Hierarchical Memory Management for Parallel Programs

Ram Raghunathan[*] Stefan K. Muller[*] Umut A. Acar[*†] Guy Blelloch[*]

[*]Carnegie Mellon University, USA [†]Inria, France

{ram.r, smuller, umut, blelloch}@cs.cmu.edu

Abstract

An important feature of functional programs is that they are parallel by default. Implementing an efficient parallel functional language, however, is a major challenge, in part because the high rate of allocation and freeing associated with functional programs requires an efficient and scalable memory manager.

In this paper, we present a technique for parallel memory management for strict functional languages with nested parallelism. At the highest level of abstraction, the approach consists of a technique to organize memory as a hierarchy of heaps, and an algorithm for performing automatic memory reclamation by taking advantage of a disentanglement property of parallel functional programs. More specifically, the idea is to assign to each parallel task its own heap in memory and organize the heaps in a hierarchy/tree that mirrors the hierarchy of tasks.

We present a nested-parallel calculus that specifies hierarchical heaps and prove in this calculus a *disentanglement* property, which prohibits a task from accessing objects allocated by another task that might execute in parallel. Leveraging the disentanglement property, we present a garbage collection technique that can operate on any subtree in the memory hierarchy concurrently as other tasks (and/or other collections) proceed in parallel. We prove the safety of this collector by formalizing it in the context of our parallel calculus. In addition, we describe how the proposed techniques can be implemented on modern shared-memory machines and present a prototype implementation as an extension to MLton, a high-performance compiler for the Standard ML language. Finally, we evaluate the performance of this implementation on a number of parallel benchmarks.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — Memory management (garbage collection); D.1.3 [*Programming Techniques*]: Concurrent Programming — Parallel Programming; D.4.1 [*Operating Systems*]: Process Management — Scheduling

General Terms Languages, Performance

Keywords Languages, Memory Management, Parallelism, Scheduling

1. Introduction

In the past two decades, there has been a large body of work on both parallel garbage collection and parallel scheduling. In this paper, we propose an approach for coupling the two together, which is particularly useful in functional languages. The basic idea is to organize memory in a way that mirrors the structure of parallelism in the computation and takes advantage of the independence of parallel tasks to perform garbage collection. While this general idea may be applicable to the broader array of parallel functional programming languages, we focus here on strict languages, such as the ML family, extended with support for nested parallelism.

More specifically, instead of using a single shared heap [6, 9, 21, 26, 32] or a two-level heap consisting of local processor heaps and a global heap [5, 8, 23, 25, 31, 37], we suggest a hierarchy of heaps (tree of heaps) that is tied to the nesting of tasks. In nested parallel (fork-join) parallelism, the tasks form a natural nesting, and memory that is allocated and referenced has a structure that is related to this nesting, which we would like to leverage. In strict functional computations, pointers only point up the hierarchy, creating what we refer to as a *disentangled* heap. Furthermore, what is often understood as temporal locality in sequential programs becomes nested task locality in nested parallel computations. For example, in divide-and-conquer algorithms, each node in the recursion tree has some locality within its subtree [14, 45].

The motivation for organizing the heap into a hierarchy is threefold. Firstly, it allows for simpler parallel collection since the nodes in the tree can be collected individually and separately, especially if they can be kept disentangled. Secondly, the hierarchical organization of heaps is better suited for supporting memory locality at multiple levels of a modern cache hierarchy, for which some of the higher levels are shared among cores on multiprocessors. By having the size of heaps grow going up the levels, and moving values up the heap as they are collected, the approach can be seen as the natural extension of a multi-level generational collector. Finally, the organization allows us to naturally tie memory management decisions to scheduler decisions. For example, a good time to collect is probably when a task finishes, or a good time to create a heap in the hierarchy is likely when stealing a task.

We propose a specific memory manager and garbage collection algorithm based on the hierarchical heaps and disentanglement. Rather than going directly to a low-level description of the algorithms, we formalize a functional, ML-like language with nested parallelism and a memory manager by presenting a reasonably high-level operational semantics that accounts for all the key aspects of the memory management and collection, while abstracting away from the details. The semantics models parallelism using nondeterministic interleaving of parallel steps. It captures a parallel fork by "activating" a parallel tuple. This is meant to capture the lower-level idea of scheduling a parallel task (e.g. stealing the task in a work-stealing scheduler), and allows for lazy task creation (i.e. if there is plenty of parallelism in the computation, a parallel fork need never be activated). The semantics captures the hierarchical nature of the heap and creates new heaps exactly when a parallel tuple is activated. The model interleaves garbage collection steps with steps of the mutator. A node in the hierarchy is "locked" and cannot evaluate when it is being collected, but other nodes are allowed to evaluate concurrently. Our semantics captures the object-by-object copying nature of our collector, but ignores details of how memory is laid out or allocated.

Based on the semantics, we prove that the disentanglement property holds on programs written in our language, and that the garbage collector is correct (memory safe and meaning-preserving).

To remain as general as possible, the semantics does not specify a scheduler, and abstracts over many important details a real implementation must consider such as concurrency, contention, and performance. We briefly describe the key algorithms and data structures for realizing the semantics (Section 4) based on a popular work-stealing scheduler [3, 7, 15].

We have implemented a prototype of our collector as part of the MLton compiler [39] by adding it to Spoonhower's parallel scheduler [47]. The implementation follows our semantics and design but has to consider a variety of other details. We present some preliminary performance results on a handful of benchmarks. The collector achieves good performance compared to both the existing collector on MLton and the Manticore runtime system [8], although the numbers are meant as a proof of concept rather than a careful analysis.

The contributions of this paper include the following.

- A hierarchical model of memory for a nested-parallel functional language.
- The precise formulation of the disentanglement property in the context of our nested-parallel functional language, and the proof that this language guarantees disentanglement.
- The formulation of a hierarchical garbage collection technique that allows portions of memory to be garbage collected concurrently with other tasks.
- The design of a runtime system that realizes the hierarchical memory management techniques.
- A prototype implementation as an extension to the MLton compiler for Standard ML, and an evaluation of its performance.

2. Overview

We present a high-level, informal overview of the ideas proposed in this paper. The rest of the paper makes these ideas precise.

We use the quicksort algorithm as an example and assume an ML-like, strict, purely functional language with nested parallelism, where the primitive `par` creates a tuple by allowing its components to be evaluated in parallel. Figure 1 shows the code for quicksort in such a language. For simplicity, we assume that the input contains no duplicates. The function `qsort` uses the first item as a pivot to partition the input list `l` into less-than and greater-than parts, written `ll` and `lg`, sorts them recursively in parallel, and concatenates the sorted lists to compute the final sorted output.

Task Tree. We can represent an evaluation of the quicksort algorithm with a *task tree* consisting of parallel *tasks* where each task corresponds to a parallel sub-computation as indicated by the `par` primitive. Figure 2 shows such a tree representation of the evaluation of the `qsort` function on the input list `[5; 6; 3; 4; 1; 9]`. The nodes of the task tree represent parallel tasks and the edges represent the control dependencies between them. For example, the root task, labeled T_A, which corresponds to the first call to the function, forks two new tasks T_B and T_C.

```
fun qsort l =
  case l of
    nil ⇒nil
  | h::nil ⇒l
  | h::t ⇒
    let
      (ll,lg) = partition h t
      (sll,slg) = par (qsort ll, qsort lg)
    in
      append sll (h::slg)
    end
```

Figure 1. The code for quicksort.

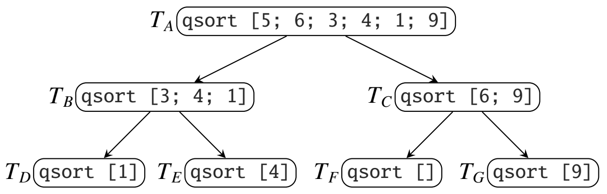

Figure 2. An example run of quicksort. The nodes represent (parallel) tasks and the edges represent the control dependencies between them.

Memory as a Heap Tree and Disentanglement. Classically, memory is viewed in programming languages as a flat, unstructured pool of objects, or a fixed hierarchy of several heaps, referred to as generations. In this paper, we propose an approach where we structure memory as a dynamic hierarchy by partitioning memory into *heaps*, each of which corresponds to a task in the task tree. To create this hierarchy, we dedicate a fresh heap to each forked task and allocate all memory requested by the task in its heap, which the task owns. When the task completes, we join its heap with that of its parent. As a result, we can organize the memory at any point during execution as a hierarchy, represented as a tree of heaps as follows. Let H_A and H_B be two heaps. If there is an edge between their owner tasks, T_A and T_B, respectively, then insert an edge between H_A and H_B. Otherwise, don't insert any edge. We refer to the resulting tree on the heaps as a *heap tree*. The motivation behind our approach is to organize memory in a way that reflects the structure—specifically the dependencies—of the computation.

Figure 3 illustrates the task and the heap trees for our example. Each "qsort" node represents a task. The gray box around the node represents its heap. Tasks and the edges between them define the task tree. Heaps and the edges between them define the heap tree. The figure also shows the data stored in each heap. The root heap, H_A, contains the input list `L0` and the two lists obtained via partitioning, `L1` and `L2`. The task T_B takes the input `L1` from the root heap H_A and partitions it into two new lists `L3` and `L4`. The task T_C takes the input `L2` from the root heap H_A and produces the lists `L5` and `L6`.

Having organized memory as a tree of heaps, we observe the following key *disentanglement* property. Let T_A and T_B be any two tasks in the task tree and let H_A and H_B be their heaps in the heap tree. If some object in H_A references (or points to) an object in H_B at any time during execution, then H_B is an ancestor of H_A in the heap tree. Intuitively speaking, this property holds because, in a purely functional language, a parallel task has access only to the objects that have been allocated before its execution or by the task itself, excluding the objects that are allocated by tasks that might have been executed in parallel (even if they are not actually executed in parallel). Such objects are allocated precisely by the tasks that are on the path from the task to the root of the task tree. Since all memory allocated by a task is placed in the heap of that task, and since there

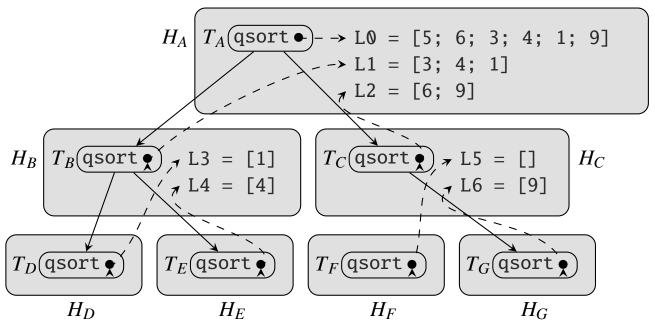

Figure 3. The heap and task tree for the the quicksort example. Heaps are shown as gray boxes. As shown by the dashed arrows, pointers in heaps may point to higher, but not lower or cross, heaps.

is an isomorphism between the heap tree and the task tree as defined by the ownership relation, the disentanglement property holds. As a corollary, we can establish the following: if T_A is neither an ancestor nor a descendant of T_B, then objects in H_A do not reference objects in H_B and objects in H_B do not reference objects in H_A. In other words, if T_A and T_B are concurrent tasks that may be executed in parallel, then H_A and H_B are guaranteed to be "disentangled". As suggested by disentanglement, in the quicksort example shown in Figure 3, all pointers in the memory point "up" in the heap tree: if an object in heap H_A points to another object in another heap H_B, then H_B is an ancestor of H_A.

In Sections 3.1 and 3.2, we make precise this high-level description of hierarchical heaps and disentanglement by presenting a semantics that restricts a task's memory access to its own heap and ancestor heaps. We prove that a program evaluated with hierarchical heaps and restricted memory access has the same behavior as it would have under a standard operational semantics, thus establishing the disentanglement property.

Garbage Collection. When considered in the context of the heap tree, the disentanglement property leads us to observe that any leaf heap H in the heap tree can be collected by the task T that owns that heap independently of all the other heaps and tasks. This holds because, via disentanglement, H does not have any "roots" or pointers into it. Thus, garbage collection of H can proceed as all the other tasks are executing. A moving (copying) or a non-moving collection can be used. We can generalize this observation to any subtree in the heap tree as follows: the heaps in any subtree of the heap tree can be collected independently of all other heaps. As with a leaf of the heap tree, a subtree of heaps has no incoming pointers from outside the subtree, and thus the collection can proceed concurrently with all other tasks operating on the other heaps. This means that we can garbage collect a subtree of heaps by stopping only those tasks that own them as we allow other tasks to continue executing. Furthermore, the subtree can be garbage collected in parallel by taking advantage of disentanglement: all the leaves can be collected in parallel and, after their collections are completed, the parents of the leaves may be collected in parallel, and so on until the root is reached.

For example, in Figure 3, we can garbage collect heaps H_B, H_D and H_E, which form a subtree rooted at H_B, by stopping only the tasks T_B, T_D and T_E, which own the heaps. To garbage collect the subtree, we can collect H_D and H_E in parallel, and then H_B.

In Section 3.3, we make precise this informal description of garbage collection and establish its correctness by showing that collection does not free rechable locations (memory safety) or alter the behavior of the program (meaning preservation).

Types	τ	::=	$\texttt{nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2$
Large Values	v	::=	$\texttt{n} \mid \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \mid \langle \ell, \ell \rangle$
Heaps	H	::=	$\emptyset \mid H[\ell \mapsto v]$
(Heap) Paths	P	::=	$[\,] \mid H :: P$
Expressions	e	::=	$x \mid \ell \mid v \mid e\ e \mid$ $\texttt{fst}(e) \mid \texttt{snd}(e) \mid \langle e, e \rangle \mid \triangleleft e, e \triangleright \mid \blacktriangleleft T, T \blacktriangleright$
Tasks	T	::=	$H \cdot e$

Figure 4. Syntax of λ^{HP}

Heap typing $\quad \vdash_{\Sigma'} H : \Sigma$

$$\frac{}{\vdash_\Sigma \emptyset : \cdot} \qquad \frac{\cdot \vdash_{\Sigma,\Sigma'} v : \tau \qquad \vdash_{\Sigma'} H : \Sigma}{\vdash_{\Sigma'} H[\ell \mapsto v] : \Sigma, \ell : \tau}$$

Path typing $\quad P : \Sigma$

$$\frac{}{[\,] : \cdot}\ \textsc{S-EPath} \qquad \frac{\vdash_{\Sigma_2} H : \Sigma_1 \qquad P : \Sigma_2}{H :: P : \Sigma_2, \Sigma_1}\ \textsc{S-Path}$$

Expression typing $\quad \Gamma \vdash_P e : \tau$

$$\frac{}{\Gamma, x : \tau \vdash_\Sigma x : \tau}\ \textsc{S-Var} \qquad \frac{}{\Gamma \vdash_{\Sigma,\ell:\tau} \ell : \tau}\ \textsc{S-Loc} \qquad \frac{}{\Gamma \vdash_\Sigma \texttt{n} : \texttt{nat}}\ \textsc{S-Nat}$$

$$\frac{\Gamma, x : \tau, f : \tau \to \tau' \vdash_\Sigma e : \tau'}{\Gamma \vdash_\Sigma \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} : \tau \to \tau'}\ \textsc{S-Fun} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \qquad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \textsc{S-Pair}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau \to \tau' \qquad \Gamma \vdash_\Sigma e_2 : \tau}{\Gamma \vdash_\Sigma e_1\ e_2 : \tau'}\ \textsc{S-App} \qquad \frac{\Gamma \vdash_\Sigma e : \tau_1 \times \tau_2}{\Gamma \vdash_\Sigma \texttt{fst}(e) : \tau_1}\ \textsc{S-Fst}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau_1 \times \tau_2}{\Gamma \vdash_\Sigma \texttt{snd}(e) : \tau_2}\ \textsc{S-Snd} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \tau_1 \qquad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma \triangleleft e_1, e_2 \triangleright : \tau_1 \times \tau_2}\ \textsc{S-Par}$$

$$\frac{\Gamma \vdash_\Sigma T_1 : \tau_1 \qquad \Gamma \vdash_\Sigma T_2 : \tau_2}{\Gamma \vdash_\Sigma \blacktriangleleft T_1, T_2 \blacktriangleright : \tau_1 \times \tau_2}\ \textsc{S-ParA}$$

Task typing $\quad \Gamma \vdash_\Sigma T : \tau$

$$\frac{\vdash_{\Sigma'} H : \Sigma \qquad \Gamma \vdash_{\Sigma,\Sigma'} e : \tau}{\Gamma \vdash_{\Sigma'} H \cdot e : \tau}\ \textsc{S-RunTask}$$

Figure 5. Statics of λ^{HP}

3. The Language

We present an ML-style functional calculus called λ^{HP} and an operational semantics that accounts for memory allocation and the scheduling of tasks, and enforces the property that tasks access memory only in their own heap and ancestor heaps. We then show that any λ^{HP} program is disentangled in that it may be run safely and that it is both deterministic and equivalent to a standard "flat" semantics where memory is treated conventionally. We then formalize a hierarchical memory manager on top of λ^{HP} and prove its correctness.

3.1 A Core Calculus for Disentanglement

Abstract syntax. Figure 4 shows the syntax of the language λ^{HP}. The types include natural numbers (as the sole base type), functions and products. To present a precise accounting of memory operations, we distinguish between *large values* v, which are always allocated in the heap, and *small values* consisting only of memory locations, written ℓ. Large values include natural numbers, named recursive functions and pairs of heap locations. Large values are *not* irreducible, but rather step to heap locations ℓ, which are the only

$$\frac{\ell \text{ fresh}}{H; v \to_P \ell; H[\ell \mapsto v]}\ \textbf{D-Alloc} \qquad \frac{\begin{array}{c} H :: P; \ell_1 \xrightarrow{lookup} \texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end} \\ H :: P; \ell_2 \xrightarrow{lookup} v \end{array}}{H; \ell_1\ \ell_2 \to_P [\ell_1, \ell_2/f, x]e; H}\ \textbf{D-AppE} \qquad \frac{H; e_1 \to_P e_1'; H'}{H; e_1\ e_2 \to_P e_1'\ e_2; H'}\ \textbf{D-AppS1} \qquad \frac{H; e_2 \to_P e_2'; H'}{H; \ell_1\ e_2 \to_P \ell_1\ e_2'; H'}\ \textbf{D-AppS2}$$

$$\frac{H :: P; \ell \xrightarrow{lookup} \langle v_1, v_2 \rangle}{H; \texttt{fst}(\ell) \to_P v_1; H}\ \textbf{D-FstE} \qquad \frac{H; e \to_P e'; H'}{H; \texttt{fst}(e) \to_P \texttt{fst}(e'); H'}\ \textbf{D-FstS} \qquad \frac{H :: P; \ell \xrightarrow{lookup} \langle v_1, v_2 \rangle}{H; \texttt{snd}(\ell) \to_P v_2; H}\ \textbf{D-SndE} \qquad \frac{H; e \to_P e'; H'}{H; \texttt{snd}(e) \to_P \texttt{snd}(e'); H'}\ \textbf{D-SndS}$$

$$\frac{H; e_1 \to_P e_1'; H'}{H; \langle e_1, e_2 \rangle \to_P \langle e_1', e_2 \rangle; H'}\ \textbf{D-PairS1} \qquad \frac{H; e_2 \to_P e_2'; H'}{H; \langle \ell_1, e_2 \rangle \to_P \langle \ell_1, e_2' \rangle; H'}\ \textbf{D-PairS2} \qquad \frac{}{H; \lhd \ell_1, \ell_2 \rhd \to_P \langle \ell_1, \ell_2 \rangle; H}\ \textbf{D-ParE}$$

$$\frac{H; e_1 \to_P e_1'; H'}{H; \lhd e_1, e_2 \rhd \to_P \lhd e_1', e_2 \rhd; H'}\ \textbf{D-ParS1} \qquad \frac{H; e_2 \to_P e_2'; H'}{H; \lhd \ell_1, e_2 \rhd \to_P \lhd \ell_1, e_2' \rhd; H'}\ \textbf{D-ParS2} \qquad \frac{T_1 = \emptyset \cdot e_1 \qquad T_2 = \emptyset \cdot e_2}{H; \lhd e_1, e_2 \rhd \to_P \blacktriangleleft T_1, T_2 \blacktriangleright; H}\ \textbf{D-Activate}$$

$$\frac{T_1 = H_1 \cdot \ell_1 \qquad T_2 = H_2 \cdot \ell_2}{H; \blacktriangleleft T_1, T_2 \blacktriangleright \to_P \langle \ell_1, \ell_2 \rangle; H \uplus H_1 \uplus H_2}\ \textbf{D-ParAE} \qquad \frac{T_1 \Rightarrow_{H::P} T_1'}{H; \blacktriangleleft T_1, T_2 \blacktriangleright \to_P \blacktriangleleft T_1', T_2 \blacktriangleright; H}\ \textbf{D-ParAS1} \qquad \frac{T_2 \Rightarrow_{H::P} T_2'}{H; \blacktriangleleft T_1, T_2 \blacktriangleright \to_P \blacktriangleleft T_1, T_2' \blacktriangleright; H}\ \textbf{D-ParAS2}$$

Figure 6. Dynamics of λ^{HP}

Lookup

$$\frac{H(\ell) = v}{H :: P; \ell \xrightarrow{lookup} v}\ \textbf{LU1} \qquad \frac{\ell \notin \mathrm{dom}(H) \qquad P; \ell \xrightarrow{lookup} v}{H :: P; \ell \xrightarrow{lookup} v}\ \textbf{LU2}$$

Task transitions $\quad \dfrac{H; e \to_P e'; H'}{H \cdot e \Rightarrow_P H' \cdot e'}\ \textbf{D-TaskStep}$

Figure 7. Auxiliary dynamic judgments of λ^{HP}

irreducible values of the language. Since small values consist only of locations, we use the term "value" to refer to a large value and "location" to refer to a small value.

In λ^{HP}, we organize memory as a tree of heaps and restrict expressions to access only part of the memory (along a path going up the hierarchy). A *heap*, written H (and variants), is simply a mapping from locations to (large) values. A *heap path* or simply a *path*, written as P (and variants), is a list or stack of heaps. A path represents a leaf-to-root path in the heap tree; it is the collection of heaps to which an expression has access.

Expressions of λ^{HP} include locations and (large) values, introduction and elimination forms for the standard types such as function application, projection and pair creation, and parallel tuples $\lhd e_1, e_2 \rhd$. Parallel tuples are evaluated sequentially by default, but may be *activated* for parallel evaluation. An *active* parallel tuple, $\blacktriangleleft T_1, T_2 \blacktriangleright$, consists of two parallel *tasks*, T_1 and T_2, which correspond to the components of the tuple.

A *task*, $H \cdot e$, consists of an expression e and a heap H. The heap H serves as the portion of the memory that is private to the expression e, i.e., H is visible only to the expression e (including subexpressions of e). The reader may wonder why we introduce tasks as a level of indirection in the syntax since they can take only one form ($H \cdot e$). When we formalize the memory manager, we will introduce another form of task, and the separation will make the presentation cleaner, so we adopt it now.

Statics. Figure 5 illustrates the static semantics of the calculus. Apart from the treatment of memory, the statics semantics of λ^{HP} is relatively standard. We first describe the typing of heaps and paths.

We ascribe type signatures to heaps and paths. A *signature* is a mapping from locations to types. The heap typing judgment $\vdash_{\Sigma'} H : \Sigma$ ascribes the signature Σ to heap H under path signature Σ', allowing the values in H to refer to locations in Σ'. The empty heap $\emptyset$ is given the empty signature $\cdot$. The signature for a non-empty heap H is defined inductively: each binding $\ell \mapsto v$ extends the signature with $\ell : \tau$ if v has type τ under the union of Σ' and the signature Σ of H. By typing the contents of a heap under locations from both the path Σ' and the heap Σ, we allow a value in the heap to refer to locations in both the private and shared parts of the memory.

The signature of a path P, which is a list of disjoint heaps, is the union of the signatures of the individual heaps making up the path.

The expression typing judgment has the form $\Gamma \vdash_\Sigma e : \tau$, indicating that e has type τ under variable context Γ and path signature Σ. Most of the typing rules seem standard: a location is well-typed if it is in the path signature and all the other expressions are typed in the usual way. There is an important point, however. An expression is typed only with respect to a path and not the whole memory, and the expression therefore may not have access to the whole memory. This point becomes evident when typing parallel tuples and tasks. The rule for active parallel tuples, S-ParA, requires that each constituent task be well-typed under the the same path signature. For each task $H \cdot e$, the typing rule ensures that e is well-typed under the union of the path signature, representing the shared memory, and the signature of H, representing the private memory.

Dynamics. The dynamics for λ^{HP}, shown in Figure 6, use a small-step transition judgment $H; e \to_P e'; H'$, which indicates that under heap H and path P, e steps to e' and produces new heap H'. In the judgment, H is the hierarchical heap that is private to this computation and will be used for lookup as well as allocation. The path P is the shared memory for e, consisting of the hierarchical heaps on the path from H to the root (excluding H).

The step relation uses the heaps on the path for lookups but not for allocation. In other words, the heaps on the path are read-only and are never modified; for this reason, we think of evaluation taking place in the context of a path, which we write as a subscript of the evaluation judgment. The impact of the distinction between heaps and paths can be seen in the rules for lookup and allocation. The auxiliary lookup judgment $P; \ell \xrightarrow{lookup} v$, defined in Figure 7, indicates that looking up the location ℓ in path P results in v. The rules are defined inductively on the path P: the rule looks first in the heap at the head of the path (the private heap) and, if ℓ is not found, the rest of the path is scanned recursively. The allocation rule D-ALLOC allocates a fresh location ℓ in the heap H, binds the location to v, and returns ℓ.

The rules for function and product types are standard for a call-by-value calculus with left-to-right evaluation. Function application and projection evaluate the subexpressions down to heap locations, look up the locations in the path, and perform the operation. For technical reasons, the function application rule requires that the location ℓ_2 be bound, but doesn't immediately use the value.

Sequential and inactive parallel tuples evaluate the first component to a location, followed by the second. If both components of a parallel tuple are evaluated fully before the tuple is activated, it changes into a pair $\langle \ell_1, \ell_2 \rangle$ (rule D-PARE). A parallel tuple may also be activated at any time, non-deterministically, by rule D-ACTIVATE. This models the possibility that another processor may begin evaluating one of the components in parallel. The activation rule changes the tuple into an active parallel tuple with two new tasks (each with a new, initially empty, private heap). Once a parallel tuple is activated, either component task may evaluate at any time using rules D-PARAS1 and D-PARAS2. (The rule D-PARAS2 does not require e_1 to be fully reduced to a location.)

The transition judgment for tasks, $T \Rightarrow_P T'$ indicates that, under path P, task T steps to T'. It doesn't include a private heap since that is internal to the task. Only the internal private heap may be modified, so P is read-only. Rule D-TASKSTEP (Figure 7) allows e to step using the expression step judgment.

When a component of a parallel tuple is stepped (the premises of D-PARAS1 and D-PARAS2), the heap for the tuple, H, is consed onto the current path and becomes part of the shared memory. This means that a heap becomes read-only when it is not at a leaf of the heap tree. When both components of an active parallel tuple are evaluated down to locations, the tuple steps to an ordinary pair using rule D-PARAE and merges the two local heaps into the parent heap.

3.2 Disentanglement

We state and prove one of this paper's key contributions: that purely functional programs written in λ^{HP} have the disentanglement property. Recall that we have defined disentanglement informally as the property that parallel tasks can only access data allocated by ancestors. We have formulated the semantics of λ^{HP} based on this property by restricting a task to access only the locations that are allocated in its own and its ancestors' heaps. To prove that an arbitrary λ^{HP} program has the disentanglement property, we first show that a well-typed program never gets stuck. This familiar type safety property guarantees that the restrictions placed on the memory accesses do not cause a program to get stuck (such as by accessing a memory location in a sibling or descendant heap).

We then show that a well-typed program yields the same result and exhibits the same termination behavior as it does under more traditional semantics. In other words, we show that, for a well-typed program, a run using hierarchical heaps and a run using conventional semantics either 1) produce the same result or 2) both do not terminate. This result also suffices to show that λ^{HP} programs are deterministic. We prove it by giving a standard operational semantics for terms which syntactically resemble λ^{HP} without heap locations or active parallel tuples. We define a transformation, called *flattening*, from arbitrary λ^{HP} expressions to such *flat* expressions, and show that the operational semantics of Figure 6 coincide with the storeless semantics up to flattening.

Type Safety. We first establish that well-typed terms do not become stuck by establishing progress and preservation. The proofs are straightforward, but two lemmas are required. The first states that if a path P is well-typed with a context that contains ℓ, then P contains a binding for ℓ with a value of the correct type.

Lemma 1. *If $P : \Sigma, \ell : \tau$, then there exists a value v such that $P; \ell \xrightarrow{lookup} v$ and $\Gamma \vdash_{\Sigma, \ell:\tau} v : \tau$.*

Proof. By construction. □

The second lemma is an important property of heaps, which states that a merged heap is well-typed.

Lemma 2. *If $H : \Sigma$ and $H' : \Sigma'$, then $(H \uplus H') : \Sigma, \Sigma'$.*

Proof. By induction on the derivation of $H' : \Sigma'$. If $H' = \emptyset$, then the result is trivial. Otherwise, $H' = H''[\ell \mapsto v]$ and $\Sigma' = \Sigma'', \ell : \tau$. By induction, $(H \uplus H'') : \Sigma, \Sigma''$. By the heap typing rules and weakening, $H \uplus H' = (H \uplus H'')[\ell \mapsto v] : \Sigma, \Sigma'', \ell : \tau = \Sigma, \Sigma'$. □

We now state and prove the preservation and progress lemmas.

Lemma 3 (Preservation)**.** *If $\Gamma \vdash_{\Sigma_2, \Sigma_1} e : \tau$ and $H : \Sigma_1$ and $P : \Sigma_2$ and $H; e \rightarrow_P e'; H'$, then $H' : \Sigma_1'$ where Σ_1' is an extension of Σ_1, and $\Gamma \vdash_{\Sigma_2, \Sigma_1'} e : \tau$.*

Proof. By induction on the derivation of $H; e \rightarrow_P e'; H'$. □

Lemma 4 (Progress)**.** *If $\cdot \vdash_{\Sigma_2, \Sigma_1} e : \tau$ and $H : \Sigma_1$ and $P : \Sigma_2$, then either e is a location or there exist e' and H' such that $H; e \rightarrow_P e'; H'$.*

Proof. By induction on the derivation of $\cdot \vdash_{\Sigma_2, \Sigma_1} e : \tau$. □

Correspondence with flattened semantics. Figure 8 defines the flattening transformation, which converts a λ^{HP} expression into a *flat* expression with no nested tasks and no free heap locations (but which is still a valid expression in the λ^{HP} syntax). The judgment $||e||_P \rightsquigarrow \hat{e}$ indicates that expression e flattens to $\hat{e}$ under path P. The transformation is defined inductively. There are two important rules to note. Locations are looked up in the path and the resulting value is recursively flattened (this will terminate since heaps and heap paths have no cycles). The flattened value is substituted for the location. The rule for active parallel tuples recursively flattens the two subexpressions, adding the local heaps to the path, and then converts the active tuple into an inactive tuple, thus flattening the hierarchy.

We make three observations about the flattening transformation:

- If e contains no active parallel tuples or heap locations (it may be a *source program* that has not yet begun executing), then $||e||_{\emptyset::[\,]} \rightsquigarrow e$.
- If $\ell \in \text{dom}(P)$ and $||\ell||_P \rightsquigarrow \hat{v}$, then $\hat{v}$ is simply the value at ℓ "lifted" so that it is closed with respect to the heap.
- Flattening is deterministic: if $||e||_P \rightsquigarrow \hat{e}_1$ and $||e||_P \rightsquigarrow \hat{e}_2$, then $\hat{e}_1 = \hat{e}_2$.

Flattened expressions may be run using the simple operational semantics of Figure 9. No path or heap is required, since these expressions have no heap locations. There is now no notion of "active" parallel tuples, and instead (formerly "inactive") parallel tuples may evaluate in parallel using rules F-PARS1 and F-PARS2. While these parallel evaluation rules allow for nondeterministic interleavings, evaluation is nevertheless deterministic. As usual, we show this using an intermediate result, the diamond lemma, which states that if an expression steps to two different expressions in one step, they can be "brought back together" in one step. We do not immediately prove confluence, the generalization of the diamond property to multi-step evaluation. This will be part of the disentanglement theorem.

Lemma 5 (Diamond Lemma)**.** *If $e \rightarrow e_1$ and $e \rightarrow e_2$, then there exists e' such that $e_1 \rightarrow e'$ and $e_2 \rightarrow e'$.*

$$\frac{}{\|\mathtt{n}\|_P \rightsquigarrow \mathtt{n}} \qquad \frac{\|e\|_P \rightsquigarrow \hat{e}}{\|\mathtt{fun}\ f\ x\ \mathtt{is}\ e\ \mathtt{end}\|_P \rightsquigarrow \mathtt{fun}\ f\ x\ \mathtt{is}\ \hat{e}\ \mathtt{end}}$$

$$\frac{\|e_1\|_P \rightsquigarrow \hat{e}_1 \qquad \|e_2\|_P \rightsquigarrow \hat{e}_2}{\|\langle e_1, e_2\rangle\|_P \rightsquigarrow \langle \hat{e}_1, \hat{e}_2\rangle}$$

$$\frac{P; \ell \xrightarrow{lookup} v \qquad \|v\|_P \rightsquigarrow \hat{v}}{\|\ell\|_P \rightsquigarrow \hat{v}} \qquad \frac{\|e_1\|_P \rightsquigarrow \hat{e}_1 \qquad \|e_2\|_P \rightsquigarrow \hat{e}_2}{\|e_1\ e_2\|_P \rightsquigarrow \hat{e}_1\ \hat{e}_2}$$

$$\frac{\|e\|_P \rightsquigarrow \hat{e}}{\|\mathtt{fst}(e)\|_P \rightsquigarrow \mathtt{fst}(\hat{e})} \qquad \frac{\|e\|_P \rightsquigarrow \hat{e}}{\|\mathtt{snd}(e)\|_P \rightsquigarrow \mathtt{snd}(\hat{e})}$$

$$\frac{\|e_1\|_P \rightsquigarrow \hat{e}_1 \qquad \|e_2\|_P \rightsquigarrow \hat{e}_2}{\|\triangleleft\, e_1, e_2 \triangleright\|_P \rightsquigarrow \triangleleft\, \hat{e}_1, \hat{e}_2 \triangleright} \qquad \frac{\|e_1\|_{H_1::P} \rightsquigarrow \hat{e}_1 \qquad \|e_2\|_{H_2::P} \rightsquigarrow \hat{e}_2}{\|\blacktriangleleft H_1 \cdot e_1, H_2 \cdot e_2 \blacktriangleright\|_P \rightsquigarrow \triangleleft\, \hat{e}_1, \hat{e}_2 \triangleright}$$

Figure 8. The flattening transformation

Proof. By induction on the derivations of $e \to e_1$ and $e \to e_2$. The only cases in which the two derivations instantiate different rules are (F-ParS1, F-ParS2) and (F-ParS2, F-ParS1). These are symmetric, so we consider the first one. In this case, $\triangleleft\, e_1, e_2 \triangleright \to \triangleleft\, e_1', e_2 \triangleright$ and $\triangleleft\, e_1, e_2 \triangleright \to \triangleleft\, e_1, e_2' \triangleright$, where $e_1 \to e_1'$ and $e_2 \to e_2'$. Let $e' = \triangleleft\, e_1', e_2' \triangleright$. By rule F-ParS2, $\triangleleft\, e_1', e_2 \triangleright \to e'$ and by rule F-ParS1, $\triangleleft\, e_1, e_2' \triangleright \to e'$. □

We prove two lemmas regarding the flattening operation which will be necessary in the disentanglement proof. First, flattening commutes with substitution.

Lemma 6. *If $\|v\|_P \rightsquigarrow \hat{v}$ and $\|e\|_P \rightsquigarrow \hat{e}$, then $\|[v/x]e\|_P \rightsquigarrow [\hat{v}/x]\hat{e}$.*

Proof. By induction on the derivation of $\|e\|_P \rightsquigarrow \hat{e}$. □

Next, the flattening transformation is unaffected by changing the structure of the path (as long as existing bindings are preserved) or by adding bindings.

Lemma 7. *Suppose $dom(H_1) \cap dom(P) = \emptyset$ and $dom(H_2) \cap dom(H_1 :: P) = \emptyset$.*

1. *If $\|e\|_{H_2::H_1::P} \rightsquigarrow \hat{e}$ then $\|e\|_{(H_2 \uplus H_1)::P} \rightsquigarrow \hat{e}$.*
2. *If $\|e\|_{H_1::P} \rightsquigarrow \hat{e}$ then $\|e\|_{(H_2 \uplus H_1)::P} \rightsquigarrow \hat{e}$.*

Proof. Both parts are by induction on the flattening derivation. The interesting case in both parts is the case for locations. (1) If $H_2 :: H_1 :: P; \ell \xrightarrow{lookup} v$, then either $H_1(\ell) = v$ or $H_2(\ell) = v$ or $P; \ell \xrightarrow{lookup} v$. In any of these cases, $H_2 \uplus H_1 :: P; \ell \xrightarrow{lookup} v$. By induction, $\|v\|_{H_2 \uplus H_1::P} \rightsquigarrow \hat{e}$.
(2) If $H_1 :: P; \ell \xrightarrow{lookup} v$, then either $H_1(\ell) = v$ or $P; \ell \xrightarrow{lookup} v$, so $H_2 \uplus H_1 :: P; \ell \xrightarrow{lookup} v$. By induction, $\|v\|_{H_2 \uplus H_1::P} \rightsquigarrow \hat{e}$. □

We now show that the hierarchical semantics of Figure 6 and the "flattened" operational semantics of Figure 9 can simulate each other. Lemma 8 shows one direction: a step of hierarchical evaluation can be simulated by zero or one steps of flattened evaluation (possibly zero since flattened evaluation does not have to perform allocation or activation). Conversely, Lemma 9 shows that a single step of flattened evaluation can be simulated by one or more steps of λ^{HP} evaluation. This situation is shown graphically in Figure 10.

Lemma 8. *Suppose that $H; e \to_P e'; H'$ and $\|e\|_{H::P} \rightsquigarrow \hat{e}$. Then either $\|e'\|_{H'::P} \rightsquigarrow \hat{e}$ or $\|e'\|_{H'::P} \rightsquigarrow \hat{e}'$ and $\hat{e} \to \hat{e}'$.*

Proof. By induction on the derivation of $H; e \to_P e'; H'$. □

Lemma 9. *Suppose that $\hat{e} \to \hat{e}'$ and $\|e\|_{H::P} \rightsquigarrow \hat{e}$. Then there exist H' and e' such that $H; e \to_P^* e'; H'$ and $\|e'\|_{H'::P} \rightsquigarrow \hat{e}'$.*

Proof. By induction on the derivation of $\hat{e} \to \hat{e}'$. □

Disentanglement. Finally, we state and prove the disentanglement theorem, which formalizes the two properties (type safety and correspondence with flattened semantics) outlined at the beginning of this section. If a source program e is well-typed and it evaluates to e', then (1) e' is not stuck and (2) for any flattened evaluation of e to $\hat{e}'$ (e' and $\hat{e}'$ need not be related), e' and $\hat{e}'$ may be brought back together in the style of confluence.

Theorem 1 (Disentanglement)**.** *Suppose e is a source program with no active parallel tuples. If $\cdot \vdash_{\cdot} e : \tau$ and $\emptyset; e \to_{[\,]}^* e'; H'$, then*

1. *Either e' is a location or there exists e'' such that*
$$H'; e' \to_{[\,]} e''; H''$$
2. *If $e \to^* \hat{e}'$, then there exist H'' and e'' and $\hat{e}''$ such that*
$$H'; e' \to_{[\,]}^* e''; H''$$
and $\hat{e}' \to^* \hat{e}''$ and $\|e''\|_{H''::[\,]} \rightsquigarrow \hat{e}''$.

Proof. 1. We show by induction on the derivation of $\emptyset; e \to_{[\,]}^* e'; H'$ that $\vdash_{\cdot} H' : \Sigma$ for some Σ and that $\cdot \vdash_\Sigma e' : \tau$. From this, the result follows from Lemma 4. If $e' = e$ and $H' = \emptyset$, then $\vdash_{\cdot} H' : \cdot$ and the typing result on e' was assumed. Otherwise, $\emptyset; e \to_{[\,]}^* e_0; H_0$ and $H_0; e_0; \to_{[\,]} e'; H'$. By induction, there exists Σ_0 such that $\vdash_{\cdot} H_0 : \Sigma_0$ and $\cdot \vdash_{\Sigma_0} e_0 : \tau$. The typing result for e' follows from Lemma 3.

2. We have $\|e\|_{\emptyset::[\,]} \rightsquigarrow e$. By inductive application of Lemma 8, we have $e \to^* \hat{e}_1'$ and $\|e'\|_{H'::[\,]} \rightsquigarrow \hat{e}_1'$. By a standard inductive application of Lemma 5, there exists $\hat{e}''$ such that $\hat{e}' \to^* \hat{e}''$ and $\hat{e}_1' \to^* \hat{e}''$. By an inductive application of Lemma 9 to the latter derivation, there exist e'' and H'' such that $H'; e' \to_{[\,]}^* e''; H''$ and $\|e''\|_{H''::[\,]} \rightsquigarrow \hat{e}''$.

□

Part (2) of Theorem 1, as well as its proof, is illustrated graphically in Figure 11. This statement is surprisingly powerful. In particular, it immediately implies that if e terminates with flattened semantics, it terminates with hierarchical semantics, and vice versa, that both semantics are deterministic and that they both produce the same result up to flattening. Each of these statements is simply an instantiation of Theorem 1.

Corollary 1. *1. If $e \to^* \hat{v}$, then there exist ℓ and H such that $\emptyset; e \to_{[\,]}^* \ell; H$ and $\|\ell\|_{H::[\,]} \rightsquigarrow \hat{v}$.*

2. *If $\emptyset; e \to_{[\,]}^* \ell; H$, then there exists $\hat{v}$ such that $e \to^* \hat{v}$ and $\|\ell\|_{H::[\,]} \rightsquigarrow \hat{v}$.*
3. *If $\emptyset; e \to_{[\,]}^* \ell; H$ and $e \to^* \hat{v}$, then $\|\ell\|_{H::[\,]} \rightsquigarrow \hat{v}$.*
4. *If $e \to^* \hat{v}_1$ and $e \to^* \hat{v}_2$, then $\hat{v}_1 = \hat{v}_2$.*
5. *If $\emptyset; e \to_{[\,]}^* \ell_1; H_1$ and $\emptyset; e \to_{[\,]}^* \ell_2; H_2$, then there exists $\hat{v}$ such that $\|\ell_1\|_{H_1::[\,]} \rightsquigarrow \hat{v}$ and $\|\ell_2\|_{H_2::[\,]} \rightsquigarrow \hat{v}$.*

3.3 Hierarchical Garbage Collection

We describe a hierarchical garbage collector, dubbed HGC, that takes advantage of disentanglement to perform concurrent garbage collection at the granularity of tasks. The collector HGC can collect any heap in the heap hierarchy by stopping the task that owns the heap along with its descendant tasks, while all other (non-descendant) tasks continue executing. Any number of independent heaps, which are not descendants and ancestors of each other, can be collected in parallel. The collector HGC is thus able to perform garbage collection without incurring additional synchronization

$$\frac{}{(\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end})\ v \to [\texttt{fun}\ f\ x\ \texttt{is}\ e\ \texttt{end}, v/f, x]e}\ \textbf{F-AppE} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \textbf{F-AppS1} \qquad \frac{e_2 \to e_2'}{(\texttt{fun}\ f\ x\ \texttt{is}\ e_1\ \texttt{end})\ e_2 \to (\texttt{fun}\ f\ x\ \texttt{is}\ e_1\ \texttt{end})\ e_2'}\ \textbf{F-AppS2}$$

$$\frac{}{\lhd v_1, v_2 \rhd \to \langle v_1, v_2\rangle}\ \textbf{F-ParE} \qquad \frac{e_1 \to e_1'}{\lhd e_1, e_2 \rhd \to \lhd e_1', e_2 \rhd}\ \textbf{F-ParS1} \qquad \frac{e_2 \to e_2'}{\lhd e_1, e_2 \rhd \to \lhd e_1, e_2' \rhd}\ \textbf{F-ParS2}$$

$$\frac{e_1 \to e_1'}{\langle e_1, e_2\rangle \to \langle e_1', e_2\rangle}\ \textbf{F-PairS1} \qquad \frac{e_2 \to e_2'}{\langle v_1, e_2\rangle \to \langle v_1, e_2'\rangle}\ \textbf{F-PairS2}$$

$$\frac{}{\texttt{fst}(\langle v_1, v_2\rangle) \to v_1}\ \textbf{D-FstE} \qquad \frac{e \to e'}{\texttt{fst}(e) \to \texttt{fst}(e')}\ \textbf{D-FstS} \qquad \frac{}{\texttt{snd}(\langle v_1, v_2\rangle) \to v_2}\ \textbf{D-SndE} \qquad \frac{e \to e'}{\texttt{snd}(e) \to \texttt{snd}(e')}\ \textbf{D-SndS}$$

Figure 9. The dynamics for the flattened semantics.

$$\begin{tikzcd}
{} \arrow[d, dashed, "\text{Lem. 8}"'] & e_1 \arrow[r] \arrow[d, squiggly] & e_2 \arrow[r] \arrow[d, squiggly] & e_3 \arrow[r] \arrow[dl, squiggly] & e_4 \arrow[r] \arrow[d, squiggly] & e_5 \arrow[r] \arrow[dl, squiggly] & e_6 \arrow[dll, squiggly] & {} \\
{} & \hat{e}_1 \arrow[r] & \hat{e}_2 \arrow[rr] & & \hat{e}_3 & & & {} \arrow[u, dashed, "\text{Lem. 9}"']
\end{tikzcd}$$

Figure 10. An illustration of the relationship between hierarchical and flattened evaluation. Straight lines are (hierarchical or flattened) evaluation and curved lines indicate the flattening transformation.

$$\begin{tikzcd}
& e \arrow[dl, "*"'] \arrow[dr, dotted, "\text{Lem. 8}"] & & \emptyset; e \arrow[ll, squiggly, "\text{reflexivity}"'] \arrow[dr, "*"] & \\
\hat{e}' \arrow[dr, dashed, "*"'] & \text{Lem. 5} & \hat{e}_1 \arrow[dl, dotted] & & H'; e \arrow[ll, dotted, "\text{Lem. 8}"'] \arrow[dl, dashed, "*\ \text{Lem. 9}"] \\
& \hat{e}'' & & H'', e'' \arrow[ll, dashed, "\text{Lem. 9}"] &
\end{tikzcd}$$

Figure 11. Theorem 1 and its proof. Assumptions are shown by solid black lines, results of the theorem by dashed black lines and intermediate results in the proof by dotted gray lines.

$$\textit{Tasks} \quad T \quad ::= \quad H \cdot e \mid \langle H; S; H; F\rangle \cdot [e]$$

$$\frac{\vdash_{\Sigma'} F(H_f) \uplus H_t : \Sigma \qquad \Gamma \vdash_{\Sigma,\Sigma'} e : \tau \qquad S = \mathsf{FL}(H_t \cdot e)}{\Gamma \vdash_{\Sigma'} \langle H_f; S; H_t; F\rangle \cdot [e] : \tau}\ \textbf{S-GCTask}$$

$$\frac{T \to_{\text{GC}} T'}{T \Rightarrow_P T'}\ \textbf{D-GCStep}$$

$$\frac{\begin{array}{c} S' = S \cup (\mathsf{FL}(F'(v)) \setminus (\text{dom}(H_t) \uplus \{\ell'\})) \qquad \ell' \text{ fresh} \\ F' = F[\ell \mapsto \ell'] \qquad H_t' = [\ell \mapsto \ell'](H_t[\ell \mapsto F(v)]) \end{array}}{\begin{array}{c} \langle H_f[\ell \mapsto v]; S \uplus \{\ell\}; H_t; F\rangle \cdot [e] \to_{\text{GC}} \\ \langle H_f; S'; H_t'; F'\rangle \cdot [[\ell \mapsto \ell'](e)] \end{array}}\ \textbf{HGC-Copy}$$

$$\frac{}{H \cdot e \to_{\text{GC}} \langle H; \mathsf{FL}(e); \emptyset; \emptyset\rangle \cdot [e]}\ \textbf{HGC-StartGC}$$

$$\frac{\text{dom}(H_f) \cap S = \emptyset}{\langle H_f; S; H_t; F\rangle \cdot [e] \to_{\text{GC}} H_t \cdot e}\ \textbf{HGC-EndGC}$$

Figure 12. Syntax, Statics and Dynamics for garbage collection

between the mutator and the collector. The main result of this subsection is the proof of correctness for HGC.

We formalize the outlined hierarchical garbage collector for λ^{HP}. The collector we formalize is a semispace copying collector, though other garbage collection algorithms could be used as well. Since the structure of a λ^{HP} program already contains hierarchical tasks and heaps, formalizing the collection algorithm requires few changes to the semantics. In fact, we simply need to introduce a new form for tasks to represent a task which is locked for collection, and several dynamic rules to perform garbage collection on tasks.

Our approach in formalizing the collection algorithm is inspired by Morrisett et al. [40]. While they analyze only stop-the-world collectors for sequential languages and their formalisms do not capture some of the details that ours does, we borrow their language-based approach for high-level descriptions of collection algorithms as well as much of their notation.

Syntax. The first change is to introduce a new form of task, as shown at the top of Figure 12. The new form, $\langle H_f; S; H_t; F\rangle \cdot [e]$, represents an expression e which is locked to perform garbage collection. The tuple represents a heap which is in an intermediate state of collection, in which some locations have already been copied to the *to-heap* H_t and the rest remain in the *from-heap* H_f. The second component, S, is the *scan set* or *frontier*, the set of locations that have been seen by the collector but not yet copied. The final component, F, is a *forwarding map*. It is a finite map whose domain and codomain both consist of heap locations, and it maps locations formerly bound in H_f to the locations in H_t to which they have been copied. This allows us to make sense of values in H_f, which may refer to locations which have already been renamed and copied to H_t. The forwarding map models *forwarding pointers*, which are commonly used to perform this function in practical copying garbage collectors. For notational convenience, we implicitly extend the domain of forwarding maps by mapping locations that are not in the domain to themselves:

$$F(\ell) := \begin{cases} F(\ell) & \ell \in \text{dom}(F) \\ \ell & \ell \notin \text{dom}(F). \end{cases}$$

We will also use the notation $F(e)$ to denote an expression in which every free location ℓ has been replaced with $F(\ell)$. We extend F to operate on heaps in the following way:

$$F(H) := \{F(\ell) \mapsto F(v) \mid \ell \mapsto v \in H\}$$

and on signatures in the following way:

$$F(\Sigma) := \{F(\ell) \mapsto F(v) \mid \ell \mapsto v \in \Sigma\}$$

A particularly common usage of the forwarding map is the notation $[\ell \mapsto \ell'](e)$ (and similar for heaps and signatures), in which e is forwarded with the singleton forwarding map forwarding ℓ to ℓ'. In addition, we use $F[\ell \mapsto \ell']$ to indicate the extension of the map F with a mapping from ℓ to ℓ'.

Statics. The typing rule for the new form of task is given in Figure 12. The rule uses the rules for typing heaps to assign a signature to the concatenation of the from- and to-heaps. Since values in H_f might refer to locations that have already been copied, we forward the values of H_f before concatenating with H_t. The expression must be well-typed under the concatenation of the global signature Σ' and this new signature Σ. In addition, the typing rule also enforces a key invariant of garbage collection: that the scan set S is equal to the set $\mathsf{FL}(H_t \cdot e)$ of locations that appear free in either e or a binding of H_t and are not bound in H_t. Formally,

$$\mathsf{FL}(H_t \cdot e) \triangleq \left(\mathsf{FL}(e) \cup \bigcup_{\ell \in \mathrm{dom}(H_t)} \mathsf{FL}(H_t(\ell))\right) \setminus \mathrm{dom}(H_t)$$

Dynamics. Note that if a collection is in progress and the task is of the form $H \cdot [e]$, no rule allows e to step, so no changes are required to the semantics to ensure that e is in fact locked during collection. Instead, we introduce a new transition rule for tasks, D-GCStep, which allows the task to proceed with collection. The auxiliary judgment $T \to_{\mathrm{GC}} T'$ indicates that a step of garbage collection transforms T into T'.

Our garbage collection algorithm uses a small-step semantics, where one step (atomically) copies a single location. A full garbage collection will therefore involve many individual steps, which can be freely interleaved with evaluation or collection on other processors. Figure 12 gives the transition rules for collection. Rule HGC-StartGC locks the task for collection and sets up the tuple by using the existing heap, H, as the from-heap, initializing the to-heap and the forwarding map to be empty and the scan set to be $\mathsf{FL}(e)$, the roots of the expression. Rule HGC-Copy performs one step of collection. The rule takes a location from S which is present in H_f, copies its value v, and adds to S the locations of v which have not already been copied. Rule HGC-EndGC requires that $\mathrm{dom}(H_f) \cap S = \emptyset$, i.e. that there are no remaining locations to be collected. If this is the case, e is unlocked and the to-heap becomes the new heap.

Rule HGC-StartGC can apply to a task nondeterministically at any time, and so our semantics does not specify when collections can or should occur. Also note that a collection in the semantics can collect any heap in the hierarchy at any time. An entire subtree is locked, but only the heap at the root of that subtree is collected. However, this property also means that, because our safety proofs are parametrized over any possible evaluation, our theorems will allow for a variety of possible policies for both when to trigger a collection and what subtrees to collect at what times. For example, the policy for collecting a subtree described in Section 2 may be reproduced in the semantics by locking and collecting all leaves of a subtree, then immediately locking and collecting their parents, and so on until the root of the subtree is collected. Because any collection occurs over a number of single steps, unrelated tasks can execute or perform garbage collection while a collection is taking place. Thus, we effectively model parallel collection.

3.3.1 Safety and Correctness Proofs

We will now proceed to prove that our collection algorithm is memory safe, that is, it never frees a location that will be dereferenced later. Because our calculus is typed and dereferencing a freed location is a stuck state, memory safety is an immediate corollary of type safety. Thus, we only need to update the progress and preservation proofs to handle the new rules. Several lemmas are required. One important lemma is that garbage collection preserves typing, in that if e is well-typed under heap $H \uplus H'$ and H is discarded by garbage collection, e is also well-typed under heap H'. The signature of the higher levels of the hierarchy, notated Σ_1 in the lemma, does not change. Both H and H' are typed under Σ_1. We also allow the expression to refer to additional locations in another signature, Σ_2. This contains locations at levels *lower* than the level being collected. In using this lemma to prove preservation, we will only be interested in the case where Σ_2 is empty because we want to show type preservation of the expression whose heap is being collected. However, Σ_2 is necessary for the proof of the lemma, when inducting on expressions containing nested heaps.

Lemma 10. *If* $\vdash_{\Sigma_1} H \uplus H' : \Sigma$ *and* $\vdash_{\Sigma_1} H' : \Sigma'$ *and* $\Gamma \vdash_{\Sigma_1,\Sigma,\Sigma_2} e : \tau$ *and* $\mathsf{FL}(H' \cdot e) \cap \mathit{dom}(H) = \emptyset$*, then* $\Gamma \vdash_{\Sigma_1,\Sigma',\Sigma_2} e : \tau$.

Proof. By induction on the derivation of $\Gamma \vdash_{\Sigma_1,\Sigma,\Sigma_2} e : \tau$. □

Lemma 11 states that typing of an expression is preserved by performing renamings on both the expression and signature.

Lemma 11. *If* $\Gamma \vdash_\Sigma e : \tau$*, then* $\Gamma \vdash_{[\ell \mapsto \ell'](\Sigma)} [\ell \mapsto \ell'](e) : \tau$.

Proof. By induction on the derivation of $\Gamma \vdash_\Sigma e : \tau$. □

We now restate the preservation and progress theorems and prove them for the full language, including garbage collection.

Theorem 2 (Preservation). *If* $\Gamma \vdash_{\Sigma_2,\Sigma_1} e : \tau$ *and* $H : \Sigma_1$ *and* $P : \Sigma_2$ *and* $H; e \to_P e'; H'$*, then* $H' : \Sigma_1'$ *where* Σ_1' *is an extension of* Σ_1*, and* $\Gamma \vdash_{\Sigma_2,\Sigma_1'} e : \tau$.

Proof. We consider the cases for garbage collection $T \to_{\mathrm{GC}} T'$ where, without loss of generality, $e = \blacktriangleleft T, T_2 \blacktriangleright$ and $\tau = \tau_1 \times \tau_2$.

- HGC-StartGC. Then $T = H \cdot e_1$. By typing inversion, $H : \Sigma$ and $\Gamma \vdash_{\Sigma_2,\Sigma_1,\Sigma} e_1 : \tau_1$. We have that $\emptyset(H) \uplus \emptyset = H : \Sigma$, so by task typing, $\Gamma \vdash_{\Sigma_2,\Sigma_1} \langle H; \mathsf{FL}(e_1); \emptyset; \emptyset\rangle \cdot [e_1] : \tau_1$. By definition, $\mathsf{FL}(e_1) = \mathsf{FL}(\emptyset \cdot e_1)$.
- HGC-Copy. Then $T = \langle H_t[\ell \mapsto v]; S \uplus \{\ell\}; H_t; F\rangle \cdot [e_1]$. By typing inversion, $F(H_f[\ell \mapsto v]) \uplus H_t : \Sigma$ and $\Gamma \vdash_{\Sigma_2,\Sigma_1,\Sigma} e_1 : \tau_1$. We have $F'(H_f) \uplus H_t' = [\ell \mapsto \ell'](F(H_f) \uplus H_t[\ell \mapsto F(v)]) = [\ell \mapsto \ell'](F(H_f[\ell \mapsto v]) \uplus H_t) : [\ell \mapsto \ell'](\Sigma)$ and, since $[\ell \mapsto \ell'](\Sigma_2, \Sigma_1, \Sigma) = \Sigma_2, \Sigma_1, [\ell \mapsto \ell'](\Sigma)$, Lemma 11 gives $\Gamma \vdash_{\Sigma_2,\Sigma_1,[\ell\mapsto\ell'](\Sigma)} [\ell \mapsto \ell'](e_1) : \tau_1$. It is a straightforward exercise to show that $S' = \mathsf{FL}(H_t \cdot [\ell \mapsto \ell'](e_1))$.
- HGC-EndGC. Then $T = \langle H_f; S; H_t; F\rangle \cdot [e]$. By typing inversion, $F(H_f) \uplus H_t : \Sigma$ and $\Gamma \vdash_{\Sigma_2,\Sigma_1,\Sigma} e_1 : \tau_1$ and $S = \mathsf{FL}(H_t \cdot e_1)$, so $\mathsf{FL}(H_t \cdot e_1) \cap \mathrm{dom}(H_f) = \emptyset$. If $H_t : \Sigma_t$, by Lemma 10, $\Gamma \vdash_{\Sigma_2,\Sigma_1,\Sigma_t} e_1 : \tau_1$. By the task typing rules, we have $\Gamma \vdash_{\Sigma_2,\Sigma_1} H_t \cdot e_1 : \tau_1$.

□

Theorem 3 (Progress). *If* $\cdot \vdash_{\Sigma_2,\Sigma_1} e : \tau$ *and* $H : \Sigma_1$ *and* $P : \Sigma_2$*, then either e is a location or there exist* e' *and* H' *such that* $H; e \to_P e'; H'$.

Proof. We consider the case of $e = \blacktriangleleft T_1, T_2 \blacktriangleright$ where either $T_1 = \langle H_f; S; H_t; F\rangle \cdot [e_1]$ or $T_2 = \langle H_f'; S'; H_t'; F'\rangle \cdot [e_2]$ or both. Without loss of generality, suppose $T_1 = \langle H_f; S; H_t; F\rangle \cdot [e_1]$. If $\mathrm{dom}(H_f) \cap S = \emptyset$, then apply HGC-EndGC. If $\ell \in \mathrm{dom}(H_f) \cap S$, then apply HGC-Copy. In either case, T_1 steps with D-GCStep. □

Together, the progress and preservation theorems imply the standard type safety property: a well-typed term will not become stuck. Formally, if $\cdot \vdash_\cdot e : \tau$, then, for some ℓ and H, we have $\emptyset; e \to^*_{[\,]} \ell; H$. In particular, evaluation of e will never dereference a freed memory location. This completes the proof of memory safety. This is only half of the correctness of the garbage collector; we have not yet shown that collection is meaning-preserving, i.e. that the behavior of a garbage-collected program is identical to an execution without collection. This property is straightforward to prove using the same flattening machinery that was introduced in the previous section. The proof simply requires extending the definitions and proofs related to flattening to account for garbage collection. We first extend the definition of flattening to cover tasks which are in the process of collection:

$$\frac{\|e_1\|_{(F(H_f)\uplus H_t)::P} \rightsquigarrow \hat{e}_1 \qquad \|e_2\|_{(F'(H'_f)\uplus H'_t)::P} \rightsquigarrow \hat{e}_2}{\left\| \blacktriangleleft \langle H_f; S; H_t; F\rangle \cdot e_1, \langle H'_f; S'; H'_t; F'\rangle \cdot e_2 \blacktriangleright \right\|_P \rightsquigarrow \triangleleft\, \hat{e}_1, \hat{e}_2 \,\triangleright}$$

The two other rules for parallel tuples in which one task is of the form $H \cdot e$ and the other is of the form $\langle H_f; S; H_t; F\rangle \cdot [e]$ are omitted but are defined in the natural way. We also require two more lemmas regarding flattening, which will be used in the extension of the simulation proof.

Lemma 12. *If $dom(H) \cap dom(P) = \emptyset$ and $dom(F) \subset dom(H)$ and $\|e\|_{H::P} \rightsquigarrow \hat{e}$ then $\|F(e)\|_{F(H)::P} \rightsquigarrow \hat{e}$.*

Proof. By induction on the derivation of $\|e\|_{H::P} \rightsquigarrow \hat{e}$. The interesting case is the case for a location ℓ. □

Lemma 13. *If $dom(H_1) \cap \mathsf{FL}(e) = \emptyset$ and $\|e\|_{(H_1 \uplus H_2)::P} \rightsquigarrow \hat{e}$ then $\|e\|_{H_2::P} \rightsquigarrow \hat{e}$.*

Proof. By induction on the derivation of $\|e\|_{(H_1 \uplus H_2)::P} \rightsquigarrow \hat{e}$. The interesting case is the case for a location ℓ. □

Lemma 14 is simply a restatement of Lemma 8, which showed that a step of the hierarchical semantics can be simulated by zero or one steps of the flattened semantics. The lemma now includes garbage collection steps.

Lemma 14. *Suppose that $H; e \rightarrow_P e'; H'$ and $H :: P : \Sigma$ and $\Gamma \vdash_\Sigma e : \tau$ $\|e\|_{H::P} \rightsquigarrow \hat{e}$. Then either $\|e'\|_{H'::P} \rightsquigarrow \hat{e}$ or $\|e'\|_{H'::P} \rightsquigarrow \hat{e}'$ and $\hat{e} \rightarrow \hat{e}'$.*

Proof. The new cases are those that instantiate D-ParAS1 or D-ParAS2 with one of the GC rules. We show that if $T \rightarrow_{\mathrm{GC}} T'$, then for any T_2, if $\|\blacktriangleleft T, T_2 \blacktriangleright\|_{H::P} \rightsquigarrow \hat{e}$ then $\|\blacktriangleleft T', T_2 \blacktriangleright\|_{H::P} \rightsquigarrow \hat{e}$.

- HGC-Copy. Then $T = \langle H_f[\ell \mapsto v]; S \uplus \{\ell\}; H_t; F\rangle \cdot [e]$. Let
$$\begin{array}{rcl} H_T & = & F(H_f[\ell \mapsto v]) \uplus H_t \\ H'_T & = & F'(H_f) \uplus [\ell \mapsto \ell'](H_t[\ell \mapsto F(v)]) \end{array}$$
We have $\|e_1\|_{H_T::H::P} \rightsquigarrow \hat{e_1}$ and wish to show $\|e_1\|_{H'_T::H::P} \rightsquigarrow \hat{e_1}$. This follows from Lemma 12 since
$$\begin{array}{rcl} [\ell \mapsto \ell](H_T) & = & [\ell \mapsto \ell'](F(H_f[\ell \mapsto v]) \uplus H_t) \\ & = & [\ell \mapsto \ell'](F(H_f) \uplus H_t[\ell \mapsto F(v)]) \\ & = & F'(H_f) \uplus [\ell \mapsto \ell'](H_t[\ell \mapsto F(v)]) \\ & = & H'_T \end{array}$$
- HGC-StartGC. Then $T = H_1 \cdot e_1$ and $\|e_1\|_{H_1::H::P} \rightsquigarrow \hat{e_1}$. If $F = \emptyset$, then $F(H_1) \uplus \emptyset = H_1$, so $\|\blacktriangleleft \langle H_1; \mathsf{FL}(e_1); \emptyset; \emptyset\rangle \cdot e_1, T_2 \blacktriangleright\|_{H::P} \rightsquigarrow \hat{e}$.
- HGC-EndGC. Then $T = \langle H_f; S; H_t; F\rangle \cdot e_1$ and
$$\|e_1\|_{F(H_f)\uplus H_t::H::P} \rightsquigarrow \hat{e_1}$$
By typing, $S = \mathsf{FL}(H_t \cdot e_1)$, so $\mathrm{dom}(H_f) \cap \mathsf{FL}(e_1) = \emptyset$ and Lemma 13 gives $\|e_1\|_{H_t::H::P} \rightsquigarrow \hat{e_1}$ and $\|\blacktriangleleft H_t \cdot e_1, T_2 \blacktriangleright\|_{H::P} \rightsquigarrow \hat{e}$.

□

Finally, Theorem 4 states the correctness of HGC: the hierarchical semantics produces a value from an expression if and only if the flattened semantics produces a value from the same expression, and the two values agree up to flattening. It was earlier shown that the flattened semantics is consistent with the non-collected hierarchical semantics, so this is sufficient to show that garbage collection does not change the meaning of a hierarchical program.

Theorem 4. *Let e and τ be such that $\cdot \vdash_\cdot e : \tau$.*

- *If $e \rightarrow^* \hat{v}$, then there exist ℓ and H such that $\emptyset; e \rightarrow^*_{[\,]} \ell; H$ and $\|\ell\|_{H::[\,]} \rightsquigarrow \hat{v}$.*
- *If $\emptyset; e \rightarrow^*_{[\,]} \ell; H$, then there exists $\hat{v}$ such that $e \rightarrow^* \hat{v}$ and $\|\ell\|_{H::[\,]} \rightsquigarrow \hat{v}$.*

Proof. These are simply items 1 and 2 of Corollary 1, and are instances of Theorem 1, which is easily proven for the full language including garbage collection by using Lemma 14 instead of Lemma 8 in the proof. □

4. The Design

There are three primary challenges to the design of a practical implementation of the hierarchical semantics. These challenges primarily stem from the fact that a practical design must handle concurrency to realize parallelism.

- **Scheduling:** the semantics models scheduling using a nondeterministic interleaving of threads; a practical design must use a specific scheduler.
- **Algorithms for the heap hierarchy:** a practical design must use efficient and correct data structures and algorithms to represent the heap hierarchy.
- **Hierarchical garbage collection:** a practical design must ensure correct and performant garbage collection.

4.1 Scheduling

For this design, we use a work-stealing scheduler as described by Blumofe et al. [15]. This scheduler assigns every processor a double-ended queue, or "deque", containing tasks associated with that processor. As a computation spawns new tasks, they are pushed onto the bottom end of the processor's deque. Once a task is completed, the processor fetches the next task by popping the task from the bottom end of the deque. If the processor finds the deque to be empty, the processor steals a task from a randomly chosen victim processor by popping from the top of the victim's deque.

The semantics presented in Section 3.1 accounts for some of the scheduler actions, such as the creation of parallel tasks, by distinguishing between active and inactive parallel tuples. There is a subtle but a crucial difference, however, between the management of tasks in the semantics and in work stealing. The semantics allows a parallel pair to be activated at any time by creating a task for each component and by pairing each task with its own heap. In work stealing, however, task creation is a lazy and asynchronous operation: a task is created only when one of the components of a parallel tuple is stolen, as the other component continues to execute serially. This subtle difference creates a challenge for identifying the heap for a task.

For example, consider a processor Q executing a parallel tuple $\triangleleft\, e_1, e_2 \,\triangleright$. To execute the tuple, the processor starts executing the left component e_1. Sometime after e_1 is reduced to e'_1, e_2 may be stolen and converted to a task T_2 so that it can be executed by the thief processor. Since the algorithm has not converted e_1 to a task before e_2 was stolen, we don't have a heap for it, and thus don't know the objects allocated during the execution of e_1 thus far.

To solve this problem, we assign to each parallel pair of the form $\triangleleft\, e_1, e_2 \,\triangleright$ a *level* that corresponds to its nesting depth. For example, if the expression $e = \triangleleft\, e_1, e_2 \,\triangleright$ has level m, then e_1 and e_2 have level $m + 1$. We then use levels to identify the heap of a task by tracking the levels at which steals take place.

4.2 Hierarchical Heaps

In our design, the abstract heaps of the semantics are implemented using *pages* and *chunks*, described below. These smaller units of memory are collected in data structures which we call *superheaps*,

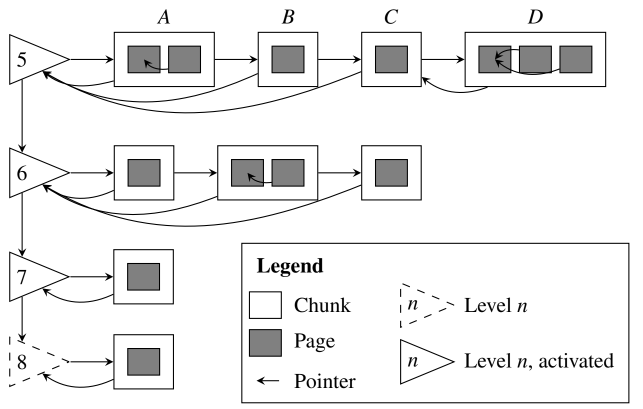

Figure 13. The structure of a superheap.

which implement the heap hierarchy. The structure of the pages, chunks and superheaps allow efficient allocation, lookup, and other required operations on heaps and the heap hierarchy.

Pages, Chunks, and Chunk Pool. To reduce contention at memory allocation, we partition memory into *chunks*, each of which is a contiguous block of memory. Each chunk is further divided into one or more contiguous *pages*. A page can coincide with a system-level page, typically 4KB, but can be set to any power of two bytes.

Processors allocate and free chunks from and to a shared *chunk pool*. Once a chunk is allocated, it is locally divided into individual objects without further communication with other processors. Prior work has proposed design techniques for such a data structure [21, 27, 38]. Typical designs maintain a number of free chunk lists, each of which may, for example, contain chunks of a particular size. A number of different allocation policies can be employed to maximize efficiency and minimize fragmentation of the global memory [10, 18].

Distinguishing between chunks and pages might seem redundant, but it is not. Our design uses pages to find the chunk and the heap to which a given object belongs to by storing this information in the metadata for each page. Since a page size is a fixed power of two, the start of the page containing a given object can be located by truncating the address of the object. Once the page is found, a simple look up suffices to find the relevant metadata.

Superheaps. In our design, we use superheaps to identify the heap of a task by using levels. A *superheap* is a collection of chunks, where each chunk is tagged with the level at which it is allocated. The computation starts with a root superheap. During the execution, each steal starts a superheap for the stolen task. Thus at any point in the execution, a processor works on one superheap. When a steal happens, we remember the level of the stolen task at the superheap of the victim processor by marking it as “activated.” A heap in the semantics corresponds to all chunks in one superheap belonging to an activated level and possibly a contiguous range of unactivated levels.

Figure 13 illustrates the structure of a superheap, where chunks at each level are drawn together at the same “level.” The levels 5, 6, and 7 are activated, since the scheduler has stolen the corresponding right tasks at these levels. Level 8 is not yet activated since its corresponding right task has not yet been activated and is still on the processor’s deque. Levels 5 and 6 each correspond to a heap of the semantics, and levels 7 and 8 together correspond to the local heap of the currently running task. We store various pointers in each chunk, such as a pointer to its level, and a pointer to the next chunk in its level to aid in the implementation of various operations.

Heap operations To realize the semantics, the implementation of heaps needs to support four key operations: creating heaps, merging two heaps, allocating a new location in a heap, and looking up the value of a location in a heap. The design allows creating a heap by marking the level for the stolen task in the victim processor as activated and creating a new superheap for the stolen task. Merging two heaps corresponds to simply unioning their chunks. The allocation operation is simply a bump allocation within the current chunk. When the chunk is full, a new one is fetched from the chunk pool and assigned a level. Looking up an object is simply a pointer dereference.

4.3 Garbage collection

The semantics allow for collection of any heap in the memory hierarchy. The heaps that are leaves of the heap hierarchy, however, are special in the sense that they can be collected without communicating with any other processor, since they are disentangled. In our design, we therefore distinguish between two forms of collection. Local collection performs garbage collection on a leaf in the heap hierarchy; non-local collection collects all of the heaps in a subtree of the hierarchy.

Local Collection. A processor starts a local collection by locking its deque, ensuring thus that no tasks can be stolen during collection. A steal of a task from the deque corresponds in the semantics to the activation of a parallel tuple contained in the current expression, so locking the expression in rule HGC-StartGC corresponds to locking the deque.

After locking its deque, the processor proceeds to run a standard algorithm such as Cheney (copying) collection [20] adapted for hierarchical heaps. Such a collection would scan the root set (the collecting processor’s stack) for all pointers within the scope of collection. A pointer is considered to be within scope if it points to an object residing in one of the levels being collected. All roots in scope of the collection are copied to a new collection of chunks. The in-scope check also appears in the semantics, since rule HGC-Copy only copies locations that appear in both the scan set and the heap being collected.

After all roots are copied, copy collection continues normally, level-by-level in descending order. Any pointers to objects in scope of the collection but in lower levels are copied and scanned once collection gets to that level. Note that by the disentanglement property, no pointers from lower levels to higher levels can exist, so once a level is completely scanned, it does not need to be revisited. Once copy collection completes, all chunks in the levels being collected are released to the chunk pool, the processor’s deque is unlocked and execution resumes (corresponding to transition rule HGC-EndGC).

Non-Local Collection. As specified in Section 3.3, the GC semantics allows non-deterministic collection of any heap in the hierarchy. Here, we describe a specific non-local GC algorithm that collects all heaps in a given subtree at each non-local collection. Other algorithms are also possible.

A non-local collection starts by first choosing a subtree of the heap hierarchy to collect and identifying the processors participating in the collection. All participating processors then synchronize; each locks its deque and starts collection. All other, non-participating, processors continue to work on their execution as their accessible data will not be affected by the collection.

Participating processors then, in parallel, scan their root sets for all pointers within the scope of collection, just like for local collection. These pointers are all copied to the corresponding to-space. After all the roots are copied, collection proceeds level-by-level in parallel across all superheaps involved in the collection. However, each superheap is collected sequentially on its own.

Therefore, once collection on all of a parent superheap's children is completed, one processor will continue collection on the parent superheap while the other processors remain idle. In this way, collection proceeds up the hierarchy of heaps until complete, at which point all the processors involved in the collection unlock their deques and resume execution.

Open issues. Our design leaves open several important questions: 1) when to garbage collect, 2) whether to perform a local or non-local collection and 3) if non-local, which heaps, or subtree, to collect. The answers to these questions are not as straightforward as in traditional collection algorithms, where relatively simple amortization arguments can guide them. We leave the investigation of these question to future work. In the implementation described below, we choose a simple technique adopted from traditional garbage collection literature.

5. Implementation and Evaluation

We describe a prototype implementation and preliminary evaluation of the proposed hierarchical memory manager.

5.1 Implementation

We implement our hierarchical memory manager on top of Daniel Spoonhower's parallel extension of the MLton compiler [47]. As a whole-program optimizing, high-performance compiler for Standard ML, MLton [39] offers an excellent starting point for parallel computing. Spoonhower extended MLton to support a number of parallelism primitives including nested (fork-join) parallelism and futures. Spoonhower implemented many different schedulers; we use his implementation of a standard work-stealing scheduler.

Our implementation, which we call `mlton-parmem`, closely follows the design described in Section 4 but of course has to consider many more details. One such detail is the treatment of stacks. MLton executes all code with a traditional call stack, which is allocated on the heap. We allocate computation threads' initial stacks in their associated heap. When the stack runs out of space, a new stack is allocated in the same heap at the current level of the computation. Upon first inspection, this model seems to blatantly violate disentanglement as a stack allocated in a lower level may have pointers in a higher level by virtue of computation at that level. However, since the stack cannot be pointed to from any object, there is no issue of having parallel reads from another processor. In addition, since the stack is explicitly scanned as a root set for collection, we keep track of all the entangled pointers at collection time.

Currently, our implementation only supports local collection, which collects unactivated levels in descending order until the first activated level. For example, in Figure 13, local collection will only collect level 8 as all lower levels (drawn higher in the figure) have been activated. Our algorithm for deciding when to collect is a straightforward adaptation of the corresponding algorithm from traditional GC literature. We initiate a local collection when the local heap becomes half full and we resize the heap to a small multiple (8 in our experiments) of the size of the live set.

5.2 Benchmarks

We evaluate performance under five benchmarks. Each benchmark is written in a purely functional style and in three different dialects of ML: Standard ML, our parallel extensions to it, dubbed "mlton-parmem", and Manticore.

All of our benchmarks use sequences as the core data structure. Sequences are implemented as weight-balanced trees with data elements at the leaves. This data structure allows for naturally expressing parallel computation over the sequences. The benchmarks use a manual approach to granularity control: any potentially parallel operation first checks if the length of the input sequence is less than or equal to a specified grain size. If so, the operation is carried out sequentially without creating parallel tasks. If the length of the input sequence is greater than the grain, the operation will create the parallel tasks. The grain size can be supplied at run-time.

The MemStress benchmark is a synthetic benchmark designed to heavily stress the memory manager by allocating a large amount of data and releasing it almost immediately. This is achieved by building, in parallel, an integer sequence (using the *tabulate* operation on sequences) where, at each element, a large list is sequentially allocated. The list's tabulation function is simply the identity function and the sequence's element is then either the first or second element of the list depending on whether the element index is even or odd. Note that the amount of computation done per allocation is very small, ensuring that the bulk of the work of the program lies in the allocation and collection of the data. The benchmark tabulates a 100,000 element sequence with granularity 100 and allocates a 10,000 element list per sequence element.

The Tabulate benchmark is designed to evaluate the performance of an embarrassingly parallel sequence tabulation operation. In this benchmark, a sequence is tabulated and every element is set to a Fibonacci number, calculated each time. In our benchmark, we tabulate a 100,000 element sequence with granularity 100. For each element, the 20th Fibonacci number is calculated sequentially using the classical recursive algorithm and stored.

The Raytracer benchmark is adapted from the raytracer benchmark written for the Manticore language [8]. It renders a 512px × 512px scene in parallel. The original program was written in ID [42] and does not use any special data structures to improve performance.

The SMVM benchmark performs a sparse matrix vector multiplication. The sparse matrix is represented in a typical fashion as a sequence of rows, where each row only contains the non-zero entries as an index-value pair. Multiplication is then done in parallel over the rows. Our benchmark performed the test on a $10,000 \times 10,000$ matrix with granularity 100.

The DMM benchmark performs a dense matrix matrix multiplication. The matrices are represented as a sequence of sequences. Multiplication is then done in parallel over the rows of the first matrix. Our benchmark performed the test on 500×500 matrices at the granularity of individual dot-products.

5.3 Measurements

For our experiments, we use a 64-core AMD machine with 128 gigabytes of memory, but due to technical limitations of the underlying MLton codebase, we are not able to utilize more than 32 cores. We use interleaved NUMA allocation in all of our experiments. For the grain sizes we use, our benchmarks create anywhere from several thousand to hundreds of thousands of tasks.

In addition to our `mlton-parmem` compiler, we run our benchmarks on the `manticore` compiler, the `mlton` compiler for Standard ML and Spoonhower's parallel extension of MLton, which we call `mlton-spoonhower`. The memory manager of `mlton-spoonhower` allows for parallel allocation, but performs stop-the-world sequential collection. We choose this compiler as a naïve baseline for comparing parallel garbage collectors.

The Manticore compiler for Parallel ML [28] offers support for an ML-like language extended with several paradigms for parallelism, including fork-join parallelism and speculation. Manticore uses an advanced NUMA-aware memory manager based on the Doligez-Leroy-Gonthier and Appel semi-generational collectors. It is a reasonably well-developed system with particular innovations on memory management and has been shown to deliver good performance [8, 11, 13, 28, 44]. Since it is based on ML, Manticore is a good basis for comparison for the approach suggested in this paper.

We use the `mlton` compiler timings as the sequential baseline for all our speedup calculations. We derive the sequential versions of our

	`mlton`	`mlton-spoonhower`				`mlton-parmem`				`manticore`			
Benchmark	T_s (s)	T_1 (s)	O	T_{32} (s)	S	T_1 (s)	O	T_{32} (s)	S	T_1 (s)	O	T_{32} (s)	S
MemStress	26.91	22.85	0.85	9.63	2.37	21.61	0.80	2.61	10.30	79.29	2.95	2.67	10.09
Tabulate	19.06	24.32	1.28	0.99	19.18	22.46	1.18	0.95	20.13	32.46	1.70	1.04	18.28
Raytracer	10.29	10.92	1.06	0.71	14.46	12.60	1.22	0.51	20.18	11.18	1.19	0.36	28.51
SMVM	157.48	197.14	1.25	13.85	11.37	194.83	1.24	7.48	21.04	286.04	1.82	46.31	3.40
DMM	61.62	91.44	1.48	6.07	10.15	92.91	1.51	4.09	15.05	100.04	1.62	5.19	11.7

Figure 14. Selected runtimes and speedups

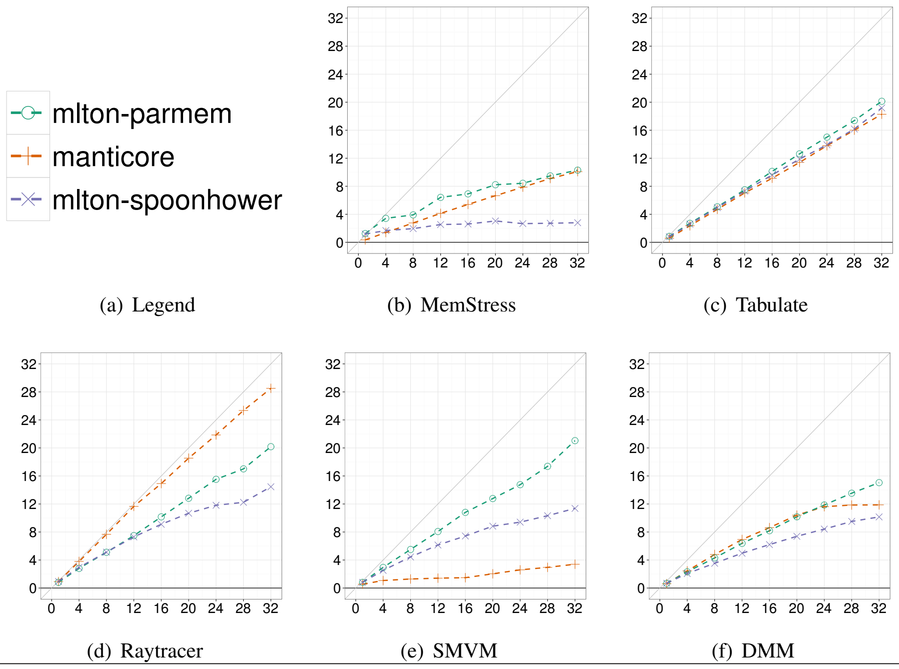

Figure 15. Speedup plots for selected benchmarks. X-Axis is Number of Processors, Y-Axis is Speedup.

benchmarks by replacing the `par` calls by sequential composition. In performance evaluations, it is sometimes preferable to use highly optimized serial code as a baseline. Since, however, we wish to study the performance and the scalability of our memory manager, doing so would obscure our results.

We run each benchmark five times and report the median of the results in the tables and graphs. We measure two quantities across these benchmarks. The first quantity is the single-processor slowdown, or *overhead*, compared to a sequential run. As the name suggests, the overhead indicates the overhead incurred by the parallel runtime and memory manager and helps us establish the practical viability of the compiler for parallel computations against a sequential compiler. The second quantity is the speedup of the `mlton-spoonhower`, `mlton-parmem` and Manticore compilers against the `mlton` baseline.

5.4 Performance and Scalability

Figure 14 shows some of the key measurements across our benchmarks using the following quantities.

- Serial run time, T_s, is the time in seconds for a serial run with `mlton`.
- Uniprocessor run time, T_1, is the uniprocessor run-time of the parallel code reported in seconds;
- The overheard O is the overhead calculated as T_1/T_s.
- 32-processor run time, T_{32}, is the 32-processor run-time of the parallel code reported in seconds;
- The speedup S is the 32-core speedup calculated as T_s/T_{32}.

Across all the benchmarks , we notice that the overhead compared to `mlton` is comparable for both `mlton-spoonhower` and `mlton-parmem` and range between -15% and 51%. These measurements suggest that there is some overhead to parallelism but using a hierarchical memory manager is not significantly more expensive than the naïve parallel memory manager employed in `mlton-spoonhower`. In fact, overheads of `mlton-parmem` are slightly better in 3 benchmarks than those of `mlton-spoonhower`. The `manticore` compiler incurs significantly larger overheads compared to `mlton`. This is consistent with earlier observations with the Manticore compiler [12], which focuses on performant parallel execution but not fast sequential execution. Our observed overhead of between 1.19 and 2.95 is in line with expectations [12].

In terms of scalability, which we can observe by comparing the Speedup "S" columns across compilers, our `mlton-parmem` compiler delivers the best speedups for all benchmarks except Raytracer. Figure 15 shows the speedup curves for individual benchmarks. In all benchmarks, we observe that `mlton-parmem` scales nicely up to 32 processors. While `mlton-spoonhower` also scales reasonably in some benchmarks, it consistently fails to scale as well as `mlton-parmem`. Indeed, in both SMM and DMM, we see the speedup of `mlton-spoonhower` starting to plateau towards 32 processors. In MemStress, `mlton-spoonhower` fails to achieve much more than 3× speedup regardless of the number of processors it utilizes. The `manticore` compiler scales very well, but also trails `mlton-parmem`, except in Raytracer. As programs compiled with `manticore` typically suffer from large overhead over the baseline, `mlton`, they require more processors in order to overcome the overhead. Indeed, `manticore` requires about 3 processors just to match the performance of `mlton` on the MemStress benchmark. In Raytracer, the overhead of `manticore` against `mlton` poses less of a problem and allows `manticore` to exhibit high scalability.

These results suggest that the proposed parallel hierarchical memory manager can be implemented in the context of a high-performance compiler such as MLton and can deliver good performance. The overheads and the speedups that our `mlton-parmem` compiler delivers suggest that the basic idea behind the approach of coupling parallel scheduling and memory management is sound. We note, however, that our comparative study does not carefully account for many important factors that can have significant effects on performance including various internal "hardwired" constants such as block and page sizes and amortization constants that vary between compilers. We leave a more careful implementation and a more detailed experimental study for future work.

6. Related Work

There has been significant work on parallel garbage collection and on scheduling for parallel programs, which we discuss below.

Garbage Collection. The work on parallel garbage collection dates back at least to the multilisp [31] parallel collector. The multilisp collector kept separate heaps for each processor and had no shared heap. A variety of parallel collectors use a shared heap, sometimes with a local block-allocated pool for allocation [6, 9, 21, 26, 32]. This requires significant synchronization among the processors, and does not give the locality advantages of a generational collector.

The Doligez-Leroy-Gonthier (DLG) collector [23, 24] introduced the idea of having local nurseries (heaps) for each processor in addition to a global heap. Memory is allocated in the local heaps and promoted to the global heap on a local collection, or in various other situations, such as when writing into state that is in the global heap. The approach has a similar advantage as a single-level generational sequential collector—by the time a local heap is promoted to the global heap, much of what has been allocated is already garbage. Also if the local nursery fits in cache, its locations can be kept "warm" in cache so that allocation is cheaper. Finally, the local collections can be done independently. Several other parallel collectors have refined the idea of local nurseries and a global heap [5, 8, 25, 37, 51]. These collectors differ in precisely how the two levels are divided and what collection approach is used at each level. For example, the original DLG collector allows the global heap to be collected concurrently using Dijkstra's concurrent mark-sweep algorithm [22]. The ABFR collector [8], on the other hand, uses a form of stop-the-world copying collection on the global heap.

To handle mutable data, these two-level collectors ensure a property similar to disentanglement by requiring that most or all (depending on the strategy used by the individual collector) mutable data be stored in the global heap. Thus, mutation-heavy programs will require a great deal of global collection. In all of these collectors, the decision to collect is based purely on when memory is full, or perhaps when mutations happen across boundaries, but is not related to scheduling decisions or other aspects of the tasks themselves. In fact, they are agnostic to the tasks running on the processors.

Marlow et al. use a generational block-structured parallel collector [38] with support for multiple generations. However, the hierarchy does not form a tree, but rather is a sequence of global heaps of varying size.

Pizlo et al. describe a hierarchical heap structure [43], which has some similarities to ours. It is, however, designed for real-time collectors, and in particular as a replacement for the Real-time Specification for Java (RTSJ). The key difference is that the hierarchical structure is meant to be (mostly) static and user-defined, while ours is meant to be highly dynamic and invisible to the user. The idea in their work is that real-time threads can use heaplets at the leaf, and that each heaplet can support its own style of GC.

Bocchino et al. suggest using regions for parallel memory allocation [16]. Regions can be allocated in a tree hierarchy dynamically based on the structure of the parallelism. Their motivation, however, is very different. In particular they use typing rules on the regions to statically prevent any race conditions. They have no discussion of garbage collection within or across regions.

Morrisett et al. used an operational semantics for abstracting garbage collection in a way similar to our operational semantics [40]. However, it was in a purely sequential context, and is more abstract than ours, e.g. not even capturing the distinction between copying and mark-sweep collection.

The results presented here build on our recent paper [4], which proposed coupling computation and memory management.

Scheduling. Nearly all modern parallel programming languages rely on a run-time scheduler to map tasks to processors. Determining an optimal schedule is actually NP-hard [50], but a classic result by Brent [17] shows that a 2-factor approximation can be computed by using a greedy-scheduling principle. Brent's theorem, however, does not take into account the cost of the scheduling algorithm itself. Over the years, many efficient scheduling algorithms implementing Brent's greedy principle have been developed and bounds have been proven on runtime and space use [3, 15, 29, 30, 41], and locality [1, 14]. These schedulers perform well in practice [3, 29, 48] when the granularity of parallel tasks can be controlled so as to prevent the creation of tiny tasks [2, 12, 49] Much of this work on schedulers, however, does not try to couple the structure of memory with that of the schedule as we do here.

7. Conclusion

This paper presents a memory management technique for parallel functional programs that is based on the general strategy of closely coupling the structure of the memory with that of the computation. As we show, such close coupling enables using the invariants of the semantics of the language, such as the disentanglement property, in managing memory.

There are many interesting directions for future research. Here, we consider fork-join programs, which include a relatively large class of computations. Extending the technique to include other forms of parallelism such as async-finish and futures may be interesting and desirable. Another natural direction for future research would be to extend the techniques to allow some mutable state. We present an approach to realizing the semantics in practice by considering a work-stealing scheduler, by describing the data structures for implementing hierarchical heaps, and by describing an algorithm for performing garbage collection. The initial experimental results for our extension of MLton suggest that the proposed techniques can be implemented efficiently. The implementation, however, is incomplete—it employs local collection only—and is not carefully optimized. A natural next step would be to finish a complete and optimized implementation that can perform both local and nonlocal collections. Finally, it would be interesting to investigate if the techniques described here can be applicable to other parallel languages and systems [19, 28, 29, 33–36, 46].

Acknowledgments

This research is partially supported by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940, by the European Research Council under grant number ERC-2012-StG-308246, and by Microsoft Research. We also thank the reviewers for their insightful comments and in particular the idea of using flattening to prove correctness.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems (TOCS)*, 35(3):321–347, 2002.

[2] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[3] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2013.

[4] U. A. Acar, G. Blelloch, M. Fluet, S. K. Muller, and R. Raghunathan. Coupling memory and computation for locality management. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[5] T. A. Anderson. Optimizations in a private nursery-based garbage collector. In J. Vitek and D. Lea, editors, *9th International Symposium on Memory Management*, pages 21–30, Toronto, Canada, June 2010. ACM Press.

[6] T. A. Anderson, M. O'Neill, and J. Sarracino. Chihuahua: A concurrent, moving, garbage collector using transactional memory. In *TRANSACT*, 2015.

[7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2): 115–144, 2001.

[8] S. Auhagen, L. Bergstrom, M. Fluet, and J. 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.

[9] K. Barabash, O. Ben-Yitzhak, I. Goft, E. K. Kolodner, V. Leikehman, Y. Ossia, A. Owshanko, and E. Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Transactions on Programming Languages and Systems*, 27(6):1097–1146, Nov. 2005.

[10] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM SIGPLAN Notices 35(11), pages 117–128, Cambridge, MA, Nov. 2000. ACM Press.

[11] L. Bergstrom. *Parallel Functional Programming with Mutable State*. PhD thesis, The University of Chicago, June 2013. URL `http://manticore.cs.uchicago.edu/papers/bergstrom-phd.pdf`.

[12] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. *J. Funct. Program.*, 22(4-5):382–438, Aug. 2012. ISSN 0956-7968.

[13] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 81–92, 2013. ISBN 978-1-4503-1922-5.

[14] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, 2011. ISBN 978-1-4503-0743-7.

[15] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, Sept. 1999.

[16] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proc. ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 97–116, 2009.

[17] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[18] A. G. Bromley. Memory fragmentation in buddy methods for dynamic storage allocation. *Acta Informatica*, 14(2):107–117, Aug. 1980.

[19] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005. ISBN 1-59593-031-0.

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

[21] P. Cheng and G. Blelloch. A parallel, real-time garbage collector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 36(5), pages 125–136, Snowbird, UT, June 2001. ACM Press.

[22] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, Nov. 1978.

[23] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *21st Annual ACM Symposium on Principles of Programming Languages*, pages 70–83, Portland, OR, Jan. 1994. ACM Press. .

[24] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *20th Annual ACM Symposium on Principles of Programming Languages*, pages 113–123, Charleston, SC, Jan. 1993. ACM Press.

[25] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for java. In *Proc. International Symposium on Memory Management (ISMM)*, pages 183–194, 2002.

[26] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *ACM/IEEE Conference on Supercomputing*, San Jose, CA, Nov. 1997.

[27] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *1st Java Virtual Machine Research and Technology Symposium*, Monterey, CA, Apr. 2001. USENIX.

[28] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6): 1–40, 2011.

[29] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. PLDI '98, pages 212–223, 1998.

[30] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. Program. Lang. Syst.*, 21(2):240–285, Mar. 1999.

[31] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985. .

[32] M. Herlihy and J. E. B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, May 1992. .

[33] S. M. Imam and V. 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, Cracow, Poland, September 23-26, 2014*, pages 75–86, 2014.

[34] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010. ISBN 978-1-60558-794-3.

[35] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000. ISBN 1-58113-288-3.

[36] S. Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 339–401, 2011.

[37] S. Marlow and S. L. P. Jones. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management (ISMM)*, pages 21–32, 2011.

[38] S. Marlow, T. Harris, R. James, and S. L. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In R. Jones and S. Blackburn, editors, *7th International Symposium on Memory Management*, pages 11–20, Tucson, AZ, June 2008. ACM Press.

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

[40] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 66–77, 1995.

[41] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21, 1999.

[42] R. S. Nikhil. ID language reference manual, 1991.

[43] F. Pizlo, A. L. Hosking, and J. Vitek. Hierarchical real-time garbage collection. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM SIGPLAN Notices 42(7), pages 123–133, San Diego, CA, June 2007. ACM Press.

[44] M. A. Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, The University of Chicago, Aug. 2010. URL `http://manticore.cs.uchicago.edu/papers/rainey-phd.pdf`.

[45] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola. Experimental analysis of space-bounded schedulers. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 30–41, 2014.

[46] K. C. Sivaramakrishnan, L. Ziarek, and S. Jagannathan. Multimlton: A multicore-aware runtime for standard ML. *J. Funct. Program.*, 24(6): 613–674, 2014.

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

[48] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 53–66, 2014.

[49] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Trans. Program. Lang. Syst.*, 36(3):10:1–10:51, Sept. 2014.

[50] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[51] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for java. In *Proc. European Conference on Oriented Programming (ECOOP)*, pages 445–469. Springer, 2009.