, we instead consider a restricted set of effects through a property called *disentanglement*, and discuss efficient memory management techniques for disentangled programs. General effects can lead to race conditions and concurrency bugs, but when used in a ”disciplined” manner, they can be more easily reasoned about and supported by a language's runtime system. Intuitively, disentanglement entails that a task is oblivious to allocations performed by all tasks concurrent to it. We give more precise definitions of disentanglement in Sections 2 and 4. Effects that maintain this property weaken consistency requirements in memory management, as effects performed by a task are delayed until the task has synchronized with other tasks.

In this thesis, we propose a formal definition for disentanglement as a property of programs through an ML-like language. We then formally show that the set of all determinacy race free programs are exhibit this disentanglement property. Determinacy race freedom is a well studied topic in parallel computing, and an important correctness property of parallel programs [20, 21, 22]. Then, we describe existing techniques for memory management for disentangled programs, and address weaknesses in these techniques with a novel concurrent garbage collection scheme for disentangled programs.

2 Background

We present a brief overview of disentanglement and hierarchical memory management, and build on these ideas in the rest of the thesis. We present a simple example to illustrate the programming model and some details of memory

```
fun quicksort l =
  case l of
    [] ⇒ []
  | [x] ⇒ [x]
  | x::xs ⇒
    let
      (ll, lr) = partition x xs
      (sll, slr) = par (quicksort ll, quicksort lr)
    in
      sll @ [x] @ slr
    end
```

Figure 1: Code for quicksort.

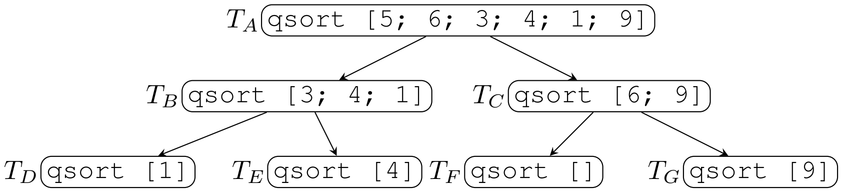

Figure 2: An example run of quicksort. The nodes represent (parallel) tasks and the edges represent the control dependencies between them.

management. In doing so, we introduce some terminology that will be used in the rest of the thesis.

Consider the parallel quicksort in Figure 1. The implementation is a standard quicksort, which partitions the input, and then recursively sorts the partitioned halves. Parallelism is exposed to the programmer through the `par` construct, which creates new tasks. Using `par`, tasks may spawn two new tasks, which establishes a parent/child relationship between tasks. Figure 2 shows the task tree that results from executing quicksort on the list `[5; 6; 3; 4; 1; 9]`.

Clasically, memory is viewed in programs as a flat pool of memory, or a fixed hierarchy of multiple heaps called generations. Raghunathan et al [35] proposes a form of hierarchical memory management, in which each task is assigned its own heap to allocate data within. Along with the task tree created by dependencies between tasks, a *heap tree* is created with structure that mirrors the task tree. We refer to the heap tree as the *hierarchy*. When a task is created, it is given a fresh heap to allocate objects into. When two tasks complete, they join thier heaps into the heap of thier parent, and the parent continues executing. In Figure 1, we can see this structure, as each task holds the lists it creates from partitioning the input list, and the list it creates from appending the sorted results (we only show the sorted results).

Previous work identifies a *disentanglement* property on the hierarchy. Specifically, a hierarchy is disentangled if for all objects x in heap H_x, the following property holds. If there is a pointer in x to another object y residing in heap H_y, then H_y must be an ancestor of H_x in the hierarchy. Visually, these means that all pointers in the hierarchy ”point up” from tasks to their ancestors. Because pointers only point up in the hierarchy, garbage collection on leaves in the hierarchy can proceed without synchronization with other tasks. The property of ”pointing up” means

$$\begin{array}{rccl}
\textit{Types} & \tau & ::= & \mathsf{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \tau\ \mathsf{ref} \\
\textit{Memory Locations} & \ell & & \\
\textit{Storables} & s & ::= & n \mid \mathsf{fun}\ f\ x\ \mathsf{is}\ e \mid \langle \ell, \ell \rangle \mid \mathsf{ref}\ \ell \\
\textit{Expressions} & e & ::= & x \mid \ell \mid s \mid e\ e \mid \langle e, e \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{ref}\ e \mid !\,e \mid e := e \mid \langle e \parallel e \rangle \\
\textit{Tasks} & T & ::= & e \mid T\ e \mid \ell\ T \mid \langle T, e \rangle \mid \langle \ell, T \rangle \mid \mathsf{fst}\ T \mid \mathsf{snd}\ T \mid \mathsf{ref}\ T \mid !\,T \mid T := e \mid \ell := T \mid (\!| H \cdot T \parallel H \cdot T |\!) \\
\textit{Heaps} & H & ::= & \emptyset \mid H \uplus \{\ell\} \\
\textit{Memory} & \mu & ::= & \emptyset \mid \mu[\ell \hookrightarrow s]
\end{array}$$

Figure 3: Syntax of our language

that tasks never gain information about allocations made by tasks concurrent to them. Raghunathan et al use this independence property to implement efficient memory management and garbage collection.

This disentanglement property is guaranteed in purely functional programs, as there is no way of changing the direction of a pointer. However, purely functional programs are not as efficient as those with mutation. To allow for mutation, work by Guatto et al [26] allow for generation mutation, but restructure the memory when mutation occurs. When a write occurs that might create a pointer that "points down" in the hierarchy, they *promote* the object to a place in the hierarchy where a down pointer can no longer be created. This strategy maintains the disentanglement property, but is expensive in practice, especially when updating objects shared by many tasks.

Our work and current work instead strengthens the disentanglement property to allow for both "up" and "down" pointers, but still disallowing pointers that point across the tree. Specifically, an object x in H_x can have pointers to only ancestors and descendants of H_x. This strengthening property allows for mutation, but only in restricted ways. Additionally, these guarantees are still strong enough to allow for independence in garbage collection. This thesis develops the formalism for *disentanglement*, shows the determinacy race free programs are disentangled, and develops memory management techniques for disentangled programs.

3 Language

To formalize the intuitive notion of disentanglement, we present a ML-like language to model memory managed languages. Then, we define disentanglement as a property on programs within this language.

3.1 Abstract Syntax

Figure 3 shows the abstract syntax of our language. The types include natural numbers, functions, products, and mutable references. To provide the notion of memory in our language, we distinguish between locations ℓ, and storables s, which are stored in the heap. The only irreducible values in our language are locations, as storables will step to locations. We distinguish between *expressions* and *tasks* in our language as well. The main difference between expressions and tasks are that tasks can contain the activated parallel tuple, while expressions can only contain the unactivated parallel tuple. This distinction is made to enforce the structure of parallel programs in our language, so that the only active parallel tuples are those created through evaluation of tasks. In our language, tasks are paired with *heaps*, which are sets of locations ℓ that the task has allocated. We use a global memory μ to represent the memory the memory of the program during execution. The memory is a mapping from locations to

Memory Typing $\boxed{\Sigma' \vdash \mu : \Sigma}$

$$\frac{}{\Sigma \vdash \emptyset : \cdot} \qquad \frac{\Sigma', \ell{:}\tau \vdash \mu : \Sigma \qquad \cdot\,;(\Sigma,\Sigma') \vdash s : \tau}{\Sigma' \vdash \mu[\ell \hookrightarrow s] : \Sigma, \ell{:}\tau}$$

Expression Typing $\boxed{\Gamma\,;\Sigma \vdash e : \tau}$

$$\frac{(x:\tau) \in \Gamma}{\Gamma\,;\Sigma \vdash x : \tau} \qquad \frac{(\ell:\tau) \in \Sigma}{\Gamma\,;\Sigma \vdash \ell : \tau} \qquad \frac{}{\Gamma\,;\Sigma \vdash n : \mathsf{nat}}$$

$$\frac{(\Gamma, f{:}\tau_2 \to \tau_1, x{:}\tau_2)\,;\Sigma \vdash e : \tau_1}{\Gamma\,;\Sigma \vdash (\mathsf{fun}\ f\ x\ \mathsf{is}\ e) : \tau_2 \to \tau_1} \qquad \frac{\Gamma\,;\Sigma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma\,;\Sigma \vdash e_2 : \tau_2}{\Gamma\,;\Sigma \vdash e_1\ e_2 : \tau_1}$$

$$\frac{\Gamma\,;\Sigma \vdash e_1 : \tau_1 \qquad \Gamma\,;\Sigma \vdash e_2 : \tau_2}{\Gamma\,;\Sigma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma\,;\Sigma \vdash e : \tau_1 \times \tau_2}{\Gamma\,;\Sigma \vdash \mathsf{fst}\ e : \tau_1} \qquad \frac{\Gamma\,;\Sigma \vdash e : \tau_1 \times \tau_2}{\Gamma\,;\Sigma \vdash \mathsf{snd}\ e : \tau_2}$$

$$\frac{\Gamma\,;\Sigma \vdash e : \tau}{\Gamma\,;\Sigma \vdash \mathsf{ref}\ e : \tau\ \mathsf{ref}} \qquad \frac{\Gamma\,;\Sigma \vdash e : \tau\ \mathsf{ref}}{\Gamma\,;\Sigma \vdash\ !\,e : \tau} \qquad \frac{\Gamma\,;\Sigma \vdash e_1 : \tau\ \mathsf{ref} \qquad \Gamma\,;\Sigma \vdash e_2 : \tau}{\Gamma\,;\Sigma \vdash e_1 := e_2 : \tau}$$

$$\frac{\Gamma\,;\Sigma \vdash e_1 : \tau_1 \qquad \Gamma\,;\Sigma \vdash e_2 : \tau_2}{\Gamma\,;\Sigma \vdash \langle e_1 \parallel e_2 \rangle : \tau_1 \times \tau_2}$$

Heap-Task Typing $\boxed{\Gamma\,;\Sigma\,;\Delta \vdash H{\cdot}T : \tau}$

$$\frac{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau \qquad H \subseteq \mathrm{dom}(\Sigma)}{\Gamma\,;\Sigma\,;(H \uplus \Delta) \vdash H{\cdot}T : \tau}$$

Task Typing $\boxed{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau}$

$$\frac{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau_2 \to \tau_1 \qquad \Gamma\,;\Sigma \vdash e : \tau_2}{\Gamma\,;\Sigma\,;\Delta \vdash (T\ e) : \tau_1} \qquad \frac{(\ell : \tau_2 \to \tau_1) \in \Sigma \qquad \Gamma\,;\Sigma\,;\Delta \vdash T : \tau_2}{\Gamma\,;\Sigma\,;\Delta \vdash (\ell\ T) : \tau_1}$$

$$\frac{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau_1 \qquad \Gamma\,;\Sigma \vdash e : \tau_2}{\Gamma\,;\Sigma\,;\Delta \vdash \langle T, e \rangle : \tau_1 \times \tau_2} \qquad \frac{(\ell : \tau_1) \in \Sigma \qquad \Gamma\,;\Sigma\,;\Delta \vdash T : \tau_2}{\Gamma\,;\Sigma\,;\Delta \vdash \langle \ell, T \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau_1 \times \tau_2}{\Gamma\,;\Sigma\,;\Delta \vdash \mathsf{fst}\ T : \tau_1} \qquad \frac{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau_1 \times \tau_2}{\Gamma\,;\Sigma\,;\Delta \vdash \mathsf{snd}\ T : \tau_2} \qquad \frac{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau}{\Gamma\,;\Sigma\,;\Delta \vdash \mathsf{ref}\ T : \tau\ \mathsf{ref}} \qquad \frac{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau\ \mathsf{ref}}{\Gamma\,;\Sigma\,;\Delta \vdash\ !\,T : \tau}$$

$$\frac{\Gamma\,;\Sigma\,;\Delta \vdash T : \tau\ \mathsf{ref} \qquad \Gamma\,;\Sigma \vdash e : \tau}{\Gamma\,;\Sigma\,;\Delta \vdash (T := e) : \tau} \qquad \frac{(\ell : \tau\ \mathsf{ref}) \in \Sigma \qquad \Gamma\,;\Sigma\,;\Delta \vdash T : \tau}{\Gamma\,;\Sigma\,;\Delta \vdash (\ell := T) : \tau}$$

$$\frac{\Gamma\,;\Sigma \vdash e : \tau}{\Gamma\,;\Sigma\,;\emptyset \vdash e : \tau} \qquad \frac{\Gamma\,;\Sigma\,;\Delta_1 \vdash H_1{\cdot}T_1 : \tau_1 \qquad \Gamma\,;\Sigma\,;\Delta_2 \vdash H_2{\cdot}T_2 : \tau_2}{\Gamma\,;\Sigma\,;(\Delta_1 \uplus \Delta_2) \vdash (\!|\, H_1{\cdot}T_1 \parallel H_2{\cdot}T_2 \,|\!) : \tau_1 \times \tau_2}$$

Descendant Heap Locations

$$\begin{aligned} D(e) &\triangleq \emptyset \\ D(T\ e) \triangleq D(\ell\ T) \triangleq \cdots &\triangleq D(T) \\ D((\!|\, H_1{\cdot}T_1 \parallel H_2{\cdot}T_2 \,|\!)) &\triangleq H_1 \cup H_2 \cup D(T_1) \cup D(T_2) \end{aligned}$$

Figure 4: Statics of our language. Interesting rules are highlighted.

storables, and is updated and extended during execution.

3.2 Statics

Figure 4 describes the statics of our language. The memory μ has type signature Σ, where Σ holds the types of all locations in μ. Expressions are typed under a variable context Γ and a memory context Σ, and are typed in a standard way. The only thing to note is that a location is typed according to the type of that location under the memory context Σ. The Heap-Task typing judgment types a heap-task pair. The judgment ensures that all locations within the heap are within the memory context Σ and that the task itself is well typed. The typing judgment for tasks is similar to that of expressions, but includes the location set Δ. These rules are all similar to that of expressions, except for the typing rule for activated parallel tuples. In the activated parallel tuple, we ensure through the typing judgment that all locations mentioned in H_1 and the heaps of T_1 are distinct from those in H_2 and the heaps of T_2. This typing judgment ensures statically that there is no overlap between heaps of parallel tasks in activated parallel tuples. Lastly, we include an inductive definition to extract all locations in the heaps task.

3.3 Dynamics

Figure 5 describes the dynamics of our language. In our language, tasks execute under a heap H and global memory μ, using a small step transition judgment. The step relation uses the heap H for allocations, which can be seen in ALLOC, where a storable s steps to a new fresh location ℓ, and ℓ is mapped to s in μ and added to H. The evaluation rules for pairs and functions are standard semantics for call-by-value dynamics. These rules evaluate pairs and functions down to locations, and lookup values in μ to perform the operation. The rule FORK describes how an unactivated parallel tuple expression is activated by promoting each expression within it to a task with an empty heap. The rule JOIN takes the result of a parallel execution and steps to a pair of the results, while joining the heaps of the two children with the parent. The rules for parallel execution step the inner tasks non-deterministically, allowing for the tasks within an activate tuple to execute logically in parallel. Lastly, we include an inductive definition to extract all mentioned locations within a task. Aside from the rules relating to parallel constructs, the remaining dynamics are very standard for call-by-value, left-to-right evaluation dynamics.

3.4 Type Safety

We establish that well-typed terms do not become stuck by establishing progress and preservation. The proofs are straightforward, and proceed by rule induction.

Lemma 1 (Progress)**.** *For any* $\cdot\,; \Sigma\,; \Delta \vdash H{\cdot}T : \tau$ *and* $\mu : \Sigma$*, either* T *is a location or* $\mu\,; H\,; T \longmapsto \mu'\,; H'\,; T'$*.*

Proof. By induction on the derivation of $\cdot\,; \Sigma\,; \Delta \vdash H{\cdot}T : \tau$. □

Lemma 2 (Preservation)**.** *For any* $\cdot;\Sigma;\Delta \vdash H{\cdot}T : \tau$ *and* $\mu : \Sigma$*, if* $\mu;H;T \longmapsto \mu';H';T'$*, then* $\cdot;\Sigma';\Delta' \vdash H'{\cdot}T' : \tau$ *and* $\mu' : \Sigma'$ *where* H' *and* Σ' *are extensions of* H *and* Σ*, respectively.*

Proof. By induction on the derivation of $\mu\,; H\,; T \longmapsto \mu'\,; H'\,; T'$. □

Task Execution $\boxed{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}$

$$\frac{\ell \notin \mathrm{dom}(\mu)}{\mu \,;\, H \,;\, s \longmapsto \mu[\ell \hookrightarrow s] \,;\, (H \uplus \{\ell\}) \,;\, \ell}\ \textsc{Alloc} \qquad \frac{\mu(\ell_1) = \mathsf{fun}\ f\ x\ \mathsf{is}\ e}{\mu \,;\, H \,;\, (\ell_1\ \ell_2) \longmapsto \mu \,;\, H \,;\, [\ell_1, \ell_2 \,/\, f, x]e}\ \textsc{App}$$

$$\frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, (T\ e) \longmapsto \mu' \,;\, H' \,;\, (T'\ e)}\ \textsc{AppSL} \qquad \frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, (\ell\ T) \longmapsto \mu' \,;\, H' \,;\, (\ell\ T')}\ \textsc{AppSR}$$

$$\frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, \langle T, e\rangle \longmapsto \mu' \,;\, H' \,;\, \langle T', e\rangle}\ \textsc{PairSL} \qquad \frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, \langle \ell, T\rangle \longmapsto \mu' \,;\, H' \,;\, \langle \ell, T'\rangle}\ \textsc{PairSR}$$

$$\frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, (\mathsf{fst}\ T) \longmapsto \mu' \,;\, H' \,;\, (\mathsf{fst}\ T')}\ \textsc{FstS} \qquad \frac{\mu(\ell) = \langle \ell_1, \ell_2\rangle}{\mu \,;\, H \,;\, (\mathsf{fst}\ \ell) \longmapsto \mu \,;\, H \,;\, \ell_1}\ \textsc{Fst}$$

$$\frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, (\mathsf{snd}\ T) \longmapsto \mu' \,;\, H' \,;\, (\mathsf{snd}\ T')}\ \textsc{SndS} \qquad \frac{\mu(\ell) = \langle \ell_1, \ell_2\rangle}{\mu \,;\, H \,;\, (\mathsf{snd}\ \ell) \longmapsto \mu \,;\, H \,;\, \ell_2}\ \textsc{Snd}$$

$$\frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, (\mathsf{ref}\ T) \longmapsto \mu' \,;\, H' \,;\, (\mathsf{ref}\ T')}\ \textsc{RefS} \qquad \frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, (!\,T) \longmapsto \mu' \,;\, H' \,;\, (!\,T')}\ \textsc{BangS} \qquad \frac{\mu(\ell) = \mathsf{ref}\ \ell'}{\mu \,;\, H \,;\, (!\,\ell) \longmapsto \mu \,;\, H \,;\, \ell'}\ \textsc{Bang}$$

$$\frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, (T := e) \longmapsto \mu' \,;\, H' \,;\, (T' := e)}\ \textsc{UpdSL} \qquad \frac{\mu \,;\, H \,;\, T \longmapsto \mu' \,;\, H' \,;\, T'}{\mu \,;\, H \,;\, (\ell := T) \longmapsto \mu' \,;\, H' \,;\, (\ell := T')}\ \textsc{UpdSR}$$

$$\frac{}{\mu[\ell_1 \hookrightarrow s] \,;\, H \,;\, (\ell_1 := \ell_2) \longmapsto \mu[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2] \,;\, H \,;\, \ell_2}\ \textsc{Upd}$$

$$\frac{}{\mu \,;\, H \,;\, \langle e_1 \parallel e_2\rangle \longmapsto \mu \,;\, H \,;\, \llparenthesis\, \emptyset \cdot e_1 \parallel \emptyset \cdot e_2 \,\rrparenthesis}\ \textsc{Fork} \qquad \frac{}{\mu \,;\, H \,;\, \llparenthesis\, H_1 \cdot \ell_1 \parallel H_2 \cdot \ell_2 \,\rrparenthesis \longmapsto \mu \,;\, (H \uplus H_1 \uplus H_2) \,;\, \langle \ell_1, \ell_2\rangle}\ \textsc{Join}$$

$$\frac{\mu \,;\, H_1 \,;\, T_1 \longmapsto \mu' \,;\, H_1' \,;\, T_1'}{\mu \,;\, H \,;\, \llparenthesis\, H_1 \cdot T_1 \parallel H_2 \cdot T_2 \,\rrparenthesis \longmapsto \mu' \,;\, H \,;\, \llparenthesis\, H_1' \cdot T_1' \parallel H_2 \cdot T_2 \,\rrparenthesis}\ \textsc{ParL}$$

$$\frac{\mu \,;\, H_2 \,;\, T_2 \longmapsto \mu' \,;\, H_2' \,;\, T_2'}{\mu \,;\, H \,;\, \llparenthesis\, H_1 \cdot T_1 \parallel H_2 \cdot T_2 \,\rrparenthesis \longmapsto \mu' \,;\, H \,;\, \llparenthesis\, H_1 \cdot T_1 \parallel H_2' \cdot T_2' \,\rrparenthesis}\ \textsc{ParR}$$

Figure 5: Dynamics of our language

$$\begin{aligned}
\mathsf{locs}(\ell) &\triangleq \{\ell\} \\
\mathsf{locs}(n) &\triangleq \emptyset \\
\mathsf{locs}(\mathsf{fun}\ f\ x\ \mathsf{is}\ e) \triangleq \mathsf{locs}(\mathsf{fst}\ e) \triangleq \mathsf{locs}(\mathsf{snd}\ e) \triangleq \mathsf{locs}(\mathsf{ref}\ e) \triangleq \mathsf{locs}(!\,e) &\triangleq \mathsf{locs}(e) \\
\mathsf{locs}(e_1\ e_2) \triangleq \mathsf{locs}(\langle e_1, e_2\rangle) \triangleq \mathsf{locs}(e_1 := e_2) \triangleq \mathsf{locs}(\langle e_1 \parallel e_2\rangle) &\triangleq \mathsf{locs}(e_1) \cup \mathsf{locs}(e_2)
\end{aligned}$$

Figure 6: Mentioned locations (expression roots)

$\boxed{A \vdash_\mu H \cdot T \; \mathsf{de}}$

$$\frac{\mathsf{locs}(e) \subseteq A \uplus H \qquad \forall \ell \in H.\, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H}{A \vdash_\mu H \cdot e \; \mathsf{de}}$$

$$\frac{A \uplus H \vdash_\mu H_1 \cdot T_1 \; \mathsf{de} \qquad A \uplus H \vdash_\mu H_2 \cdot T_2 \; \mathsf{de} \qquad \forall \ell \in H.\, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus H_1 \uplus H_2 \uplus D(T_1) \uplus D(T_2)}{A \vdash_\mu H \cdot \llparenthesis H_1 \cdot T_1 \parallel H_2 \cdot T_2 \rrparenthesis \; \mathsf{de}}$$

$$\frac{A \vdash_\mu H \cdot T \; \mathsf{de} \qquad \mathsf{locs}(e) \subseteq A \uplus H}{A \vdash_\mu H \cdot T\, e \; \mathsf{de}} \qquad \frac{\ell \in A \uplus H \qquad A \vdash_\mu H \cdot T \; \mathsf{de}}{A \vdash_\mu H \cdot \ell\, T \; \mathsf{de}} \qquad \frac{A \vdash_\mu H \cdot T \; \mathsf{de} \qquad \mathsf{locs}(e) \subseteq A \uplus H}{A \vdash_\mu H \cdot \langle T, e \rangle \; \mathsf{de}}$$

$$\frac{\ell \in A \uplus H \qquad A \vdash_\mu H \cdot T \; \mathsf{de}}{A \vdash_\mu H \cdot \langle \ell, T \rangle \; \mathsf{de}} \qquad \frac{A \vdash_\mu H \cdot T \; \mathsf{de}}{A \vdash_\mu H \cdot \mathsf{fst}\, T \; \mathsf{de}} \qquad \frac{A \vdash_\mu H \cdot T \; \mathsf{de}}{A \vdash_\mu H \cdot \mathsf{snd}\, T \; \mathsf{de}} \qquad \frac{A \vdash_\mu H \cdot T \; \mathsf{de}}{A \vdash_\mu H \cdot \mathsf{ref}\, T \; \mathsf{de}}$$

$$\frac{A \vdash_\mu H \cdot T \; \mathsf{de}}{A \vdash_\mu H \cdot !\, T \; \mathsf{de}} \qquad \frac{A \vdash_\mu H \cdot T \; \mathsf{de} \qquad \mathsf{locs}(e) \subseteq A \uplus H}{A \vdash_\mu H \cdot T := e \; \mathsf{de}} \qquad \frac{\ell \in A \uplus H \qquad A \vdash_\mu H \cdot T \; \mathsf{de}}{A \vdash_\mu H \cdot \ell := T \; \mathsf{de}}$$

Figure 7: Definition of Disentanglement. Interesting rules highlighted.

4 Disentanglement

We can now formally define disentanglement as an inductive property of tasks in our language. Intuitively, disentanglement is obliviousness to all allocations performed by concurrent tasks, or that the pointers in the task tree can only point up or down. Thus, for a task T to be disentangled, all locations that the task has access to need to be within the heaps of the ancestors or descendants of T. Figure 12 defines the inductive judgment $A \vdash_\mu H \cdot T$ de, which is interpreted as "the task T is disentangled under memory μ with respect to heap H and ancestors A". In this judgment, A includes the heaps of all ancestors that T has, and is built up recursively. The interesting rules in this definition are the rules for expressions and activated parallel tuples. For an expression e to be disentangled, all the locations within e must be in either the set of ancestors, or the heap of the expression. Additionally, all locations in the heap of e must point to storables that satisfy the same property above. For the activated parallel tuple, a similar condition must be upheld: the components of the tuple must be disentangled, but now have access to the tuple's heap in ancestry. Additionally, all locations in the tuples heap must point to locations within the ancestry, or the heaps within the components of the tuple. This inductive definition formalizes the intuitive notion of pointers only pointing up or down, as all mentioned locations must be included in either the ancestors or descendants of a task. Using this definition of disentanglement, we can now examine formally what programs are disentangled, and how other properties interact with disentanglement.

5 Determinacy Race Freedom

Given the focus of this thesis on disentangled programs, a natural question is "what programs are actually disentangled?". In this section, we present one of the main contributions of this thesis by showing that all determinacy race free programs are disentangled.

Tasks with Computation Graphs

$$\begin{array}{rrcl}
\textit{Tasks} & T & ::= & \ldots \mid \llparenthesis\, G_1 \cdot H_1 \cdot T_1 \parallel G_2 \cdot H_2 \cdot T_2 \,\rrparenthesis \\
\textit{Computation Graphs} & G & ::= & \bullet \mid \alpha \mid G \oplus G \mid G \otimes G \\
\textit{Actions} & \alpha & ::= & \mathsf{none} \mid \mathsf{read}\ \ell \mid \mathsf{write}\ \ell
\end{array}$$

Figure 8: Definition of Computation Graphs

Computation Graph Writes, Extraction, and Erasure

$$\begin{array}{rcc}
\mathsf{W}(\bullet) \triangleq \emptyset & \mathsf{GT}(G, e) \triangleq G & \llbracket e \rrbracket \triangleq e \\
\mathsf{W}(\mathsf{none}) \triangleq \emptyset & \mathsf{GT}(G, T\ e) \triangleq \mathsf{GT}(G, T) & \llbracket T\ e \rrbracket \triangleq \llbracket T \rrbracket\ e \\
\mathsf{W}(\mathsf{read}\ \ell) \triangleq \emptyset & \mathsf{GT}(G, \ell\ T) \triangleq \mathsf{GT}(G, T) & \llbracket \ell\ T \rrbracket \triangleq \ell\ \llbracket T \rrbracket \\
\mathsf{W}(\mathsf{write}\ \ell) \triangleq \{\ell\} & \cdots & \cdots \\
\mathsf{W}(G_1 \oplus G_2) \triangleq \mathsf{W}(G_1) \cup \mathsf{W}(G_2) & \mathsf{GT}(G, \llparenthesis\, G_1 \cdot H_1 \cdot T_1 \parallel G_2 \cdot H_2 \cdot T_2 \,\rrparenthesis) & \llbracket \llparenthesis\, G_1 \cdot H_1 \cdot T_1 \parallel G_2 \cdot H_2 \cdot T_2 \,\rrparenthesis \rrbracket \\
\mathsf{W}(G_1 \otimes G_2) \triangleq \mathsf{W}(G_1) \cup \mathsf{W}(G_2) & \triangleq G \oplus (\mathsf{GT}(G_1, T_1) \otimes \mathsf{GT}(G_2, T_2)) & \triangleq \llparenthesis\, H_1 \cdot \llbracket T_1 \rrbracket \parallel H_2 \cdot \llbracket T_2 \rrbracket \,\rrparenthesis
\end{array}$$

Figure 9: Operations on Computation

5.1 Background

Determinacy race freedom is an important correctness property of parallel programs, and is a well studied topic in parallel computing literature [20, 21, 22, 39]. There are many tools to detect whether a program has determinacy races [20, 21, 39], and a large number of parallel algorithms can be implemented in a determinacy race free manner. A determinacy race occurs when two parallel tasks perform an operation at a location ℓ, where at least one of the operations is a write. A program that has no determinacy races is *determinacy race free*. Determinacy race freedom is not equivalent to disentanglement, but is actually more restrictive, as disentanglement can allow for write-write races, while such operations are considered determinacy races.

5.2 Instrumentation

Determinacy race freedom has traditionally been analyzed in terms of program execution DAGs, instead of through programming languages. We show how to define determinacy race freedom as a language level property, and do so by instrumenting our existing language with computation graphs. In Figure 8 we describe the structure of computation graphs. In addition to heaps in our task structure, we now also include computation graphs, which represent the graphs that each task is building while executing. Computation graphs are inductively defined as either empty, an action, or the sequential or parallel composition of two computation graphs. Actions are either empty, or reading a particular location, or writing to a particular location. Figure 9 details inductive definitions of operations on graph and tasks, including extracting all writes from a graph, constructing the full graph from a graph and a task, and erasing graphs from a task. Lastly, Figure 10 shows selected rules for computation graph instrumented execution of our language. The stepping judgment is very similar to that of the non-instrumented step judgment, except now we annotate onto the computation graph the action that was taken by the step. For example, when a location ℓ is read, then $\mathsf{read}\ \ell$ is sequentially composed onto the computation graph. The goal of instrumentation with computation graphs is to have a record of the actions that the program has taken. Using this record of actions,

Instrumented Execution $\boxed{\mu \,;\, G \,;\, H \,;\, T \longmapsto \mu' \,;\, G' \,;\, H' \,;\, T'}$

$$\frac{\ell \notin \mathrm{dom}(\mu)}{\mu \,;\, G \,;\, H \,;\, s \longmapsto \mu[\ell \hookrightarrow s] \,;\, (G \oplus \mathsf{write}\ \ell) \,;\, (H \uplus \{\ell\}) \,;\, \ell}\ \textsc{Alloc-G}$$

$$\frac{\mu(\ell) = \langle \ell_1, \ell_2 \rangle}{\mu \,;\, G \,;\, H \,;\, (\mathsf{fst}\ \ell) \longmapsto \mu \,;\, (G \oplus \mathsf{read}\ \ell) \,;\, H \,;\, \ell_1}\ \textsc{Fst-G} \qquad \frac{\mu(\ell) = \mathsf{ref}\ \ell'}{\mu \,;\, G \,;\, H \,;\, (!\,\ell) \longmapsto \mu \,;\, (G \oplus \mathsf{read}\ \ell) \,;\, H \,;\, \ell'}\ \textsc{Bang-G}$$

$$\frac{}{\mu[\ell_1 \hookrightarrow s] \,;\, G \,;\, H \,;\, (\ell_1 := \ell_2) \longmapsto \mu[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2] \,;\, (G \oplus \mathsf{write}\ \ell_1) \,;\, H \,;\, \ell_2}\ \textsc{Upd-G}$$

$$\frac{}{\mu \,;\, G \,;\, H \,;\, \langle e_1 \parallel e_2 \rangle \longmapsto \mu \,;\, (G \oplus \mathsf{none}) \,;\, H \,;\, (\!|\, \bullet \cdot \emptyset \cdot e_1 \parallel \bullet \cdot \emptyset \cdot e_2 \,|\!)}\ \textsc{Fork-G}$$

$$\frac{}{\mu \,;\, G \,;\, H \,;\, (\!|\, G_1 \cdot H_1 \cdot \ell_1 \parallel G_2 \cdot H_2 \cdot \ell_2 \,|\!) \longmapsto \mu \,;\, G \oplus (G_1 \otimes G_2) \,;\, H \uplus H_1 \uplus H_2 \,;\, \langle \ell_1, \ell_2 \rangle}\ \textsc{Join-G}$$

$$\frac{\mu \,;\, G_1 \,;\, H_1 \,;\, T_1 \longmapsto \mu' \,;\, G_1' \,;\, H_1' \,;\, T_1'}{\mu \,;\, H \,;\, (\!|\, G_1 \cdot H_1 \cdot T_1 \parallel G_2 \cdot H_2 \cdot T_2 \,|\!) \longmapsto \mu' \,;\, H \,;\, (\!|\, G_1' \cdot H_1' \cdot T_1' \parallel G_2 \cdot H_2 \cdot T_2 \,|\!)}\ \textsc{ParL-G}$$

Figure 10: Operational semantics instrumented with computation graphs (selected rules).

Determinacy Race Freedom $\boxed{W \vdash G\ \mathsf{drf}}$

$$\frac{}{W \vdash \bullet\ \mathsf{drf}} \qquad \frac{}{W \vdash \mathsf{none}\ \mathsf{drf}} \qquad \frac{\ell \notin W}{W \vdash (\mathsf{write}\ \ell)\ \mathsf{drf}} \qquad \frac{\ell \notin W}{W \vdash (\mathsf{read}\ \ell)\ \mathsf{drf}}$$

$$\frac{W \vdash G_1\ \mathsf{drf} \qquad W \vdash G_2\ \mathsf{drf}}{W \vdash G_1 \oplus G_2\ \mathsf{drf}} \qquad \frac{W \cup \mathsf{W}(G_2) \vdash G_1\ \mathsf{drf} \qquad W \cup \mathsf{W}(G_1) \vdash G_2\ \mathsf{drf}}{W \vdash G_1 \otimes G_2\ \mathsf{drf}}$$

Figure 11: Definition of determinacy race freedom.

we can define properties on the actions that the program has taken in the past, and use those to reason about actions the program will take in the future. It is clear to see that if a task can take a step in the instrumented dynamics, then it can take a step in the standard dynamics.

5.3 Determinacy Race Freedom

Given computation a graph G, we can define what it means for G to be determinacy race free, abbreviated to DRF. As in literature, a program is DRF if there does not exist an e_1 and e_2 such that e_1 runs concurrently with e_2, and e_1, e_2 operate on a location where at least one action is a write. To encode this inductively, we define the judgment $W \vdash G\ \mathsf{drf}$, where W is a set of all writes performed concurrently to G. Under this judgment, a single node is DRF if the location it reads or writes to is not in W. For two graphs composed in parallel, the entire graph is DRF if each graph is DRF w.r.t the incoming set of concurrent writes along with all the writes that the other graph in the composition performs. This definition is consistent as with in literature, because if there was an e_1 and e_2 running

$$\boxed{W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde}}$$

$$\frac{\begin{array}{c}\mathsf{locs}(e) \subseteq A \uplus H \\ W \vdash G \ \mathsf{drf} \qquad \forall \ell \in A.\ (\ell \notin W) \Rightarrow \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus D(\llbracket T \rrbracket) \qquad \forall \ell \in H.\, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H\end{array}}{W \,;\, A \vdash_\mu G \cdot H \cdot e \ \mathsf{drfde}}\ \textsc{DRFDE-e}$$

$$\frac{\begin{array}{c}W \vdash G \ \mathsf{drf} \\ W \cup \mathsf{W}(\mathsf{GT}(G_2, T_2)) \,;\, A \uplus H \vdash_\mu G_1 \cdot H_1 \cdot T_1 \ \mathsf{drfde} \qquad W \cup \mathsf{W}(\mathsf{GT}(G_1, T_1)) \,;\, A \uplus H \vdash_\mu G_2 \cdot H_2 \cdot T_2 \ \mathsf{drfde} \\ \forall \ell \in A.\ (\ell \notin W) \Rightarrow \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus H_1 \uplus H_2 \uplus D(\llbracket T_1 \rrbracket) \uplus D(\llbracket T_2 \rrbracket) \\ \forall \ell \in H.\, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus H_1 \uplus H_2 \uplus D(\llbracket T_1 \rrbracket) \uplus D(\llbracket T_2 \rrbracket)\end{array}}{W \,;\, A \vdash_\mu G \cdot H \cdot (\!| G_1 \cdot H_1 \cdot T_1 \parallel G_2 \cdot H_2 \cdot T_2 |\!) \ \mathsf{drfde}}\ \textsc{DRFDE-par}$$

$$\frac{W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde} \qquad \mathsf{locs}(e) \subseteq A \uplus H}{W \,;\, A \vdash_\mu G \cdot H \cdot T\ e \ \mathsf{drfde}}\ \textsc{DRFDE-TApp} \qquad \frac{\ell \in A \uplus H \qquad W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde}}{W \,;\, A \vdash_\mu G \cdot H \cdot \ell\ T \ \mathsf{drfde}}\ \textsc{DRFDE-}\ell\textsc{app}$$

$$\frac{W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde} \qquad \mathsf{locs}(e) \subseteq A \uplus H}{W \,;\, A \vdash_\mu G \cdot H \cdot \langle T, e \rangle \ \mathsf{drfde}}\ \textsc{DRFDE-TPair} \qquad \frac{\ell \in A \uplus H \qquad W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde}}{W \,;\, A \vdash_\mu G \cdot H \cdot \langle \ell, T \rangle \ \mathsf{drfde}}\ \textsc{DRFDE-}\ell\textsc{pair}$$

$$\frac{W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde}}{W \,;\, A \vdash_\mu G \cdot H \cdot \mathsf{fst}\ T \ \mathsf{drfde}}\ \textsc{DRFDE-fst} \qquad \frac{W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde}}{W \,;\, A \vdash_\mu G \cdot H \cdot \mathsf{snd}\ T \ \mathsf{drfde}}\ \textsc{DRFDE-snd}$$

$$\frac{W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde}}{W \,;\, A \vdash_\mu G \cdot H \cdot \mathsf{ref}\ T \ \mathsf{drfde}}\ \textsc{DRFDE-ref} \qquad \frac{W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde}}{W \,;\, A \vdash_\mu G \cdot H \cdot !\, T \ \mathsf{drfde}}\ \textsc{DRFDE-bang}$$

$$\frac{W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde} \qquad \mathsf{locs}(e) \subseteq A \uplus H}{W \,;\, A \vdash_\mu G \cdot H \cdot T := e \ \mathsf{drfde}}\ \textsc{DRFDE-TUpd} \qquad \frac{\ell \in A \uplus H \qquad W \,;\, A \vdash_\mu G \cdot H \cdot T \ \mathsf{drfde}}{W \,;\, A \vdash_\mu G \cdot H \cdot \ell := T \ \mathsf{drfde}}\ \textsc{DRFDE-}\ell\textsc{upd}$$

Figure 12: Strengthening of disentanglement with DRF invariant. Interesting rules highlighted.

concurrently, they would a node within a parallel composition of two computation graphs. If e_1 performed a write to a location ℓ and e_2 operated on that location, then ℓ would be within the concurrent write set for e_2, and determinacy race freedom would not be derivable.

5.4 Strengthening Disentanglement

In order to show that determinacy race freedom of a program implies that it is disentangled, we provide a strengthened definition of disentanglement in Figure 12. In the judgment $W \,;\, A \vdash_\mu G \cdot H \cdot T$ drfde, we strengthen disentanglement with access to the concurrent write set and the computation graph. The rules are similar to the standard disentanglement definition, but we add on new conditions in the expression and activated parallel tuple cases. In the expression case, we check that all locations in the expression are within the ancestors and heap, and that all locations in the heap point to locations within the ancestors and heap as well. In the strengthened judgment, we now require that the computation graph G is DRF, and that all locations in the ancestors that are not in W point to locations within the ancestors and heap. The intuition behind this strengthening comes from the fact observation that the only way to obtain a pointer outside of the ancestors and heap would have to come from a location that a concurrent task wrote a local value into. However, determinacy race freedom ensures that once a location has been written to by a task, all concurrent tasks are prohibited from touching it. Therefore, we can maintain that all locations that are permitted to touch point to locations within the ancestors and heap. The judgment for the active parallel tuple is similar to that of the standard definition, but with the extra checks that the case for expression performs.

5.5 Determinacy Race Freedom Implies Disentanglement

Now that we have formally defined disentanglement and determinacy race freedom, we can show that all programs that are determinacy race free are disentangled. Throughout, we implicitly only consider judgments $W \,;\, A \vdash_{\mu} G \cdot H \cdot T \; \mathsf{drfde}$ where $A, H \subseteq \mathrm{dom}(\mu)$.

To begin, we first show that the strengthened definition of disentanglement on an instrumented task implies that the task is disentangled in the standard semantics.

Lemma 3. *If* $W \,;\, A \vdash_{\mu} G \cdot H \cdot T \; \mathsf{drfde}$ *then* $W \vdash \mathsf{GT}(G, T) \; \mathsf{drf}$ *and* $A \vdash_{\mu} H \cdot [\![T]\!] \; \mathsf{de}$.

The main step of the proof is to show that the step relation maintains disentanglement, determinacy race freedom is maintained as well. Specifically, we will show that if a task T is disentangled and determinacy race free, and steps to a task T' that is determinacy race free, then it must be disentangled. To do this, we need some auxiliary lemmas for parts of the proof.

Lemma 4. *If* $W \,;\, A \vdash_{\mu} G \cdot H \cdot T \; \mathsf{drfde}$ *and* $W' \vdash \mathsf{W}(\mathsf{GT}(G, T)) \; \mathsf{drf}$ *then* $W' \,;\, A \vdash_{\mu'} G \cdot H \cdot T \; \mathsf{drfde}$, *where* $W' = W \cup \mathit{write}\; \ell$, *and* μ, μ' *agree at all locations* $\ell' \neq \ell$, *and* $\ell \notin H \uplus D([\![T]\!])$.

Proof. Proof is in Appendix 14.1. □

This lemma tells us that if a task is DRF and disentangled w.r.t to a particular write set, and if it is DRF w.r.t to write lets that contain locations outside of its local allocations, then it is also disentangled. Next, we need a lemma that tells us something about locations that get written to.

Lemma 5. *For any* $\mu : \Sigma$ *and* $\cdot\,; \Sigma ; \Delta \vdash H \cdot [\![T]\!] : \tau$ *and* $W ; A \vdash_{\mu} G \cdot H \cdot T \; \mathsf{drfde}$ *and* $\mu ; G ; H ; T \longmapsto \mu' ; G' ; H' ; T'$, *if* $\mathsf{W}(\mathsf{GT}(G', T')) = \mathsf{W}(\mathsf{GT}(G, T)) \cup \mathit{write}\; \ell$, *then* $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus D([\![T]\!])$ *and* $\ell \in A \uplus H \uplus D([\![T]\!])$.

Proof. Proof is in Appendix 14.2. □

In particular, if a task is disentangled and DRF and performs a write to a location ℓ, then ℓ is within the heaps of ancestors or descendants of the task. Additionally, the locations that ℓ now points to are also all within the heaps of ancestors and descendants of the tasks. This lemma helps us argue for disentanglement when writes occur.

Using these auxiliary lemmas, we have enough setup to prove the core lemma: if a task is disentangled and DRF, and takes a step to a task that is DRF, then the resulting task is disentangled as well. The intuition behind this proof is that once a concurrent task writes to a location, we no longer need to store any information about that location.

Lemma 6. *For any* $\mu : \Sigma$ *and* $\cdot\,; \Sigma ; \Delta \vdash H \cdot [\![T]\!] : \tau$ *and* $W ; A \vdash_{\mu} G \cdot H \cdot T \; \mathsf{drfde}$ *and* $\mu ; G ; H ; T \longmapsto \mu' ; G' ; H' ; T'$, *if* $W \vdash \mathsf{GT}(G', T') \; \mathsf{drf}$ *then* $W \,;\, A \vdash_{\mu'} G' \cdot H' \cdot T' \; \mathsf{drfde}$.

Proof. Proof is in Appendix 14.3. □

Using Lemma 6, we can prove the main theorem, which states that if a task T takes a number of steps to another task T', and if T' is DRF, then T' is disentangled.

Theorem 1 (DRF ⇒ DE)**.** *For any* $\cdot\,;\cdot \vdash e : \tau$ *and* $\emptyset ; \bullet ; \emptyset ; e \longmapsto^{*} \mu ; G ; H ; T$*, if* $\emptyset \vdash \mathsf{GT}(G, T)\ \mathsf{drf}$*, then* $\emptyset \vdash_{\mu} H \cdot [\![T]\!]\ \mathsf{de}$.

Proof. By induction on the derivation of $\emptyset ; \bullet ; \emptyset ; e \longmapsto^{*} \mu ; G ; H ; T$. Initially we have $\emptyset ; \emptyset \vdash_{\emptyset} \bullet\cdot\emptyset\cdot e\ \mathsf{drfde}$, and by Lemmas 2 and 6 we have $\emptyset ; \emptyset \vdash_{\mu} G\cdot H\cdot T\ \mathsf{drfde}$. By Lemma 3, we have $\emptyset \vdash_{\mu} H\cdot[\![T]\!]\ \mathsf{de}$. □

6 Hierarchical Memory Management

Now that we have shown that an important class of parallel programs, determinacy race free programs, are disentangled, we describe a memory management scheme that allows for fast and efficient execution of disentangled, nested parallel programs. We build on the techniques presented in prior work of *hierarchical memory management* [26, 35] to efficiently manage memory for disentangled programs. We first describe the existing infrastructure before moving onto our contributions in concurrent garbage collection strategies for disentangled programs.

6.1 Hierarchical Heaps

We give each parallel task its own logical *heap* where it performs all local allocations. When a task creates two children, the original task's heap is suspended, and the two children are given fresh heaps to allocate into. The children's heaps grow as they allocate memory. When a task finishes, its heap is logically joined into the heaps of its parent and sibling task, and the parent task continues. While a task is executing, the heaps of its parent and its ancestors are *suspended* until it completes. These heaps and tasks are organized into a tree, called the *hierarchy*, which mirrors the nesting structure of the tasks. Using this structure, we can classify different types of pointers that exist within the hierarchy at runtime. Pointers in the hierarchy can either be a

- *up-pointer*, that points from a task's heap into an ancestor task's heap.
- *down-pointer*, that points from a task's heap into a descendant task's heap.
- *cross-pointers*, that points from a task's heap into another task's heap that is not an ancestor or descendant of the original task.

Cross-pointers point from a task into a different subtree, causing *entanglement*. As our memory management techniques consider only programs that are disentangled, we are guaranteed that there do not exist any cross-pointers in the hierarchy at runtime. As the definitions of disentanglement entail, at any point in time, all locations that are referenced from a task access only locations that ancestors or descendants hold, meaning that all points are either up-pointers or down-pointers.

6.2 Hierarchical Garbage Collection

Preliminaries

Within a memory managed program, the *mutator* performs computation and allocates objects, and the *collector* attempts to reclaim objects that the mutator can no longer access. To do this, the collector determines the set of objects that are reachable from the program, called the *live* set. Once the live set of objects has been calculated, the garbage collector can reclaim the space used by all remaining objects, as they can no longer be accessed by the program. To determine all objects that are reachable by the mutator, the collector obtains a set of *roots* from the mutator, which consist of a set of objects that the mutator explicitly holds at any point in the program. Once the collector obtains a set of roots from the mutator, all objects that can be reached by traversing the memory graph from the roots. In practice, the root set is usually the pointer values within the registers and stack of the mutator. In order to perform correct garbage collection, the collector must obtain all the roots that could reach a particular region in the heap that it wants to collect, otherwise it might mistakenly mark objects as live that are indeed reachable. During the process of garbage collection, the collector may want to move objects between heaps or rename objects through copying collection. In this case, the collector must be able to ensure that a mutator cannot access data that is being moved and renamed while collection is occurring.

Hierarchical Memory Management Without Mutation

If we limit the scope of our programs to purely functional codes, the only pointers that can be contained within the hierarchy are up-pointers [35]. This limitation enables a conceptually simple garbage collection policy: perform collections on disjoint subtrees in parallel. Because all pointers are up-pointers, the only references that an object can have are those from descendants, or the roots. This implies that for disjoint subtrees T_1 and T_2, there exist no pointers from T_1 into T_2, and vice versa, and no pointers into T_1 from an ancestor. This guarantee allows for two different types of collection: *local collection* and *non-local collection*.

A local collection performs a collection only on the task currently executing on a worker. To perform a local collection, the collector can stop the mutator and obtain a set of roots, perform a local copying collection [16] on the task currently executing on the worker. This is a safe operation, as there can be no other tasks that have pointers into the task being collected, due to the guarantees of disentanglement. Local collections can occur completely independently and in parallel with other local collections. A non-local collection aims to collect an entire subtree, rather than just an individual leaf task. Collection proceeds in a similar way as with local collections, but all mutators executing within the subtree need to communicate a root set to the collector and stop executing while collection occurs. This is because all mutators executing within the subtree can have pointers that point within the subtree. Existing implementations in Raghunathan et al [35] and current work only implement local collections within the collector.

Supporting Mutation

Supporting general mutation is very difficult for parallel garbage collectors, resulting in the collector needing do some sort of communication with the mutator to efficiently collect shared objects. Most management systems existing within parallel functional languages moving shared data into a separate heap shared by all processors. Such copies can be expensive to perform [26]. Previous work had a weaker notion of disentanglement, where all pointers must be up-pointers [26]. In this previous work, when a write occurred that would create entanglement, the runtime would move the written object up in the hierarchy until the entanglement was resolved. This scheme, while correct, was very heavyweight and required a large amount of synchronization when mutating shared objects. Recent ongoing work as shown that mutation in a disentangled manner can be efficiently supported within the hierarchical memory management scheme.

Remembered Sets Because disentanglement permits down-pointers, the strategy for supporting disentangled mutation involves managing pointers into heaps that come from ancestors of a task. When performing a local or non-local collection on a set of heaps, the pointers to live data within the set can come from either the roots of the mutator, or from data residing within ancestor heaps. A correct but inefficient strategy to obtain these incoming pointers would be to inspect every ancestor to find down-pointers. To make this process more efficient, we remember all down-pointers into a particular heap by assigning each heap a *remembered set*. The remembered set for a particular heap consists of all objects that might contain down pointers into that heap. Using the remembered sets, we can find all pointers that the collector must use as roots by taking the roots of all mutators within the target subtree, and all objects within the remembered sets of the heaps within the subtree.

Promotions While remembered sets are enough to guarantee correctness of the garbage collection with disentangled mutation, the efficiency can still be improved. Current work includes a *promotion* phase to garbage collection which eliminates down-pointers by moving objects upwards in the hierarchy, similar to [26]. The promotion phase identifies all objects that are referenced by objects above them in the hierarchy, and moves them upwards in the hierarchy. Specifically, if object x is referenced by an ancestor object y, then x and all objects reachable from x are promoted into the ancestor that contains y. This is motivated by efficiency reasons: x cannot be reclaimed until it can be been determined that y is garbage, which in general would require tracing the entire memory graph (if y is at the root of the hierarchy). Rather than letting x stay within the scope of repeated collections, where x is known to be live, it is promoted into a higher heap which is collected less often. This strategy is similar to those used in generational collection schemes [7, 32, 38], where old objects that persist through collections should be collected less often.

Collection Policies

Current work supports these strategies for garbage collection with mutation, but revolves around only performing local collections at the leaves of the hierarchy. To decide when to actually perform these local collections, current work uses a Cheney style copying collection scheme [16]. When the heap has grown past some capacity c, a collection triggers, and the heap is resized to capacity $\alpha \cdot S$, where S is the size of the objects survived. This strategy works well in practice, but does not consider how and when to efficiently perform non local collections. Because a task's heap is suspended out of the scope of local collections once it forks new child tasks, the inability to perform local collections leads to difficulties in collecting this suspended data in the hierarchy.

7 Concurrent Garbage Collection

7.1 Motivation

Our motivation for building a concurrent garbage collection scheme within the hierarchical memory management strategy comes from existing limitations in collection policy in current work. As mentioned previously, existing work revolves around performing efficient local collections at the leaves of the hierarchy, and performing as many of these collections in parallel as possible to avoid interference with the mutator. The key problem is the suspension of data performed when new tasks are created. When data is suspended in the hierarchy, it is out of the scope of collection in the existing collection strategy. This is efficient, because it allows individual local collections to

```
fun example N =
  let
    val S1 = Seq.tabulate (fn i ⇒ i) N
    val S2 = Seq.tabulate (fn i ⇒ i) N
    val S3 = Seq.tabulate (fn i ⇒ i) N
    val S4 = Seq.tabulate (fn i ⇒ i) N
    val S5 = Seq.tabulate (fn i ⇒ i) N
  in
    S5
  end
```

Figure 13: Example program that experiences space blowup in current work.

avoid communication with other mutators, as no shared data is being collected. However, because this suspended data is outside of scope for collection, there is potential for garbage to accumulate without getting a chance to be collected. For an explicit example, see Figure 13. In Figure 13, the array `S1` is allocated and filled in parallel, creating and merging tasks. Now, when `S2` is allocated and filled in parallel, new tasks are created, which results in the suspension of all previously allocated memory, including the array `S1`. Even though `S1` is garbage, it is not able to be collected and reused while it is dead. This process continues as all 5 arrays are allocated, as more data piles up in the hierarchy while local collections occur. This space blowup occurs in actually occurs using the existing compiler, while a standard garbage collection scheme would achieve a steady state, reusing the space that is becoming garbage. Figure 13 is a contrived example, but this pattern occurs in practice when composing aggregate operations, and very noticeable in iterative style algorithms like breadth first search on graphs.

At a first glance, a solution to this problem might be to simply allow collections at the root of the hierarchy when the hierarchy collapses back to a single node. This solution is not ideal for performance, because now all processors would be stalled waiting for the collection to end. To remedy this, a parallel garbage collection strategy could be used to utilize all processors [17, 23, 33], but even then, the original program is still blocked on waiting for the collection to finish before it can continue and generate more parallel work. A solution that gets the best of both worlds is the second major contribution of this thesis, a *concurrent garbage collection* method to perform non-local collections that doesn't interrupt the mutator while generating parallel work, and is able to collect dead data that is suspended out of the reach of local collections.

7.2 Overview

Our general strategy is to instead of using a hierarchy to manage the heaps of each task, we represent the the computation explicitly as a DAG called the *history*. Each node in the history holds onto a heap, similar to how each task in the hierarchy has a heap. We can employ the same local collection strategy at the leaves of the history as in current work. We use an non-moving garbage collection strategy to reclaim unused memory in the history concurrently with the mutator.

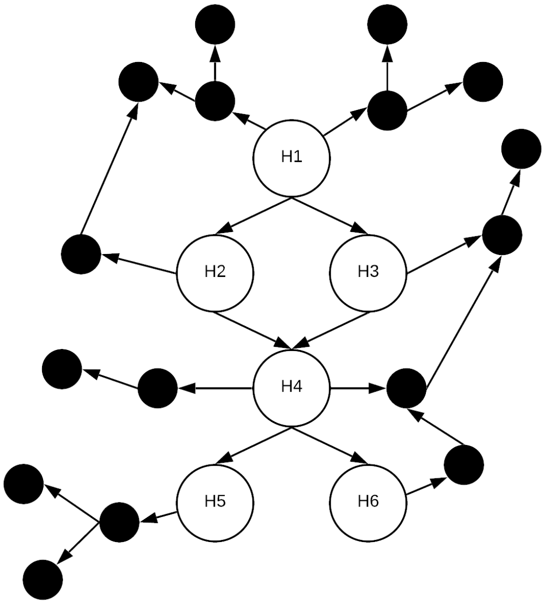

Figure 14: Diagram of the history. The heaps within the history are labeled. The black circles are objects, and the arrows between them are pointers.

7.3 The History

Figure 14 depicts an example of the DAG structure of the history. The DAG structure of the history mirrors the series-parallel DAG structure of the parallel program. Each node in the DAG has its own heap, similar to hierarchical memory management. The key difference between the history and the hierarchy is that once a task completes, it does not join its heap back into its parent. Instead, its heap survives as part of the history. Because our memory management system is built together with the scheduler, we can manage the structure of the history the scheduler creates and merges tasks. In Figure 15, we describe in psuedocode how the scheduler manages the history. Each task maintains a DAG node object, which holds information about its parents and children, along with a heap. When the mutator program submits two new tasks f and g to be run in parallel, we get the DAG D node of the current task, and allocate two new DAG nodes with D as the parent along with two new heaps. The two child tasks are then executed under their respective DAG nodes and heaps. Once the child tasks complete, a new DAG node D_n and heap are allocated for the join point, and the original parent task executes under D_n.

The history has *static* and *dynamic* components. The static part of the history is the completed series-parallel dag above where the mutators are executing. In Figure 14, nodes $H1, H2, H3$ and $H4$ are in the static component of the history. The remaining part of the history is the dynamic component, which is an "unfinished" series-parallel DAG. In Figure 14, nodes $H5$ and $H6$ are in the dynamic component. In the history, up-pointers and down-pointers are the same as in the hierarchy. However, disentanglement no longer guarantees the absence of cross-pointers in the history. In particular, mutation may cause cross-pointers to appear within the static region of the history. Disentanglement ensures that there are no cross-pointers in the dynamic portion of the history, which continues to allow for independent local collections at the leaves.

```
fun fork (f, g) =
  $D$ = DagNode(currentTask())
  $(D_f, D_g)$ = (newDagNode(parent = $D$), newDagNode(parent = $D$))
  $H_f, H_g$ = (newHeap($D_f$), newHeap($D_g$))
  $(r_f, r_g)$ = run t ∈ $f, g$ in $D_t$ and wait
  $D_n$ = newDagNode(parents = $D_f, D_g$)
  $H_n$ = newHeap($D_n$)
  run currentTask() in $D_n$
  return $(r_f, r_g)$
```

Figure 15: Psuedocode for managing the DAG structure of the history.

7.4 Collection Strategy

Our collection strategy focuses on collecting large portions of the history without having to overly synchronize with the mutators. In order to identify all live objects within a sub-graph of the history, the collector must obtain a root set of all pointers entering the sub-graph. This incoming pointer set are pointers from other nodes in the history, or the root sets of the mutators. As mentioned previously, if a sub-graph is a member of the static region of the history, the pointers that are incoming from other nodes in the history are not necessarily just from ancestors or descendants of the sub-graph. Additionally, trying to collect a sub-graph in the dynamic region of the history involves gathering a root set from each worker in the sub-graph, which requires a large amount of communication as the sub-graph gets large. The key observation for our collection strategy is that when the program joins back at the top level of nested parallelism, or when the old dynamic region becomes static, the only pointers that can point into the entire history are the root set of the mutator performing the join. Using this observation, when the scheduler is joining back at the top level, it can gather a root set from the only active mutator, and notify the concurrent collector to begin a collection at that DAG node to the rest of the history. This interruption is hidden by the cost of the scheduling action anyway, so there is no cost to this synchronization. After this, the concurrent collector can begin reclaiming data in the history, while the mutator continues to generate parallelism and perform work. We use a non-moving collector to perform the reclamation of data, so that no objects need to be renamed, which means that the collector doesn't need to pause the mutator while collecting.

Returning to the example in Figure 13, our concurrent collector would be able to reclaim the garbage arrays generated by the program after each one has been created. In a more realistic program that makes heavy use of aggregate operations, after each aggregate operation completes, the concurrent collector has a chance to begin reclaiming garbage generated by that operation without blocking the mutator at all.

Our garbage collection scheme is similar to a generational garbage collector, where data that survives multiple collections is collected less. In our case, data that becomes suspended in the history is most likely data that has survived local collections and therefore is considered in the scope of collections less.

7.5 Algorithm

We specify our concurrent collection algorithm in psuedocode in Figure 16. The `collect` method executes concurrently with the source program. Given a starting node in the DAG and a set of roots, which are computed when

```
fun collect (startNode, roots) =
  $D$ = entire DAG above startNode
  toVisit = roots
  marked = {}
  liveObjects = {}
  while toVisit is not empty:
    obj = pop (toVisit)
    if obj in marked:
      continue
    add(liveObjects, obj)
    add(marked, obj)
    for each field in obj:
      if obj.field in $D$:
        push(toVisit, obj.field)
  reclaim all objects in $D$ not in liveObjects
```

Figure 16: Psuedocode for the concurrent collector

a top-level join occurs, the collector first computes all DAG nodes in the history that are in scope of the collection. This set of DAG nodes is the set of all nodes in the history above the starting node. Once this has been computed, the collector then performs a traversal through the memory graph starting with the roots. To traverse the memory graph, the collector uses a depth first traversal, and ignore all objects that reside in DAG nodes outside of the scope of collection. Once the traversal completes, all live objects within the scope of collection have been identified, and the collector can reclaim all objects that have not been identified as live.

7.6 Concurrent Interaction

Since we now have a collector running possibly at the same time the mutators and local collections, there are some interactions between the mutators and local collections that must be specified.

Mutators

While the collector is trying to identify all live objects within the static region of the history, it is possible that a mutator performs a write into a field of an object that the collector is about to traverse, or already has traversed. We can argue that the writes mutators perform will not cause the collector to reclaim a live object.

In the first case, the mutator performs write into an object the collector has not yet considered. To be specific, consider an object x that is live in the history, with a field that points to object y. A mutator performs a write into x that overwrites the field to point to y', before the collector is able to consider x. We want to ensure that object y is not reclaimed if it is live. If y is only live because it has a reference from x, then is it dead once a mutator overwrites the reference in x. If y is live because of references from other objects, then y will be marked as live from those other objects, so it will not be reclaimed.

In the second case, if the mutator overwrites a field after the collector has considered it, then the collector has a chance of not reclaiming a dead object. This is still correct, but it is not exact, as the collector is not always able to reclaim all dead data, but the ability to proceed without synchronizing with the mutators.

Local Collection

The concurrent collector must also conflict with local collection. This is achieved by adapting the remembered set and promotion strategy used in current work with the history. For local collections, we need to remember the pointers into a particular heap that is currently being worked on by a mutator. We used remembered sets to remember all objects from the history that have pointers into a particular active heap, similar to how current work does. Once heap has been suspended, we no longer need the remembered set because it is tracking pointers into a local heap from the history, and is unnecessary once the local heap joins the history. We allow for promotions to occur during local collections in a similar way as in current work. Instead of promoting objects into the heap where the object that references them resides, we can just keep the objects in the mutator's DAG node out of the scope of local collection. Because the concurrent collector is collecting the static portions of the history and the local collections occur in the dynamic portions of the history, there is no more interaction between local collections and concurrent collection.

8 Implementation

We implemented our techniques by extending the MLton whole-program optimizing compiler [34]. The core of hierarchical memory management within MLton has been developed through existing [26, 35] and current work. Existing work uses a work-stealing scheduler [12] written in Standard ML, and a runtime system in C which implements the memory management system and garbage collection.

8.1 Scheduler

The scheduler is implemented based on the private deques scheduling algorithm [2]. The scheduler maps high level tasks onto a set of worker threads, which are implemented in the runtime with OS-level threads. Using the scheduler to manage the history allows us to coarsen the history to represent only tasks are actually run in parallel via being scheduled by a steal event in the scheduler.

8.2 Block Structured Heaps

The memory management system uses a block structured system, meaning it divides logically divides the virtual memory space into fixed size blocks. Each worker recycles these blocks in a local free list, and requests new blocks from the OS when the free list is not able to satisfy a request. We represent heaps as lists of *chunk*s, which are contiguous blocks in memory. Each chunk has a *chunk descriptor*, which holds metadata such as which DAG node the chunk is in.

8.3 Remembered Sets and Write Barriers

Each heap is equipped with a second list of chunks that is used to implement the remembered set. The remembered set for a particular heap remembers tuples (x, i, y), where $x[i]$ might hold a pointer from the history into y. To ensure that all writes that need to create remembered set entries do, we use a write barrier for the update $x[i] \leftarrow y$. The write barrier is a small piece of code executed before each write, as the compiler can use the type system of

Standard ML to know when writes to mutable data occur. The write barrier inserts a remembered set entry into the heap containing y if y is not yet part of the history (y is in a currently executing node), and x is suspended in the history. In the other cases, we don't need to add any remembered set entries because we don't need to maintain inter history pointers.

8.4 Concurrent Garbage Collection

The concurrent garbage collector is implemented as a special thread that is not under control of the scheduler. The concurrent collector interfaces with the scheduler through a condition variable, which the collector waits on to receive a root set and starting DAG node. To actually perform the collection, we use a standard mark-sweep style collection algorithm, which uses a depth first search to mark all objects reachable from the roots. Additionally, we mark all *chunks* that contain at least one live object during this traversal. After the memory graph traversal has been completed, the collector considers all chunks within the scope of collection, and reclaims all chunks that contain no live objects. This can lead to fragmentation, as one small live object can cause an entire chunk to not be reclaimed. However, since most data that joins the history has most likely survived local collections already and has been compacted through these collections, we hypothesize that these situations leading to extreme fragmentation don't occur often. Another topic of discussion is that our collector only uses a single thread to perform the traversal and reclamation operations. This limitation means that as the number of mutators increases, our collector simply cannot keep up with the rate of allocation on a single thread. The topic of parallelizing these traversals has been studied in literature [3], and is orthogonal to our work, though can be implemented in the logic of our collection algorithm.

9 Experiments

We evaluate the performance of our concurrent collection techniques compared to current work.

9.1 Benchmarks

We perform our comparison on a selection of benchmarks that are implemented in a disentangled manner.

- Fibonacci. Calculate the 43'rd Fibonacci number using the naive recursive formula.
- Tabulate. Given a function f : *int* $\to \alpha$, and a length n, produce a sequence where the i'th element is the result of computing $f(i)$.
- Filter. Given a sequence S and a function $f : \alpha \to$ *bool*, produce a new sequence S' containing all elements $x \in S$ such that $f(x)$.
- Reduce. Given a sequence S and a function $f : \alpha \times \alpha \to \alpha$, compute the reduction of the sequence under f.
- Sort. Give a sequence S and a function $f : \alpha \times \alpha \to$ *order*, produce a new sequence S' containing all elements of S sorted by the comparison operator f.
- IterTabulate. The contrived example from Figure 13, a tabulate operation repeated 10 times.
- Breadth First Search. Perform a breadth first traversal of an input graph G.
- Seam Carving. Find the cost of the minimum weighted seam in a 2 dimensional array [10].

Time Overhead on N Cores (mpl-cgc / mpl)							
Benchmark	1	2	4	8	16	32	64
tabulate	1.03x	1.04x	0.94x	1.03x	0.94x	0.96x	1.03x
filter	0.99x	0.96x	0.97x	0.87x	0.98x	0.99x	0.96x
fib	0.94x	0.94x	0.98x	0.89x	1.00x	0.95x	0.81x
sort	1.00x	0.96x	0.99x	1.01x	1.00x	1.07x	1.14x
reduce	0.96x	1.01x	0.92x	0.91x	0.98x	1.03x	0.94x
seam-carve	0.92x	0.77x	0.77x	0.95x	1.09x	1.23x	X
bfs	1.04x	1.08x	1.01x	1.01x	1.19x	1.25x	2.27x
itertabulate	0.99x	0.95x	1.10x	1.18x	1.22x	1.13x	1.35x

Figure 17: Time Overhead of Concurrent Collection on N cores. (Smaller is better. Values < 1 indicate mpl-cgc performs better.)

In our experiments, for the sequence operations, we use a sequence of size $n = 10^7$ of immutable strings generated by a hash function that produces random binary strings of lengths ranging from 0 to 24. We use strings because they force object allocation, stressing the memory system. The Filter predicate selects strings whose hashed value has an even number of `1` characters in it. The Reduce benchmark uses an combination function that produces a new string with an XOR between the characters in the string. The Sort benchmark sorts strings in lexicographically increasing order. The Breadth First Search operates on the Cage-15 example graph, publicly available at [1], which has 5154859 nodes and 94044692 edges. The Seam Carve benchmark performs the dynamic programming implementation of the seam carving algorithm on an image that is $10^7 \times 100$.

These benchmark problems contain a mixture of kernels, such as Tabulate, Filter, Reduce and Sort, and larger full problems like Breadth First Search and Seam Carving. Since the kernel benchmarks are smaller problems that don't exhibit the behavior that causes space blowup in current work, we don't expect to see an improvement on these benchmarks. We hope to have minimal overhead in both space and time, showing that our concurrent collection techniques don't negatively impact programs that don't benefit from the collection. The larger benchmarks make heavy use of parallel operations, and in particular are round based algorithms that allocate new arrays at each level of the computation. These programs are ones we are targeting with our concurrent system, and expect to see an improvement in space usage on these programs.

9.2 Results

Figures 17 and 18 show the results of our experiments on these benchmarks. We refer to current work as the *mpl* compiler, and our work as the *mpl-cgc* compiler. There remain some bugs in our implementation, so tests that were not able to run to completion are denoted by an X.

In Figure 17, we measure the overhead that using mpl-cgc has over mpl. Specifically, we measure the average time taken to execute a benchmark by mpl-cgc divided by the average time taken by mpl. This metric measures the time overhead of using our concurrent collection system. Values that are larger than 1 indicate that our system is slower than mpl, while values less than 1 indicate our system performed better than mpl. In general we see that our system suffers little overhead from mpl, and sometimes performs a little bit better than mpl. These results indicate

Space Improvement on N Cores (mpl / mpl-cgc)							
Benchmark	1	2	4	8	16	32	64
tabulate	1.01x	0.79x	0.93x	0.81x	0.85x	0.82x	0.84x
filter	1.01x	0.72x	0.81x	0.76x	0.76x	0.82x	0.84x
fib	1.06x	1.15x	1.07x	1.07x	1.21x	0.89x	0.75x
sort	1.00x	0.75x	0.77x	0.81x	0.76x	0.84x	0.79x
reduce	1.00x	1.14x	1.18x	1.17x	1.16x	1.21x	1.53x
seam-carve	1.00x	1.34x	1.35x	1.40x	1.45x	0.91x	X
bfs	0.98x	1.70x	1.74x	1.74x	1.75x	1.16x	0.98x
itertabulate	0.99x	2.00x	2.08x	1.21x	0.84x	0.80x	0.81x

Figure 18: Space (Maximum Residence) Improvement using Concurrent Collection on N cores. (Larger is better. Values $>$ 1 indicate mpl-cgc performs better.)

that our concurrent collection techniques do not impose a large overhead on the existing memory management system. In the case of breadth first search on 64 cores we see a large overhead however, which we are not able to explain. We aim to investigate the cause of this slowdown more in future work. In Figure 18, we measure the space improvement that using our system gets over mpl. To calculate this, we measure the space used by mpl on a benchmark divided by the space used by mpl-cgc on the same benchmark. To calculate the space used, we use the Unix `time` utility, and measure the maximum resident set size. We can see on the smaller benchmarks that our system performs worse than mpl, which is expected, as there is much less opportunity for concurrent collections to occur. Additionally, the concurrent collector suspends more memory outside the scope of local collections, and thus suffers a penalty when no concurrent collections are able to occur. However, as we move to the breadth first search and seam carving benchmarks we see large improvements in the space utilization – 1.7x and 1.4x respectively. As these benchmarks compose the kernel operations from above, there are many opportunities for the concurrent collector to run, which is why the results show improvement. Additionally, when we consider the iterated tabulate benchmark, we see a 2x space usage improvement on low core counts. This drastic improvement highlights how mpl is unable to effectively addressed repeated aggregate operations. However, as the cores increase the improvement plateaus and eventually falls off. We postulate this is due to the fact that a single collection thread can't compete with many mutators, as the mutators can generate garbage faster than the collector can collect.

10 Related Work

10.1 Parallel Programming Languages

There has been much work in designing both imperative and functional parallel programming languages. Work in parallel imperative languages extends these languages to support parallel programming by adding parallel constructs into the languages or compilers [13, 28, 29, 31]. This work includes languages like Cilk/Cilk++, Java Fork-Join and Habanero Java and many more. Programs written in these languages perform well due to taking advantage of effects and imperative programming features. However, reasoning about correctness within these imperative languages is challenging due to the presence of data races [4, 5, 14, 37]. Work in parallel functional languages considers functional languages and extends them and their runtime systems to support parallelism. There are multiple existing works in this area, including Parallel ML [24], MultiMLton [40], and forms of Parallel Haskell [15]. Due to functional programs don't use effects, they avoid the complications due to race conditions that imperative programs suffer from. The main research work now is to bridge the gap in performance between functional languages and imperative languages, which this work contributes to.

10.2 Memory Management

Automatic memory management is a feature of nearly all high level languages, and there has been much development in extending these systems to support parallelism and concurrency. A large amount of work has been in developing systems that use processor-local or thread-local heaps combined with a shared global heap that is collected cooperatively [6, 9, 18, 19]. These systems that use processor-local or thread-local techniques are inherently different from our approach which dynamically structures the heap according to the task structure of the program. Our system resembles these systems in a sense, where each task maintains a local heap, and the history is a global heap that is collected by the concurrent collection system. Linking memory management with the program level tasks rather than processors and OS-level threads allows us to take advantage of structural properties within the parallel program.

We can contrast most strongly with prior work by Raghunathan et al [35] and Guatto et al [26]. Raghunathan et al considered purely functional languages without effects, and a weaker notion of disentanglement that only allowed pointers to point upwards in the hierarchy. In contrast, our work allows for effects, and a stronger version of disentanglement that allows for pointers to point up and down in the hierarchy. Similarly to Raghunathan et al, Guatto et al consider a similar weak notion of disentanglement, but allow for effects. They manage effects by using a promotion technique to move objects up the hierarchy to maintain that all pointers point up. Our system allows for down pointers and effects as long as they satisfy disentanglement, avoiding costly promotions.

11 Future Work

11.1 Disentanglement Theory

Our formalization of disentanglement and determinacy race freedom with computation graphs is a powerful abstraction, which can open the doors to proving a wide variety of properties about disentanglement. For instance, we are interesting in investigating under what conditions disentanglement is compositional, which this formalization could prove helpful. Additionally, properties such as runtime checks for entanglement could be proved correct through these semantics. There is also an opportunity to mechanize these proofs in a formal theorem prover such as Coq to be confident about the absence of bugs. There is also a large amount of opportunity to simplify the presentation of the entire disentanglement proof to make it more accessible. Lastly, future work can formalize the semantics of garbage collection within the language framework we have described, and formally prove that disentanglement allows us to use hierarchical garbage collection safely.

11.2 Concurrent Collection

Our concurrent collection scheme shows promise at providing an efficient memory management scheme for certain classes of parallel programs. However, there are many limitations that should be addressed in future work.

As mentioned, our concurrent collector is a single threaded program, meaning that it cannot scale to compete with a high number of mutators. Implementing a parallel garbage collection scheme using a parallel depth first search [3] or other parallel non-moving collection algorithms is an interesting direction to see how this scheme can scale to higher core counts. Additionally, future work can try to exploit the structure of the history to perform collection in parallel more efficiently than a standard parallel traversal. We avoided following these paths during investigation of our work due to the significant implementation effort of such an endeavor, and the limited time available for this project.

Another limitation with our current scheme is that we can only schedule these concurrent collections when the program joins back into the top level. This means for certain types of programs that make heavy use of nested parallelism before joining back into the top level, our collector is not able to effectively collect on these programs. Future work to investigate how to perform collection on sub-graphs of the history rather than the entire history could more effectively support these types of programs.

Lastly, there is a large amount of interesting work on integrating the concurrent collection thread or threads with the work stealing scheduling system to increase efficiency of the collector. Integrating the collector with the scheduler could allow the scheduler to make more educated decisions about which hardware threads to execute the collection to make better use of data locality and caching within modern computer systems.

12 Conclusion

In this thesis, we examine disentanglement from a formal point of view, and show how disentanglement can be used to implement an memory management system that allows for efficient garbage collection for parallel programs. In particular, parallel functional programs that make use of effects in a disentangled way can be executed efficiently in parallel, and scale well to high core counts. As we show formally, the large class of determinacy race free parallel programs are disentangled, and can take advantage of our efficient memory management techniques.

13 Acknowledgments

This thesis is partly joint work with Sam Westrick (swestric@cs.cmu.edu), in particular the sections relating to formalization of disentanglement and the proof of determinacy race freedom implying disentanglement. I also thank Sam for his many hours of discussion and guidance over architecting the concurrent collection system with MLton. I thank Umut Acar for his constant guidance and patience in advising me over the past 2 years. My experience as an undergraduate without meeting Umut would have been drastically different, and I am grateful I was given the opportunity to work with him.

Bibliography

[1] URL `https://www.cc.gatech.edu/dimacs10/archive/matrix.shtml`. 9.1

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, 2013. 8.1

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 67:1–67:12, 2015. 8.4, 11.2

[4] Sarita V. Adve. Data races are evil with no exceptions: technical perspective. *Commun. ACM*, 53(11):84, 2010. doi: 10.1145/1839676.1839697. URL `http://doi.acm.org/10.1145/1839676.1839697`. 10.1

[5] T. R. Allen and D. A. Padua. Debugging Fortran on a shared memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, August 1987. 10.1

[6] Todd A. Anderson. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, pages 21–30, 2010. 10.2

[7] Andrew W. Appel. Simple generational garbage collection and fast allocation. 19(2):171–183, 1989. URL `http://www.cs.princeton.edu/fac/~appel/papers/143.ps`. 1, 6.2

[8] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC)*, pages 51–57, 2011. 1

[9] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC)*, pages 51–57, 2011. 10.2

[10] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM. doi: 10.1145/1275808.1276390. URL `http://doi.acm.org/10.1145/1275808.1276390`. 9.1

[11] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994. 1

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J.*

ACM, 46:720–748, September 1999. 8

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995. 10.1

[14] 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. 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*, OOPSLA '09, pages 97–116, 2009. 10.1

[15] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007*, pages 10–18, 2007. 10.1

[16] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970. 6.2, 6.2

[17] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 125–136, 2001. 7.1

[18] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. URL `ftp://ftp.inria.fr/INRIA/Projects/para/doligez/DoligezGonthier94.ps.gz`. 1, 10.2

[19] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. pages 113–123. URL `file://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/publications/concurrent-gc.ps.gz`. 1, 10.2

[20] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, June 1997. 1, 5.1

[21] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999. 1, 5.1

[22] Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2005. 1, 5.1

[23] Christine H Flood, David Detlefs, Nir Shavit, and Xiolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *JVM '01*, 2001. 7.1

[24] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008. 10.1

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

[26] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut Acar, and Matthew Fluet. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 81–93, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4982-6. doi: 10.1145/3178487.3178494. URL `http://doi.acm.org/10.1145/3178487.3178494`. 1, 2, 6, 6.2, 6.2, 8, 10.2

[27] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17. ACM, 1984. 1

[28] Shams Mahmood Imam and Vivek Sarkar. 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*, pages 75–86, 2014. 10.1

[29] *Intel Cilk++ SDK Programmer's Guide*. Intel Corporation, October 2009. Document Number: 322581-001US. 10.1

[30] Matthew Le and Matthew Fluet. Partial aborts for transactions via first-class continuations. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 230–242, 2015. ISBN 978-1-4503-3669-7. 1

[31] Doug Lea. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*, pages 36–43, 2000. 10.1

[32] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. AI Memo 569a, MIT, April 1981. URL `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-569a.pdf`. 6.2

[33] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-134-7. doi: 10.1145/1375634.1375637. URL `http://doi.acm.org/10.1145/1375634.1375637`. 7.1

[34] MLton. MLton web site. `http://www.mlton.org`. 8

[35] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 392–406, New York, NY, USA, 2016. ACM. 2, 6, 6.2, 8, 10.2

[36] Daniel Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, May 2009. URL `https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf`. 1

[37] Guy L. Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 218–231. ACM Press, 1990. 10.1

[38] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984. 6.2

[39] Yifan Xu, I-Ting Angelina Lee, and Kunal Agrawal. Efficient parallel determinacy race detection for two-

dimensional dags. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 368–380, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4982-6. doi: 10.1145/3178487.3178515. URL http://doi.acm.org/10.1145/3178487.3178515. 5.1

[40] Lukasz Ziarek, K. C. Sivaramakrishnan, and Suresh Jagannathan. Composable asynchronous events. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 628–639, 2011. 10.1

14 Appendix

14.1 Proof of Lemma 4

Proof. By induction on the derivation of $W \,;\, A \vdash_{\mu} G \cdot H \cdot T\ \mathsf{drfde}$.

Case DRFDE-E. By assumption, $\ell \notin H$, and μ, μ' agree at all locations except ℓ, therefore $\forall \ell' \in H.\ \mathsf{locs}(\mu'(\ell')) \subseteq A \uplus H$. By assumption, $W' \vdash \mathsf{W}(\mathsf{GT}(G, e))\ \mathsf{drf}$, therefore $W' \vdash G\ \mathsf{drf}$. By assumption, $\forall \ell' \in A.(\ell' \notin W) \Rightarrow \mathsf{locs}(\mu(\ell')) \subseteq A \uplus H$. $\ell \in W'$, therefore $\forall \ell' \in A.(\ell' \notin W) \Rightarrow \mathsf{locs}(\mu'(\ell')) \subseteq A \uplus H$, as ℓ is excluded from locations considered in the premise.

Case DRFDE-PAR. By assumption, $W' \vdash \mathsf{W}(\mathsf{GT}(G, T))\ \mathsf{drf}$, therefore $W' \vdash G\ \mathsf{drf}$. By induction, both $W' \cup \mathsf{W}(\mathsf{GT}(G_2, T_2)) \,;\, A \uplus H \vdash_{\mu'} G_1 \cdot H_1 \cdot T_1\ \mathsf{drfde}$ and $W' \cup \mathsf{W}(\mathsf{GT}(G_1, T_1)) \,;\, A \uplus H \vdash_{\mu'} G_2 \cdot H_2 \cdot T_2\ \mathsf{drfde}$, as $\ell \notin H \uplus D(\llbracket T \rrbracket)$. By similar reasoning in the previous case, the final two premises hold as well.

Case DRFDE-TAPP. By induction, the first premise holds. Since A, H, e are unchanged, the second premise holds.

Case DRFDE-ℓAPP, DRFDE-TPAIR ,DRFDE-ℓPAIR ,DRFDE-FST ,DRFDE-SND ,DRFDE-REF ,DRFDE-BANG ,DRFDE-TUPD, and DRFDE-ℓUPD are all similar to case DRFDE-TAPP. □

14.2 Proof of Lemma 5

Proof. By induction on the derivation of $\mu \,;\, G \,;\, H \,;\, T \longmapsto \mu' \,;\, G' \,;\, H' \,;\, T'$.

Case PARL-G. By induction, if $\mathsf{W}(\mathsf{GT}(G_1', T_1')) = \mathsf{W}(\mathsf{GT}(G_1, T_1)) \cup \mathsf{write}\ \ell$, then $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus H_1 \uplus D(\llbracket T_1 \rrbracket)$. Therefore, $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus H_1 \uplus H_2 \uplus D(\llbracket T_1 \rrbracket) \uplus D(\llbracket T_2 \rrbracket)$, and same for ℓ.

Case PARR-G similar to PARL-G.

Case ALLOC-G. By assumption, $\mathsf{locs}(s) \subseteq A \uplus H$, therefore $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus H'$, and $\ell \in H'$.

Case UPD-G similar to ALLOC-G.

Cases APPSL-G, APPSR-G, PAIRSL-G, PAIRSR-G, FSTS-G, SNDS-G, REFS-G, BANGS-G, UPDSL-G, and UPDSR-G all similar to PARL-G.

Cases BANG-G, APP-G, FORK-G, JOIN-G hold vacuously. □

14.3 Proof of Lemma 6

Proof. By induction on the derivation of $\mu \,;\, G \,;\, H \,;\, T \longmapsto \mu' \,;\, G' \,;\, H' \,;\, T'$.

Case ALLOC-G. We have $W ; A \vdash_{\mu} G{\cdot}H{\cdot}s\ \mathsf{drfde}$ and $W \vdash \mathsf{GT}(G', \ell)\ \mathsf{drf}$ where $\ell \notin \mathrm{dom}(\mu)$ and $G' = G \oplus \mathsf{write}\ \ell$. Need to show $W \,;\, A \vdash_{\mu'} G' \cdot H' \cdot \ell\ \mathsf{drfde}$ where $\mu' = \mu[\ell \hookrightarrow s]$ and $H' = H \uplus \ell$. We have $\ell \in A \uplus H'$. By $W \,;\, A \vdash_{\mu} G \cdot H \cdot s\ \mathsf{drfde}$ we have $\mathsf{locs}(s) \subseteq A \uplus H$, thus $\mathsf{locs}(\mu'(\ell)) = \mathsf{locs}(s) \subset A \uplus H'$, and μ' agrees with μ on all other locations. Therefore $W \,;\, A \vdash_{\mu'} G' \cdot H' \cdot \ell\ \mathsf{drfde}$.

Case UPD-G. We have $W \,; A \vdash_{\mu[\ell_1 \hookrightarrow s]} G{\cdot}H{\cdot}(\ell_1 := \ell_2)\ \mathsf{drfde}$ and $W \vdash \mathsf{GT}(G', \ell_2)\ \mathsf{drf}$ where $G' = G \oplus \mathsf{write}\ \ell_1$. Need to show $W \,;\, A \vdash_{\mu'} G' \cdot H \cdot \ell_2\ \mathsf{drfde}$ where $\mu' = \mu[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2]$. By $W \,;\, A \vdash_{\mu[\ell_1 \hookrightarrow s]} G \cdot H \cdot (\ell_1 := \ell_2)\ \mathsf{drfde}$ we have $\ell_2 \in A \uplus H$ and therefore $\mathsf{locs}(\mu'(\ell_1)) \subseteq A \uplus H$. Since μ' agrees with μ on all other locations, we have $W \,;\, A \vdash_{\mu'} G' \cdot H \cdot \ell_2\ \mathsf{drfde}$.

Case BANG-G. We have $W \,;\, A \vdash_{\mu} G \cdot H \cdot !\,\ell\ \mathsf{drfde}$, and $W \vdash \mathsf{GT}(G', \ell')\ \mathsf{drf}$ where $G' = G \oplus \mathsf{read}\ \ell$. Need to show $W \,;\, A \vdash_{\mu} G' \cdot H \cdot \ell'\ \mathsf{drfde}$. Because $W \,;\, A \vdash_{\mu} G \cdot H \cdot !\,\ell\ \mathsf{drfde}$, and $W \vdash \mathsf{GT}(G', \ell')\ \mathsf{drf}$, $\mathsf{locs}(\ell) \subseteq A \uplus H$

therefore $W \, ; A \vdash_{\mu} G' \cdot H \cdot \ell' \ \mathsf{drfde}$.

Cases Fst-G and Snd-G are similar to Bang-G.

Case App-G. We have $W \, ; A \vdash_{\mu} G \cdot H \cdot (\ell_1 \ \ell_2) \ \mathsf{drfde}$, and $W \vdash \mathsf{GT}(G', ([\ell_1, \ell_2 / f, x] e)) \ \mathsf{drf}$, where $\mu(\ell_1) = \mathsf{fun} \ f \ x \ \mathsf{is} \ e$ and $G' = G \oplus \mathsf{read} \, \ell_1$. Need to show $W \, ; A \vdash_{\mu} G' \cdot H \cdot ([\ell_1, \ell_2 / f, x] e) \ \mathsf{drfde}$. By $W \, ; A \vdash_{\mu} G \cdot H \cdot (\ell_1 \ \ell_2) \ \mathsf{drfde}$ and $W \vdash \mathsf{GT}(G', ([\ell_1, \ell_2 / f, x] e)) \ \mathsf{drf}$, $\ell_1, \ell_2 \in A \uplus H$, and $\mathsf{locs}(\mu(\ell_1)) \subseteq A \uplus H$, therefore $W \, ; A \vdash_{\mu} G' \cdot H \cdot ([\ell_1, \ell_2 / f, x] e) \ \mathsf{drfde}$.

Case AppSL-G. We have $W \, ; A \vdash_{\mu} G \cdot H \cdot (T \ e) \ \mathsf{drfde}$, and $W \vdash \mathsf{GT}(G', (T' \ e)) \ \mathsf{drf}$. Need to show $W \, ; A \vdash_{\mu'} G' \cdot H' \cdot (T' \ e) \ \mathsf{drfde}$. Inductively we have $W \, ; A \vdash_{\mu'} G' \cdot H' \cdot T' \ \mathsf{drfde}$, where H' is an extension of H respectively. Therefore, $\mathsf{locs}(e) \subseteq A \uplus H'$ and $W \, ; A \vdash_{\mu'} G' \cdot H' \cdot (T' \ e) \ \mathsf{drfde}$.

Cases AppSL-G, AppSR-G, PairSL-G, PairSR-G, FstS-G, SndS-G, RefS-G, BangS-G, UpdSL-G, and UpdSR-G all similar to AppSL-G.

Case Fork-G. We have $W \, ; A \vdash_{\mu} G \cdot H \cdot \langle e_1 \parallel e_2 \rangle \ \mathsf{drfde}$, and $W \vdash \mathsf{GT}(G', \llparenthesis \bullet \cdot \emptyset \cdot e_1 \parallel \bullet \cdot \emptyset \cdot e_2 \rrparenthesis) \ \mathsf{drf}$, where $G' = G' \oplus \mathsf{none}$. Need to show $W \, ; A \vdash_{\mu} G' \cdot H \cdot \llparenthesis \bullet \cdot \emptyset \cdot e_1 \parallel \bullet \cdot \emptyset \cdot e_2 \rrparenthesis \ \mathsf{drfde}$. Need to show $(W \cup \mathsf{W}(\bullet)) \, ; (A \uplus H) \vdash_{\mu} \bullet \cdot \emptyset \cdot e_1 \ \mathsf{drfde}$ (other case is symmetric), as other premises hold by assumption. By $W \, ; A \vdash_{\mu} G \cdot H \cdot \langle e_1 \parallel e_2 \rangle \ \mathsf{drfde}$, $\mathsf{locs}(e_1) \subseteq A \uplus H$. $W \cup \mathsf{W}(\bullet) \vdash \bullet \ \mathsf{drf}$ as an axiom. By $W \, ; A \vdash_{\mu} G \cdot H \cdot \langle e_1 \parallel e_2 \rangle \ \mathsf{drfde}$, $\forall \ell \in A. \ (\ell \notin W) \Rightarrow \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H$, and $\forall \ell \in H. \, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H$. Therefore, $\forall \ell \in (A \uplus H). \ (\ell \notin (W \cup \mathsf{W}(\bullet))) \Rightarrow \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H$. This completes the case.

Case Join-G. We have $W \, ; A \vdash_{\mu} G \cdot H \cdot \llparenthesis G_1 \cdot H_1 \cdot \ell_1 \parallel G_2 \cdot H_2 \cdot \ell_2 \rrparenthesis \ \mathsf{drfde}$, and $W \vdash \mathsf{GT}(G', \langle \ell_1, \ell_2 \rangle) \ \mathsf{drf}$, where $G' = G \oplus (G_1 \otimes G_2)$. Need to show $W \, ; A \vdash_{\mu} G \oplus (G_1 \otimes G_2) \cdot (H \uplus H_1 \uplus H_2) \cdot \langle \ell_1, \ell_2 \rangle \ \mathsf{drfde}$. By assumption, $W \vdash G' \ \mathsf{drf}$. By $W \, ; A \vdash_{\mu} G \cdot H \cdot \llparenthesis G_1 \cdot H_1 \cdot \ell_1 \parallel G_2 \cdot H_2 \cdot \ell_2 \rrparenthesis \ \mathsf{drfde}$, $\ell_1 \in A \uplus H \uplus H_1$ and $\ell_2 \in A \uplus H \uplus H_2$. By $W \, ; A \vdash_{\mu} G \cdot H \cdot \llparenthesis G_1 \cdot H_1 \cdot \ell_1 \parallel G_2 \cdot H_2 \cdot \ell_2 \rrparenthesis \ \mathsf{drfde}$, $\forall \ell \in H_1. \, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus H_1$, and similarly for H_2. Therefore, $\forall \ell \in H \uplus H_1 \uplus H_2. \, \mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus H_1 \uplus H_2$. This concludes the case.

Case ParL-G. Consider two separate cases, $\mathsf{W}(\mathsf{GT}(G_1, T_1)) = \mathsf{W}(\mathsf{GT}(G_1', T_1'))$ and $\mathsf{W}(\mathsf{GT}(G_1, T_1)) \neq \mathsf{W}(\mathsf{GT}(G_1', T_1'))$. When $\mathsf{W}(\mathsf{GT}(G_1, T_1)) = \mathsf{W}(\mathsf{GT}(G_1', T_1'))$, $\mu = \mu'$, and $H = H'$. Therefore, $W \cup \mathsf{W}(\mathsf{GT}(G_2, T_2)) ; A \uplus H \vdash_{\mu'} G_1' \cdot H_1' \cdot T_1' \ \mathsf{drfde}$ and $W \cup \mathsf{W}(\mathsf{GT}(G_1', T_1')) \, ; A \uplus H \vdash_{\mu'} G_2 \cdot H_2 \cdot T_2 \ \mathsf{drfde}$, which concludes the case.

When $\mathsf{W}(\mathsf{GT}(G_1, T_1)) \neq \mathsf{W}(\mathsf{GT}(G_1', T_1'))$, then $\mathsf{W}(\mathsf{GT}(G_1', T_1')) = \mathsf{W}(\mathsf{GT}(G_1, T_1)) \cup \mathsf{write} \, \ell$ and μ, μ' agree at all locations except ℓ. By Lemma 5, $\ell \in A \uplus H \uplus H_1' \uplus D(\llbracket T_1' \rrbracket)$ and $\mathsf{locs}(\mu(\ell)) \subseteq A \uplus H \uplus H_1' \uplus H_2 \uplus D(\llbracket T_1' \rrbracket) \uplus D(\llbracket T_2 \rrbracket)$. Therefore, $\forall \ell' \in A. (\ell' \notin W) \Rightarrow \mathsf{locs}(\mu'(\ell')) \subseteq A \uplus H \uplus H_1' \uplus H_2 \uplus D(\llbracket T_1' \rrbracket) \uplus D(\llbracket T_2 \rrbracket)$ and $\forall \ell' \in H. \, \mathsf{locs}(\mu'(\ell')) \subseteq A \uplus H \uplus H_1' \uplus H_2 \uplus D(\llbracket T_1' \rrbracket) \uplus D(\llbracket T_2 \rrbracket)$. By heap disjointedness from typing, and Lemma 5, $\ell \notin H_2 \uplus D(\llbracket T_2 \rrbracket)$. Therefore, Lemma 4 can be applied to derive $W \cup \mathsf{W}(\mathsf{GT}(G_1', T_1')) ; A \uplus H \vdash_{\mu'} G_2 \cdot H_2 \cdot T_2 \ \mathsf{drfde}$. All remaining premises hold by induction and assumption, concluding the case.

Case ParR-G similar to ParL-G.

□