MOSTLY-SERIALIZED DATA STRUCTURES FOR PARALLEL AND GENERAL-PURPOSE PROGRAMMING

Chaitanya Sunil Koparkar

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Luddy School of Informatics, Computing and Engineering
Indiana University
June 2023

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Sam Tobin-Hochstadt, Ph.D.

Ryan R. Newton, Ph.D.

Daniel P. Friedman, Ph.D.

Lawrence S. Moss, Ph.D.

Jeremy G. Siek, Ph.D.

Milind V. Kulkarni, Ph.D.

April 26, 2023

Acknowledgements

I want to thank my parents, Sulbha and Sunil, without whose support and encouragement I would never have started graduate school, much less finished. They are the most loving parents one could wish for. This dissertation is dedicated to my father, who is no longer with us. Baba, we miss you.

Jayati, my partner and best friend, is a constant source of love and support. She is the main reason I could keep going these last two years. Without her, I'm not sure I would have survived graduate school. Thank you for everything, Jayati.

My advisor, Ryan Newton, was instrumental in getting me admitted into Indiana University. Not only did he take a chance on me, he taught me how to do research and always went above and beyond to offer help and steadfast support. I will forever be grateful for all he has taught me, technical and otherwise. In 2019, I had to make the difficult decision of not moving to Purdue with him. But he respected this, and we continued our work together. Thank you, Ryan; I am fortunate to have been a part of your lab.

I want to offer special thanks to Sam Tobin-Hochstadt. He agreed to take me on as an advisee and to chair my research committee, which allowed me to stay at IU. His non-stop support made the rest of my journey very pleasant. And his technical insights and helpful feedback have made this dissertation immensely better.

Thanks to my other committee members, Larry Moss, Jeremy Siek, Dan Friedman, and Milind Kulkarni, for all they have done to help me over the years and for their suggestions to improve my research work. Whatever little category theory I know, it's because of Larry; he was kind and patient as I slogged through the text and exercises. Dan spent a tremendous amount of time proofreading this dissertation.

Mike Rainey was always there to listen to my ideas and to help me refine them. I have learned a lot by working with him. Thank you for your mentorship, Mike. I could not have gotten here without my other collaborators on the Gibbon project—thank you Michael Vollmer and Laith Sakka.

Thanks also to all the PL Wonks who made my time as a graduate student infinitely more enjoyable. I would like to specially mention some Wonks regulars—Ryan Scott, Buddhika Chamith, Vikraman Choudhury, Andre Kuhlenschmidt, Sarah Spall, Rajan Walia, Tori Lewis, Paulette Koronkevich, Laurel Carter,

Andrew Kent, Caner Derici, Matthew Heimerdinger, and Fred Fu. Ryan, thank you for all the fun conversations and for being a fantastic friend. Andre, thank you for generously hosting all the board game nights and for all your kindness.

Thanks to my friends, Advait Marathe, Parichit Sharma, Andrew Holland, and Omkar Bhide; Bloomington would not have been such a fun place to stay without you. I especially want to thank Omkar and Spruha, who nursed me for a month when I broke my elbow. I am indebted to both of you. Advait, thank you for being generally awesome.

My research was partly supported by the National Science Foundation, under the award CCF-1725679 and by the Luddy School with the Jose Blakeley Ph.D. Summer Research Award.

For Baba, who missed the end of this journey.

Chaitanya Sunil Koparkar

MOSTLY-SERIALIZED DATA STRUCTURES FOR PARALLEL AND GENERAL-PURPOSE PROGRAMMING

Implementing a functional programming language using pointer-free, serialized representations of algebraic datatypes has proved extremely efficient, especially for programs that operate over bulk tree-like data. But is it only a domain-specific solution? Whether this approach is suitable for a general-purpose language implementation hinges on whether it can be rounded out to also perform reasonably well for programs that allocate data in small pieces and out-of-order. Also, while serialized representations work well for sequential programs, there is an intrinsic tension between density and parallelism. As the name implies, *serialized* data must often be read and written *serially*, due to the lack of pointers. These challenges are the focus of this dissertation. We show that mostly-serialized data structures are a safe, efficient, and practical foundation for a parallel and general-purpose programming language.

To support parallelism, first, enough *indexing* information must be left in the representation so that parallel tasks can "skip ahead" and process multiple subtrees in parallel. Second, the allocation areas must be bifurcated to allow allocation of outputs in parallel. we propose a strategy where form follows function: where data representation retains pointers only insofar as parallelism is needed, and both data representation and control flow "bottom out" to sequential pieces of work that are pointer-free. We present Parallel Gibbon, a compiler that obtains the benefits of dense data formats *and* parallelism. we formalize the semantics of the *parallel location calculus* underpinning this novel implementation strategy, and show that it is type-safe. Parallel Gibbon exceeds the parallel performance of existing, mature compilers for purely functional programs.

Next, we propose a new memory management strategy for a language with mostly-serialized data structures. It uses a hybrid, generational garbage collector, where growable *regions* of memory are the units of allocation. Regions are bump-allocated into a young-generation and objects are bump-allocated within those regions. Minor collections copy data into larger regions in the *old* generation, which uses region-level reference counting with another (backup) tracing collector. Implementing this approach requires overcoming several challenges unique to the mostly-serialized, dense data representation strategy. The resulting system maintains excellent performance for bulk traversal and allocation programs, while greatly improving performance on other kinds of workloads. For small, out-of-order allocations where this approach is weakest, the performance more closely resembles mature compilers that have been heavily optimized for such programs, using traditional memory representations.

Contents

List of Figures

List of Tables

Introduction

Functional languages, such as Haskell and dialects of ML, offer algebraic datatypes, a powerful language based technology that can express many complex data structures and, nevertheless, provide a pleasantly high level of abstraction for application programmers. The high-level of specification of algebraic datatypes leaves a great deal of wiggle room for a range of low level implementation strategies. The way their structures are laid out in memory and how they are accessed by functions can have a major impact on performance.

Most conventional approaches resort to using fully or mostly boxed representations of values, using one heap object per data constructor as shown in Figure 1.1. Consuming such a value requires consuming a full heap object, plus a header word, plus a word-sized pointer. This representation is efficient for random access and shape-changing modifications, but not for workloads that allocate or process large values in bulk. On such workloads, pointer-heavy representations are not favored by current trends in computer architecture. Chasing pointers leads to scattered memory access patterns and poor data locality, which are very inefficient. Programs that traverse and construct trees of algebraic datatypes are widely used across all domains of computer science, ranging from compiler passes, to the web-browser Document Object Model, to particle simulations with space-partitioning trees, and thus optimizing their performance has wide-ranging benefits.

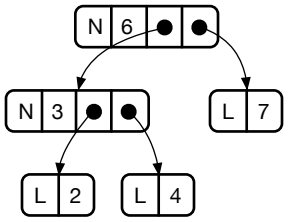


FIGURE 1.1. Standard in-memory representation of a binary search tree—a collection of heap objects linked with pointers. `N` is short for `Node` and `L` is short for `Leaf`.

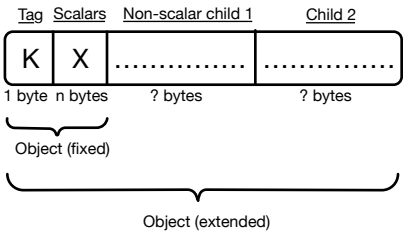


FIGURE 1.2. Anatomy of a Gibbon object. Each object starts with a one-byte tag corresponding to the data constructor within a sum type. The tag is followed by n bytes (known at compile time) of scalar fields such as numbers, booleans etc. The non-scalar, or *packed*, fields come next. Each packed field can contain an unbounded number of bytes which cannot be known at compile time.

Representing values as *pointer-free, serialized byte arrays* can have huge performance advantages [**26, 44, 38**]. Recently, several lines of work have explored computing directly with serialized data representations of varying density. Libraries like Cap'N Proto [**64**], FlatBuffers [**27**], Haskell Compact Normal Forms (CNF) [**68**] etc. allow a complete value to reside in a single contiguous chunk of memory. That is, the start of the value, or, the root of the tree representing the algebraic datatype, *and* all its subtrees can be allocated and stored in a single memory chunk using these libraries. The pointers between the subtrees are still retained so that they can be reached and traversed efficiently, but they are transformed into relative, offset-based pointers. Since it is serialized, this memory chunk can be stored on disk or sent across the network, and crucially, the resident value can be traversed by clients *without a separate deserialization pass*, thereby bypassing a major source of overhead.

Such efficiency is also achieved by Gibbon [**66**], a compiler for a polymorphic, higher-order subset of Haskell[1], which compiles its programs to C code, and employs whole-program compilation and monomorphization. Thanks to using pointer-free, serialized representations as the *native* representation of recursive algebraic datatypes, Gibbon can harness both efficient IO to external devices, skipping traditional (de)serialization like the aforementioned libraries, *and* faster traversals of data in memory, unlike previous systems. Recursive traversals over irregular tree data are over 2× faster in Gibbon than the fastest existing compilers, including `ghc`, `gcc`, `java`, `ocaml`, `mlton`, `rust`, `chez`, etc., and 3× and 10× faster than Compact Normal Forms and Cap'N Proto respectively.

[1] A subset of "Strict Haskell" (`-XStrict`), specifically.

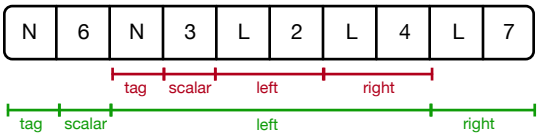


FIGURE 1.3. The tree in Figure 1.1 represented as a *fully serialized* Gibbon value. `N` is short for `Node` and `L` is short for `Leaf`.

Gibbon uses depth-first layouts similar to virtually all (de)serialization frameworks and the aforementioned libraries for operating directly on serialized data. In the context of Gibbon, "serialized" refers not just to storing subtrees consecutively in memory, but also a form of compaction: eliminating pointers between subtrees by *inlining* child structures directly into their parents where possible. Figure 1.2—which is explained in more detail in Section 2.2.2—shows the anatomy of a Gibbon object. Gibbon uses growable *regions* (essentially, memory buffers) as units of allocation. Each region contains a single *value*, which is itself made up of numerous *objects*[2]. Objects living in the same region are packed side-by-side. Pointers are retained, but they are the exception rather than ubiquitious.

A Gibbon data constructor is, by default, only a one-byte *tag*[3], followed immediately by its fields. Wherever possible, this tag occurs inline in a bytestream that hosts multiple objects. As such, values tend to be compact in RAM and be laid out in contiguous blocks, which admit efficient linear traversals. With this (mostly) contiguous layout, a structure can be reliably traversed in a linear fashion, and thereby reap substantial benefits of modern optimizations such as hardware prefetching. For example, a data constructor such as `Node`, in `data STree = Node Int STree STree | Leaf Int | Empty`, treats `Int` as a value type which is inlined immediately after a (one-byte) tag, and the type `STree` is treated just the same! The next two fields of type `STree` are inlined one after the other in the same region, as shown in Figure 1.3.

When pointers are needed for efficient operation of the program, the programmer could include `Ptr T` types explicitly, but it is more common to let the compiler automatically extend the datatype to include a *tagged indirection pointer*, `Ind (Ptr STree)`, as an implicit additional case to the sum type. Using such pointers, objects can point to objects in other regions, which is crucial to support *sharing* common structures.

[2]We use the term *value* to refer to logical values in the source language, and *object* to refer to the allocation resulting from a single data constructor: e.g., a list versus a single cons-cell.

[3]A number assigned to each data constructor of a datatype; it starts at 0 and can go up to 255.

The implicit inclusion of indirections is why Gibbon structures are only *mostly* serialized[4]. We will discuss further details of including pointers in the representation in Section 2.2.3.

There are several advantages to working directly with serialized representations of data structures:

- Data can be read from disk or network without deserialization (e.g. via `mmap`).
- A serialized representation can take many times fewer bytes to represent than a normal pointer-based representation, which improves performance of the memory subsystem.
- Since many objects are allocated in the same region, fewer memory allocations are needed to create serialized structures.
- Data can be traversed significantly faster once in memory due to predictable, linear memory accesses that are favored by optimizations like hardware prefetching.

But, while serialized representations work well for sequential programs, there is an intrinsic tension between density and parallelism. As the name implies, *serialized* data must often be read and written *serially*, due to the lack of pointers. Also, Gibbon's approach to data representation has proved effective for compiling tree-traversals. But is it only a domian-specific solution? Whether this approach is suitable for a general-purpose language implementation hinges on whether it can be rounded out to also perform reasonably well for programs that allocate data in small pieces and out-of-order. These challenges are the focus of this dissertation. We show that mostly-serialized data structures are a safe, efficient, and practical foundation for a parallel and general-purpose programming language.

1.1. Reasoning about performance of programs compiled with Gibbon

While Gibbon can compile and run all sorts of programs, certain programs receive a performance boost while certain others receive a performance penalty. This penalty is because Gibbon and its serialized representations favor a certain allocation pattern, and inefficiencies arise when programs stray from this favored pattern. In this section we provide an intuition on how to estimate the performance of programs compiled with Gibbon. We will return to this topic in Sections 2.2.2 and 2.3.

Gibbon's allocation mechanism and the associated costs are quite different from other languages that use pointer-based data structures. In those languages, each object created on the heap typically triggers an expensive memory allocation (`malloc`)[5]. For example, constructing a linked-list in C will require one

[4]Indirection pointers are sometimes needed for the compiler to preserve the program's asymptotic complexity, e.g., when compiling a program that shares common data between two values, including newer values that reference older ones.

[5]Certain compilers use a bump pointer nursery to allocate objects; this is cheap and fast, but the unit of allocation is still an individual object, and too many objects will eventually fill up the nursery and trigger a collection. On the contrary, objects

memory allocation per cons-cell. This is not true in Gibbon, however, as it uses growable regions; each region contains a single value which is made up of numerous objects packed side-by-side. Writing multiple objects in a region doesn't require any memory allocations and is therefore cheap. As a result, constructing a linked-list in a single region in Gibbon is quite efficient. But, allocating a region itself requires a memory allocation. A region also has some metadata associated with it that must be tracked and kept up-to-date, which adds further overhead. Thus, to reason about a program's performance we must reason about its region allocation behavior. How might we do that?

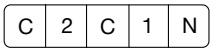
As a rough approximation, simple *naturally recursive* functions that construct values *in order* are very efficient, since they allocate only one region. Functions which construct values *out of order* need to allocate more regions and might not always perform well. Tail recursive functions usually need to allocate one region per recursive call and are therefore slow in Gibbon. (In Chapter 4, we address these performance sore spots by making region allocations efficient using a modified memory management system.) We explain what *in-order construction* means and other details regarding it next.

1.1.1. Region allocation. Values constructed using a data constructor are called *packed* values and these are stored in regions[6]. Gibbon allocates one region to store each packed value. Within a region, all objects must be written in the same order they appear in the region. Intuitively, we can imagine each region having a single write cursor that starts at the beginning and chugs along the region performing writes in a continuous fashion. The write cursor is not allowed to *jump* forward. Objects of recursive types can be arbitrarily large. Thus, we cannot delay writing an object and jump forward to a byte-address after it, because we cannot know how much space will the delayed object require. We learn the size of an object/value only after it is completely written.

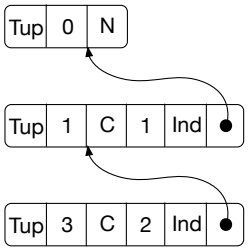
Gibbon uses "location inference" to decide *what* region an object belongs to and also *where* in that region it will be, relative to other objects. Without knowing all the details of the location inference procedure, one simple rule is quite helpful in understanding how a value will map on to a region. When all packed fields of a data constructor are initialized *in order*, the constructor tag and all the fields will be written in a single region. The tag will be written at the beginning, the first field will be written next, and so on. When a packed field of a data constructor is initialized *out of order*, it will be written in its own

created in a region do not fill up the nursery and thus don't trigger a collection. But, allocating too many *regions* will trigger a collection.

[6]Numbers, booleans etc. are called scalar values and these are stored on the stack or in registers.



(A) In-memory representation of a linked-list constructed by `mkList`. `C` is short for `Cons` and `N` is short for `Nil`.



(B) In-memory representation of a structure constructed by `mkListSum`. `Tup` is the tag for the tuple, `C` is short for `Cons`, and `N` is short for `Nil`.

FIGURE 1.4. `mkList` versus `mkListSum`.

separate region. That's because the final byte-address of this field in the original region can only be known after its preceding fields have been written—we cannot know the size of these preceding fields sooner.

For example, given a data constructor `(K x y z)`, if the program binds `x` first, `y` next, and `z` last, this is considered an *in-order construction* and this entire value will be allocated in a single region. But, if the program binds `z` first, `x` next, and `y` last, this value will *not* be allocated in a single region. Here we cannot statically know how much space will `x` and `y` occupy before they're written. Thus, these fields have to be written in the region first, and only then can we know the final byte-address where `z` should be written. But since `z` has to be initialized before `x` and `y`, it has to be allocated in a separate region. Next, an indirection pointing to the separately allocated `z` will be written at the byte-address after `y`, to avoid copying `z`.

If the data dependencies allow it, we can transform the program so that the fields of data constructors are always initialized in order. For example, if `x`, `y` and `z` can be initialized independently of each other, we can modify the program so that it initializes these fields in the desired order. But if `x` or `y` depend on `z` in some way, then their order of initialization cannot be changed.

1.1.2. In-order construction, example. The following Gibbon function constructs a list value in order. Here a single region will be created and the entire list will be written within this region. (The final result is similar to CDR-coding [**15**] techniques developed previously.)

```
mkList :: Int → List
mkList 0 = Nil
mkList n = Cons n (mkList (n-1))
```

Benchmark	Serialized Repr.	Pointer-based Repr.
mkList	0.020	0.043
mkListSum	0.610	0.079

TABLE 1.1. Run time in seconds of `mkList` and `mkListSum` compiled with Gibbon to use serialized and pointer-based representations.

Each function[7] is given its own output region that is used to write the function's output. Let us consider the two cases of `mkList`. The case when the input number is `0` is trivial; the constructor `Nil` has no fields and therefore a tag corresponding to `Nil` can be written in the output region directly. The second case is also straightforward; `Cons` only has one packed field and therefore the tag corresponding to `Cons`, the number `n`, and the recursively constructed list of length `(n-1)` can all be written to the output region directly. Figure 1.4a shows the structure Gibbon constructs.

1.1.3. Out-of-order construction, example. Let us now consider a slight variation of `mkList` that returns a tuple containing a list and the sum of its elements:

```
mkListSum :: Int → (Int, List)
mkListSum 0 = (0, Nil)
mkListSum n = let y = mkListSum (n-1) in
              case y of
                (sum, ls) → (n + sum, Cons n ls)
```

The case when the input number is `0` is trivial like before. But the second case is quite different. Here the final output value of the function *depends* on the value `y` resulting from the recursive call `mkListSum (n-1)`. Thus, `y` has to be initialized first. But `y` cannot be initialized in the function's output region because `y` is not the first packed object in the output and also because the components of `y` appear in a different order in the final output—it has to be initialized in its own separate region. We can now pattern match on `y` to extract its components. Afterwards, the tag corresponding to the tuple (`Tup`), the number `(n+sum)`, the tag corresponding to `Cons`, the number `n`, and an indirection pointing to the list `ls` (contained within `y`) will be written in the output region. Thus, `mkListSum` will allocate a separate region for each recursive call. Figure 1.4b shows the structure Gibbon constructs.

[7]Functions that return a scalar type do not use an output region.

1.1.4. Performance evaluation. Table 1.1 shows the run time performance of these functions compiled with Gibbon to use serialized and pointer-based representations. We use a single-socket Intel E5-2699 18 core machine with 64GB of memory and running Ubuntu 18.04 for this experiment. `mkListSum` performs more work of allocating and pattern matching on tuples compared to `mkList`, but they are algorithmically quite similar to each other. Correspondingly, their run time is in the same ballpark in the pointer-based representation. But this is not true for the serialized representation where `mkListSum` is 30× slower than `mkList`! In fact, `mkListSum` with serialized representation is 8× slower than in pointer-based representation as well. Of course, the excessive region allocations are to blame here. A slight change to the program has degraded its performance significantly. Presently, Gibbon doesn't warn programmers when a function has poor region allocation behavior like `mkListSum`. We plan to add this feature in the future. Other functions such as `foldLeft` versus `foldRight` have similar performance characteristics—`foldRight` is significantly more efficient since it allocates fewer regions.

1.2. Parallelism and Serialization

The lack of pointers makes running parallel computations over serialized representations challenging. For example, to sum all the leaf values of a tree node in parallel, we need to access its left and right subtree simultaneously. In a serialized representation, the only way to reach the right subtree is to walk over the left subtree first to find its end, which essentially sequentializes the whole computation. To change that, first, enough *indexing* information must be left in the representation so parallel tasks can "skip ahead" and process multiple subtrees in parallel. Second, the allocation areas must be bifurcated to allow allocation of outputs in parallel.

In this dissertation we propose a strategy where form follows function: where data representation retains pointers only insofar as parallelism is needed, and both data representation and control flow "bottom out" to sequential pieces of work that are pointer-free. That is, granularity-control in the data mirrors traditional granularity-control in parallel task scheduling. We demonstrate this solution by extending the Gibbon compiler with support for parallel computation, introducing *Parallel Gibbon*. We also extend Gibbon's typed intermediate language, adding parallelism and give an updated formal semantics (Section 3.2).

In addition to tree traversals, we show that Parallel Gibbon can efficiently compile other parallel programs, such as sort and search algorithms (Section 3.4) to match or exceed the performance of the best existing parallel functional compilers. We choose a *functional* focus for three primary reasons:

- Many tree traversals have different input and output types—as in a compiler pass that converts between intermediate languages—which necessitates *out-of-place* traversals even in an imperative language.
- Even pure programs can use mutable data, via linear types. (Gibbon uses these and eschews the *IO monad*.)
- The purely-functional parallel Gibbon programs considered in this work are intrinsically *data-race free*.

The last point is worth emphasizing: every time a language adds both parallel constructs and mutable data, it enables data-races and must define a memory model to give them meaning. In this work, we extend Gibbon with linearly-typed primitives for mutable data[8] (Section 3.3.6), while keeping the language race-free. we claim that linearly-typed mutable data, efficient data representation, and compiler-supported parallelism are a synergistic combination. In Parallel Gibbon programs, as in other purely functional parallel programs, parallelism annotations not only don't introduce races but also *do not affect program semantics*, meaning that these programs are *deterministic* as well as data-race free.

Ultimately, we believe that this work shows one path forward for high-performance, purely-functional programs. Parallelism in functional programming has long been regarded as theoretically promising, but has a spottier track record in practice, due to problems in runtime systems, data representation, and memory management. Parallel Gibbon directly addresses these sore spots, showing how a purely-functional program operating on fine-grained, irregular data can also run fast (sequentially) and parallelize efficiently. This complements more well-trodden areas of compiler research on parallelism, such as dense and sparse collective operations on arrays [**5, 1, 46, 13**]. That is, the approach described in this paper—for general-purpose, recursive functional programs, including tree traversals—could be combined with targeted EDSLs or libraries implementing additional parallel programming idioms, such as Haskell's Accelerate [**16**]. Both determinism and data-race-freedom would be *compositional* within the functional-parallelism setting. Indeed, we have taken the first steps in this direction, adding a small set of parallel array primitives to Gibbon (Section 3.3.6).

1.3. Memory Management

In classic region calculi [**62, 60**], regions can be immediately deallocated at the end of their lexical scope. On the other hand, Gibbon's indirection pointers can cause a region to stay alive beyond its lexical

[8] Leveraging the Linear Haskell [**9**] extensions now available in GHC 9

scope, for example if a pointer to it is captured by another region which is still in scope. Thus, an additional garbage collection scheme is needed to free regions that live beyond their lexical scope.

There have been previous *incomplete* answers to memory management based on regions, which lack sufficiently prompt deallocation. Prior work on Gibbon used a combination of region-based memory management, with region-level reference counting [**34**]. Like the early work on MLkit[**60**] (or more recent work on UrWeb [**19**]), object lifetime depends on the lexically-scoped region it is assigned to. For example, if an object is assigned to the "global region" it may not be freed *until the end of program execution*, even if it became unreachable much earlier than that. In contrast, while tracing collectors are not *garbage free*,they can bound (unreachable) garbage to, e.g., half of heap size using a semi-space collector, but these region-based alternatives cannot.

Consider a program that repeatedly inserts and deletes from a balanced tree (we will later return to this in Section 2.3.3). Each insertion of an element allocates $log(N)$ tree nodes and makes $log(N)$ unreachable, on average. Such a program will leak memory over time with prior region-based solutions (due to delayed collection), but will use bounded space with a traditional garbage collector (GC). No practical, general-purpose automatic memory management system can fail to free memory in this scenario. For that reason, MLKit later added backup garbage-collection within regions [**23**]. UrWeb, on the other hand, fills a specialized role where allocated memory lasts only as long as a web request. But what about Gibbon and its mostly serialized approach to data management for executing Strict Haskell code?

We propose a garbage collector that takes Gibbon from a domain-specific tool for tree traversals to a *general-purpose functional language implementation.* In this new system named *GC-Gibbon*, we adopt the tried-and-true design of the tracing, generational collector, which is used by many state-of-the-art implementations of functional languages. Such a moving collector makes for a good match with Gibbon's mostly serialized heap, where compactness and linear layout can be maintained opportunistically, as the collector copies values during collections, eliminating pointers during copying. The fast bump allocated young-generation also makes region allocations quite efficient. We employ reference counting in the old generation because Gibbon's regions contain many objects on average, especially after collection, so the overhead of (non-deferred) reference counting is amortized at the region level. Along with these advantages, there are new challenges to address, including:

- Gibbon's regions contain dense byte-aligned data and hence have **no header words** or other padding or extra space.

- Indirection pointers result in **pointers into the middle** of packed data structures, at arbitrary byte offsets.
- Regions contain **partially-written values**, for example, a tree node with a `left` field but no `right` field (yet).
- Partially-written values **grow over time**, in addition to allocating new values (that comprise multiple serialized objects).
- Gibbon programs have **new kinds of roots**—they include not just the usual local variables, but implicit *cursor variables* that point at the boundaries of objects in regions.

1.4. Thesis statement and outline of this dissertation

With the above background, I can now state the thesis:

> Mostly-serialized data structures are a safe, efficient, and practical foundation for a parallel and general-purpose programming language.

This dissertation supports this thesis as follows:

- **serialized data structures**: Recent work defined a type-safe language called the **Lo**cation **Cal**culus [**65**] that formalizes the notion of programming with serialized data structures. I give background on LoCal and the Gibbon compiler in Chapter 2.
- **parallelism**: In Chapter 3, I extend LoCal to formalize a *parallel location calculus* that supports task parallelism and show that the resulting language—named LoCal$^{\text{par}}$—is type safe. This extension requires adding back pointers to the representation in a limited way, thus making the representation *mostly-serialized*.
- **general-purpose**: I demonstrate the generality of this approach by showing that it's possible to retain Gibbon's strong performance on tree-traversals where the serialized data approach is most effective *and* also improve Gibbon to achieve reasonable performance on out-of-order small allocations where the approach is weakest, more closely resembling mature compilers and runtime systems that have been heavily optimized for such programs, using traditional memory representations. The key here is a novel memory management strategy that I describe in Chapter 4.
- **efficiency**: In Chapters 3 and 4, I defend the efficiency of the contributions by showing performance results for a variety of benchmarks. To study the garbage collector's performance, I compare it to other systems with highly optimized and mature collectors such as GHC [**39**] and Java [**21**]. To study the performance of LoCal$^{\text{par}}$ I compare it to other languages and systems that

support efficient parallelism for recursive, functional programs such as MaPLe [**67**], Multicore OCaml [**56**] and GHC.

- **practical**: I defend the practicality of this approach by showing how a language like Haskell can be compiled to use serialized data structures. Gibbon is already a whole-program compiler that can compile a subset of Haskell. In addition to this, I show in Chapter 5 how it could be used as an *accelerator* for a portion of a big Haskell application that is compiled and run using GHC.

1.5. Previously published work

This dissertation is based on research done jointly with other collaborators, which appears in the following papers:

- Chaitanya Koparkar, Vidush Singhal, Aditya Gupta, Mike Rainey, Michael Vollmer, Sam Tobin-Hochstadt, Milind Kulkarni, Ryan R. Newton. 2023. Garbage Collection for Mostly-Serialized Heaps. *Submitted to ICFP 2023*.
- Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. 2021. Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations. *In Proceedings of the ACM on Programming Languages, Volume 5, Issue ICFP.* (ICFP 2021)

And it builds on the Gibbon compiler and its LoCal intermediate language which were introduced in the following papers; while I was a collaborator on these, I consider them as prior background work:

- Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A Language for Programs Operating on Serialized Data. *In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2019).
- Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. *European Conference on Object-Oriented Programming, 2017.* (ECOOP 2017).

Background: LoCal and the Gibbon Compiler

In this chapter we give a high-level overview of LoCal and the Gibbon compiler. We will use the program given in Figure 2.1a as an example. Because these techniques for compiling tree-traversals are directly applicable to *compilers themselves*, we use a miniature compiler pass as an example. The example defines a datatype `Exp` which represents the abstract syntax of a language that supports integer arithmetic, and a function `constFold` that implements constant folding for this language. Constant folding is a common compiler optimization in which expressions with constant operands are evaluated at compile time, thus improving the run-time performance. But here we are trying to optimize the performance of *the constant folding pass itself* rather than the performance of the program produced by constant folding. `constFold` walks over the abstract-syntax-tree, and substitutes all expressions of the form `(Plus (Lit i) (Lit j))` with `(Lit (i+j))`. We only show a simplified `constFold` — for example it doesn't recur on the children of `Plus` before checking if they're literals — to keep it simple enough to serve as a running example.

The program in Figure 2.1a is written using the front-end language for Gibbon, a polymorphic, higher-order subset of Haskell, with strict rather than lazy evaluation. The `(||)` operator used on line 14 denotes a parallel tuple — it marks its operands to evaluate in parallel with each other. But with a purely functional source language, it is semantically equivalent to a sequential tuple—i.e. replacing all "||" occurences with "," yields a valid program—and should be treated as one for now. We will return to this topic in Chapter 3.

Gibbon uses LoCal (short for location calculus) as an intermediate representation (IR) with explicit byte-addressed, mostly-serialized data layout. To go from the vanilla Haskell front-end language to LoCal, it performs *location inference*, a variant of region inference [**62, 60**], on the input programs. The LoCal IR code generated by Gibbon for the `constFold` function is shown in Figure 2.1b. In the following, we use it to sketch out how LoCal works.

2.1. A Primer on Location-Calculus

LoCal is a type-safe IR that represents programs operating on densely encoded (serialized) data. In LoCal, all values reside within *regions*. Abstractly, a region is an unbounded storage area for raw data,

```
1  data Exp = Lit Int
2           | Plus Exp Exp
3           | Sub Exp Exp
4           | Let Sym Exp Exp
5           ···
6
7  constFold :: Exp → Exp
8  constFold exp = case exp of
9    Lit i → Lit i
10   Plus e1 e2 →
11     case (e1, e2) of
12       (Lit i,Lit j) → Lit (i+j)
13       _ → let (e3,e4) =
14               ( constFold e1 ‖
15                 constFold e2 )
16             in Plus e3 e4
17   Sub e1 e2 → ···
```

(A) Constant folding written using the front-end language for Gibbon (Haskell).

```
1  constFold:: ∀l1^r1 l2^r2. Exp@l1^r1 → Exp@l2^r2
2  constFold [l1^r1 l2^r2] exp = case exp of
3    Lit (i:Int@li^r1) → (Lit l2^r2 i)
4    Plus (e1: Exp@la^r1) (e2: Exp@lb^r1) →
5      case (e1,e2) of
6        (Lit(i:Int@lc^r1), Lit(j:Int@ld^r1))
7          → (Lit l2^r2 (i+j))
8        _ →
9          letloc l3^r2 = l2^r2 + 1 in
10         let e3 : Exp@l3^r2 =
11           constFold [la^r1 l3^r2] e1 in
12         letloc l4^r2 = after(Exp@l3^r2) in
13         let e4 : Exp@l4^r2 =
14           constFold [lb^r1 l4^r2] e2 in
15         (Plus l2^r2 e3 e4)
16   Sub (e1: Exp@la^r1) (e2: Exp@lb^r1) →
17     ···
```

(B) Figure 2.1a compiled into LoCal IR by Gibbon.

FIGURE 2.1. Constant folding.

a mini-heap unto itself. All programs make explicit not only the region to which a value belongs to, but also a *location* at which that value is written. In LoCal notation, a location l^r resides in region r. Locations are fine-grained indices into a region (memory addresses), but unlike pointers in languages like C, they can be written to just once and arbitrary arithmetic on locations is not allowed. Locations are only introduced relative to other locations. Once allocated at a particular location, a value cannot be *shared* with another location (within the same region or across regions), and it has to be *copied* to allocate it at a different location. In practice, the Gibbon compiler supports sharing using pointers, which we discuss in Section 2.2.3.

A new location is either: at the start of a region, one unit past an existing location[1], or *after* all elements of a value rooted at an existing location. In the program given in Figure 2.1b, the location $l_3{}^{r_2}$ is one past the location $l_2{}^{r_2}$ (line 9) and $l_4{}^{r_2}$ is after every element of the value rooted at location $l_3{}^{r_2}$ (line 12). This serial ordering imposed on locations is what serializes the resulting value in heap memory. Any expression that allocates takes an extra argument: a location-region pair that specifies where the allocation should happen. The types of such expressions are decorated with these location-region pairs. For example, the `(Lit` $l_2{}^{r_2}$ `i)` data constructor (line 3) allocates a tag at location l_2 in region r_2, and has type `(Exp@`$l_2{}^{r_2}$ `)`. Any scalar arguments passed to a data constructor, such as the unpacked integer `i` in this case, are allocated immediately after the tag. Functions may be polymorphic over any of their input or output locations, and these locations are provided at call-sites. In the example, the function `constFold` is polymorphic over an input location $l_1{}^{r_1}$ and an output location $l_2{}^{r_2}$, and values for these are given at all call-sites. In spite of the forall quantifier in its type signature, the input and output regions given at its call-site must be distinct ($r_1 \neq r_2$) to prevent overwrites. This property is checked by LoCal's type-system (described in Section 3.2.3), which makes multiple writes to any location illegal—with the use of a nursery environment— ensuring that function calls like (`constFold` [$l_x{}^{r_x}$ $l_x{}^{r_x}$] `x`) don't type check.

2.1.1. Sequential execution model. LoCal has a dynamic semantics which can run programs sequentially [**65**]. In this model, regions are represented as serialized heaps, where each heap is an array of cells that can store primitive values (data constructor tags, numbers, etc). A write operation, such as the application of a data constructor, allocates to a fresh cell on the heap, and a read operation reads the contents of a cell. Performing multiple reads on a single cell is safe, and the type-system ensures that each cell (location) is written to only once. At run time, locations in the source language translate to heap indices, which are the concrete addresses of the cells where reads/writes happen. Computing addresses of locations which are at the start of a region, or one past another location is straightforward — the addresses get initialized to `0` and `(prev + 1)` respectively. But evaluating an `after` expression, to get an address one past the end of another, variable-sized value, requires more work.

A naive computational interpretation of this `after` is to simply scan over a value to compute its end. In LoCal's formal model, this is referred to as the *end-witness judgment*. Locations computed via `after` are

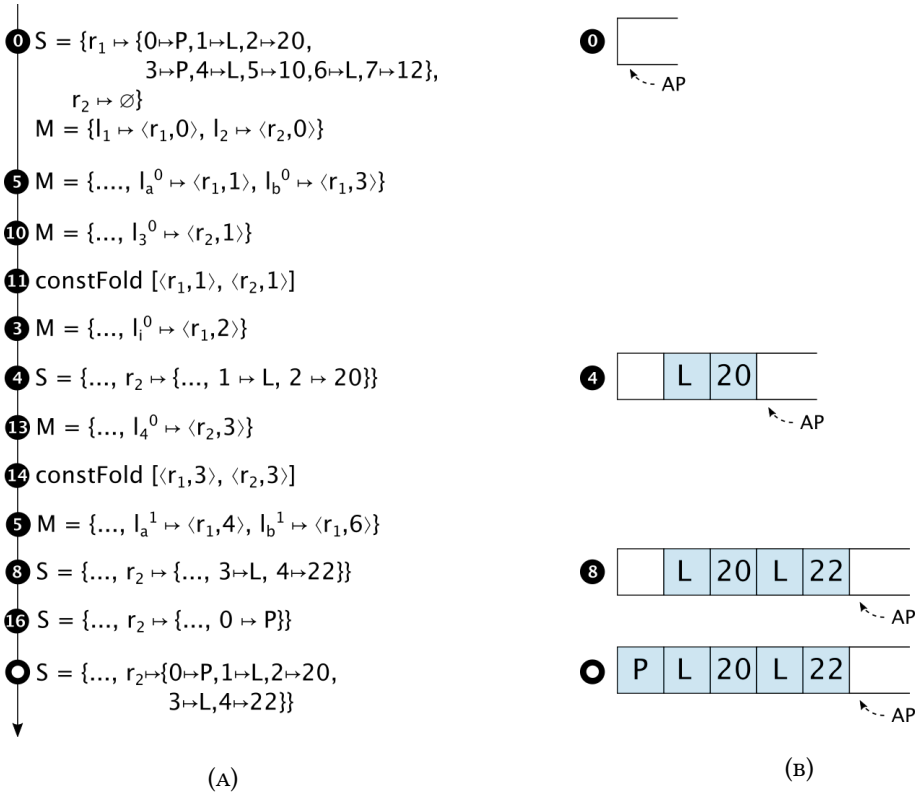
[1] LoCal's formalism uses an abstract unit, namely a heap cell, thus abstracting over computing exact memory addresses until run time. Also, it only permits binding a location that's one cell past an existing location. Practically, one cell is equal to one byte in Gibbon. Also, Gibbon allows a location to directly cross N bytes to efficiently support multi-byte scalar values such as integers or floating point numbers.

used during both read and write operations. For example, when a LoCal `case` expression pattern matches on `(Plus e1 e2)`, it has to scan past `e1` in order to know the starting address of `e2`, which adds $O(n)$ amount of extra work in a *fully* serialized representation! In practice, if values are read in the same order in which they were serialized, a linear scan can be avoided by *tracking end-witnesses* that are naturally computed in the evaluation of the program, for example, by having every read return the address of the cell after it. Intuitively, we can imagine there being a single read pointer that is used to perform all reads in the program. It always points to the next cell to be read on heap, and each read advances it by one. When the program starts executing, the read pointer starts at the beginning of the heap and it chugs along in a continuous fashion. Allocating a serialized value can be thought of in a similar way — that there is a single allocation pointer that starts at the beginning of the heap, and moves along its length performing writes, as illustrated in Figure 2.2b. To avoid changing the asymptotic complexity of programs which read values out-of-order, the Gibbon compiler by default inserts some offset information— such as pointers to some fields of a data constructor— back into the representation, as I discuss in Section 2.2.3. But this doesn't allow *out-of-order allocations*, which will be needed to add parallelism to LoCal (Chapter 3).

2.1.2. Sequential execution model, example. To make this execution model concrete, let us go over a step-by-step trace of the semantics executing `constFold` on `(Plus (Lit 20) (Plus (Lit 10) (Lit 12)))`. The execution trace is given in Figure 2.2. The store S maps regions to their corresponding heaps, and the location map M maps symbolic locations to their corresponding heap indices. The evaluation starts at `(constFold [`$l_1{}^{r_1}$ $l_2{}^{r_2}$`] e)`, and is given a store containing a fully allocated input region r_1 and an empty region r_2 to allocate the output, along with a location map containing the locations $l_1{}^{r_1}$ and $l_2{}^{r_2}$ initialized to the starting addresses of these regions. Since the input region has a `Plus` at the top, execution continues at line 5. The pattern match binds the locations $l_a^{r_1}$ and $l_b^{r_1}$ to the addresses of the sub-expressions `(Lit 20)` and `(Plus (Lit 10) (Lit 12))` respectively[2]. Since both the sub-expressions are not constants, execution continues at line 10. Then, the output location of the first sub-expression, $l_3{}^{r_2}$, is defined to be one past $l_2{}^{r_2}$, and `constFold` is invoked recursively on this sub-expression. Step 4 *copies*[3] the first sub-expression

[2]The execution model binds *unique* locations each time. For example, when execution reaches this pattern match for the first time, the model will use a unique suffix #1 to bind locations $l_a^{r_1}$#1 and $l_b^{r_1}$#1 and update all uses of $l_a^{r_1}$ and $l_b^{r_1}$ to use this fresh version. When execution reaches this pattern match again, perhaps due to a recursive call, it will bind locations $l_a^{r_1}$#2 and $l_b^{r_1}$#2, and so on. We don't show the #*n* suffix here on any location to simplify presentation.

[3]This value is copied because line 3 in Figure 2.1b has a data constructor `(Lit` $l_2{}^{r_2}$ `i)` on the right hand side of the case alternative. If we update the program to return the input expression `exp` directly, Gibbon would allocate a pointer and the value `(Lit 20)` would be *shared* between the input and the output regions. We discuss how sharing works in Section 3.3.3.



(A) (B)

FIGURE 2.2. (a) Sequential, step-by-step execution of the program from Figure 2.1b, and (b) the heap operations corresponding to the output region r_2. Each step is named after its line number in the program and only shows the changes relative to the previous step. *AP* is the allocation pointer. `P` is short for `Plus`, and `L` is short for `Lit`.

by writing a tag L (short for `Lit`), followed by the integer 20 on the heap. Then, the output location of the second sub-expression, ${l_4}^{r_2}$, is defined to be one past every element of the first sub-expression, which occupies two cells after the 0^{th} cell. Thus, ${l_4}^{r_2}$ gets initialized with the address of the 3^{rd} cell. `constFold` is now invoked recursively for the second sub-expression. Following similar steps, the second sub-expression is allocated at ${l_4}^{r_2}$. Since the second sub-expression is a `Plus` with constant operands, it is transformed to `(Lit 22)`. Finally, Step 16 writes the tag P (short for `Plus`) which completes the construction of the full expression, `(Plus (Lit 20) (Lit 22))`.

2.1.3. Another example, constructing a serialized linked-list. Figure 2.3b shows the Haskell source for a simple `mkList` function, which compiles into the intermediate LoCal code in Figure 2.3a. This code operates as follows. The arguments of `mkList` include the number `n` and a symbolic location l^r, where

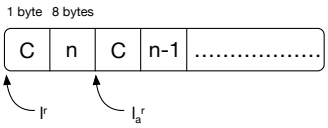
```
data List = Cons Int List | Nil

mkList :: ∀ l^r. Int → List@l^r
mkList [l^r] n =
  if n == 0 then Nil l^r else
    letloc l_1^r = l^r + 9 in
    let tail: List@l_1^r =
      mkList [l_1^r] (n-1) in
    Cons l^r n tail
```

(A) `mkList` compiled into LoCal IR by Gibbon.

```
mkList :: Int → List
mkList 0 = Nil
mkList n = Cons n (mkList (n-1))
```

(B) `mkList`'s starting Haskell source code.



(C) In-memory representation of a linked-list constructed by Figure 2.3a. `C` is short for `Cons`.

FIGURE 2.3. Constructing a *serialized* linked-list.

the resulting list of length `n` will be allocated. This allocation technique is a form of destination-passing style [**53**]. If `n` is zero, `mkList` simply allocates `Nil` at location l^r. Otherwise, it starts constructing a cons cell. First it binds a location $l_1{}^r$ that is 9 bytes past the location l^r (one byte for the `Cons` tag and 8 bytes for the integer). Next, it recursively constructs a list of length `(n-1)` at location $l_1{}^r$. Then it writes the `Cons` tag and the integer `n` starting at location l^r, which completes the construction of this cons cell. Gibbon's location inference algorithm places the cons cell and its tail within the same region because the allocations in `mkList` happen *in order*. The resulting list is shown in Figure 2.3c; it uses a compact structure similar to CDR-coding [**15**]. If the allocations were to happen out-of-order, such as when reversing a linked-list, the resulting list will not be compact. We will return to this topic in Section 2.3.1.

2.2. Compiling LoCal

In the following, we discuss how the Gibbon compiler compiles and runs LoCal programs.

2.2.1. Compiler structure. Gibbon's overall architecture is shown in Figure 2.4. It is implemented as a micropass, whole-program compiler for a polymorphic, higher-order subset of Strict Haskell (option `-XStrict`), which generates C code at the end. It employs monomorphization and specialization to optimize programs at the source language level. Within Gibbon, tuples, and built-in scalar types like Int, Bool etc. are unboxed (never require indirection pointers). To go from the vanilla Haskell front-end language to LoCal, it performs *location inference*, a variant of region inference [**62, 60**], on the input programs. It performs

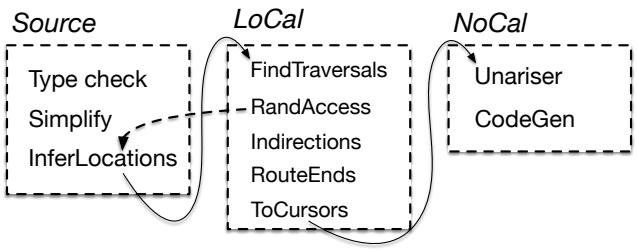


FIGURE 2.4. Gibbon compiler architecture.

full region/location type checking between every pair of passes on the LoCal intermediate representation (IR). After a series of LoCal⇒LoCal passes, the program is lowered to a second IR, NoCal. NoCal is not a calculus at all, but a C-like low-level language where memory operations are made explicit. Code in this form manipulates pointers into regions that are called cursors because of their (largely continuous) motion through regions. NoCal is internally represented as a distinct AST type, with high level (non-cursor) operations excluded. The final backend is completely standard. It eliminates tuples in the unariser, performs simple optimizations, and generates C code. Because of inter-region indirection pointers, a small runtime system is necessary to support the generated code, as we describe next.

2.2.2. Regions, objects and chunks. In this section we describe Gibbon's runtime system that is is responsible for its region-based memory management. In brief, it uses region-level reference counts. Each region is implemented as linked-list of contiguous memory chunks, doubling in size each time. This memory is write-once, and immutability allows it to track reference counts only at the region level. Exiting a `letregion` decrements the region's reference count, and it is freed when no other regions point into it.

Every region itself resembles a heap with a single allocation pointer, which is used to perform all writes in it. The allocation pointer always points to the next available cell on the heap, and new objects are bump allocated. Bump allocation is the mechanism Gibbon programs use to achieve efficient, linear allocation patterns, such as that of `mkList`, where the result value (e.g., list) is laid out in a single region. But if a function's allocation pattern differs, such as when reversing a linked-list or when building a binary tree by allocating its right subtree before the left, the resulting value is placed across multiple regions. Even in such cases where allocations happen out-of-order, Gibbon *could* place the result in a single region by *inlining* all values allocated in intermediate regions. Gibbon does not, because doing so would create extra work for copying data, and often worsen the asymptotic complexity of the input program. As such, Gibbon programs tend to feature mixed allocation patterns, and therefore tend to give rise to complicated memory

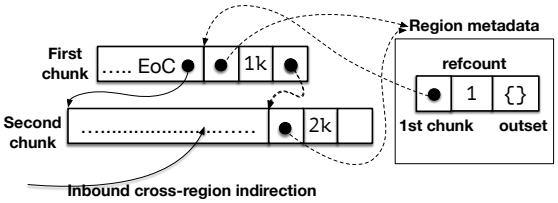


FIGURE 2.5. Run time representation of regions in Gibbon (and GC-Gibbon's old-generation). This particular logical region is made up of two chunks, with an End-of-chunk pointer linking the data. There's an incoming tagged indirection which causes the refcount to be 1. Pointers occurring in the data representation are shown with solid arrows, whereas other implementation related pointers are shown with dashed arrows.

layouts, wherein some intermediate values are allocated in separate regions and are *shared* using pointers (as we explain in Section 2.2.3).

There are two subcomponents to each object: there is the *fixed* portion of an object, consisting of a tag, and any constant-sized fields such as integer scalars. Then there is the *extended* portion of the object which fills a variable number of bytes due to child objects being "inlined" within its representation. Even the extended object may be smaller than the complete logical *value*, which would include all memory reachable by the transitive closure of any indirections.

Gibbon allocates a constant-sized *chunk* of contiguous memory for each fresh region, as Figure 2.5 shows. When this chunk is exhausted, a new one which is double in size is allocated and linked with the previous one using a pointer. This doubling policy is used up to an upper bound, after which constant-sized new chunks are allocated [4].

In order to detect if a chunk is exhausted, every write operation needs to know where the current chunk ends, so that it can perform *bounds checking*. For this reason, every location is dilated to be a pair of `(alloc,end)` pointers at run time. To mark that the serialized stream of data continues in another chunk Gibbon implicitly adds yet another reserved constructor to each datatype—`(EoC Ptr)`—which signals an *end of chunk* and stores a pointer to the head of the next chunk. We refer to these pointers as *chunk redirection pointers*. When a reader hits an `EoC` tag, they must use the stored pointer to resume reading.

2.2.3. Indirection and Shortcut pointers. The initial published version of LoCal's formalism did not model pointers [**65**], but extensions (including LoCal^{par} [**34**] described in this dissertation) do, thereby

[4]Keeping the initial chunk small is optimal in situations where a region contains a small value. But if a region needs to grow to store a large value, the doubling policy would amortize the overall allocation overhead.

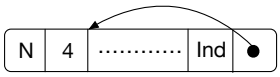
allowing fields to be initialized out of order and point indirectly into other regions. Such extensions are realized by two types of pointers: tagged indirection pointers and untagged shortcut pointers.

Sharing via tagged indirection pointers. Tagged indirection pointers enable a program to share a value among multiple locations. For example, one might write code to construct a binary tree node as `(let x = buildTree n in Node n x x)`, and expect a single, shared value to be allocated for the left and right subtrees. But without a way to store the *addresses* of previously allocated values in the data representation, the right subtree would have to be allocated by *copying* the entire left subtree at the appropriate location. To avoid copying, Gibbon implicitly compiles every datatype `d` to have an additional reserved constructor `(Ind (Ptr d))`, which stores an absolute pointer to a value of type `d`. Given such additional constructors, Gibbon can compile the above code to the following: `(let x = buildTree n in Node n x (Ind (addrOf x)))`. In this version, Gibbon can ensure that the call to `buildTree` allocates directly in place, in the left subtree after the `Node` tag. Thus a pointer is needed only for the right subtree as shown in Figure 2.6a. Similarly, the identity function `(id x = x)` becomes `(id x = Ind (addrOf x))`. Indirection pointers are critical to Gibbon's ability to compile functions without changing their asymptotic complexity. They provide "opt in" pointers, with the default case being dense serialization, and the exceptional case being indirection. The runtime overhead (matching a one byte `Ind` constructor) is placed on the exceptional case.

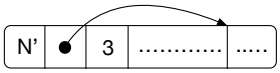
Random-access via shortcut pointers. While indirection pointers enable allocation of values out-of-order, they do not enable *reading* values out-of-order. To this end, Gibbon uses untagged shortcut pointers to enable constant-time random-access to certain fields on a per-data-constructor, per-field basis. The reason shortcut pointers are necessary is that some programs need to "skip over" certain parts of a value to read it out-of-order, and there is no way to accomplish this if the value is fully serialized—the only way to access a particular part in it is to scan past all of the data that has been serialized before it. For example, to compile a program which fetches the rightmost leaf of a binary tree with the correct asymptotic complexiy ($O(\log n)$), Gibbon stores the absolute address of the right subtree in each intermediate node[5] so that it can be accesssed directly *without* traversing the left subtree, which would make this a $O(n)$ operation. Figure 2.6 shows such a node.

In principle, shortcut pointers require less space to store than in a normal pointer-based representation, because no pointer is needed for the leftmost non-scalar field, e.g. one pointer for a binary tree, instead of two. A possible implementation choice would be to store the integer size in bytes of packed fields which

[5]Gibbon constructs such nodes using a different constructor, `(Node' (Ptr Tree) Int Tree Tree)`, to indicate the presence of a shortcut pointer.



(A) A binary tree node that uses a *tagged indirection pointer* as the right subtree to share the object allocated for the left subtree. `N` is short for `Node`, `Ind` is a reserved tag used for indirection pointers.



(B) A binary tree node that stores an *untagged shortcut pointer* to the right subtree so that it can be accessed directly. `N'` is short for `Node'`, a variant of `Node` that contains an additional `(Ptr Tree)` field.

FIGURE 2.6. Data representation with pointers used for sharing (a) and random-access (b).

can be used to *skip over* them. For example, the address of the right subtree can be computed as the address of the left subtree plus size of the left subtree, given that all the fields are serialized side-by-side in a single region. The runtime representation of regions (chunked, growable, as described previously) makes this choice slightly less efficient in practice unfortunately.

2.2.4. Memory management. In classic region calculi [**62, 60**], regions can be immediately deallocated at the end of their lexical scope. But, indirection pointers can cause a region to stay alive beyond its lexical scope, for example, if a pointer to it is captured by another region which is still in scope. Therefore Gibbon needed an additional reference-counting scheme to free regions that live beyond their lexical scope. The way this legacy approach worked is, when a region is initialized—with a `letregion`—its reference count is set to 1, and it is decremented when the region goes out of scope. At this stage, if its reference count hits zero, it is deallocated. Correspondingly, regions also maintain an *outset* that tracks other regions to which this region points. When a region is freed, the reference counts of all pointed-to regions are decremented by iterating this outset. Reference counts are stored in per-region metadata, with per-chunk footers linking to the metadata. Thus the pointer to the chunk footer does double duty for bounds checking and accessing metadata.

2.3. Limitations of Gibbon

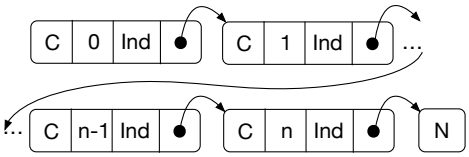
Prior work on Gibbon supports a wide variety of programs in a functionally-correct way, but with significant excessive memory retention in some cases, plus a a major skew in its performance landscape—some programs massively sped up, and others massively slowed down. Specifically, programs which allocate large structures in order, or which map or fold over such structures, get compiled to use a maximally-serialized heap, making them significantly more efficient than their pointer-based counterparts. On the

```
reverse :: List → List → List
reverse Nil acc = acc
reverse (x:xs) acc = reverse xs (x:acc)
```

(A) Starting Haskell source code for accumulator-style list-reverse.

```
reverse :: ∀ l^r m^s n^t. List@l^r → List@m^s → List@n^t
reverse [l^r m^s n^t] xs acc = case xs of
  Nil → Ind n^t (addrOf acc)
  Cons (y:Int@l_y^r) (ys:List@l_ys^r) →
    letregion u in letloc o_1^u =start u in
    letloc o_2^u =o_1^u + 9 in
    let cpy: List@o_2^u = Ind o_2^u (addrOf acc)
    let acc': List@o_1^u = Cons o_1^u y cpy in
    reverse [l_ys^r o_1^u n^t] ys acc'
```

(B) `reverse` compiled into LoCal IR by Gibbon.



(C) In-memory representation of the linked-list in Figure 2.3c reversed using Figure 2.7b. `C` is short for `Cons` and `N` is short for `Nil`.

FIGURE 2.7. Reversing a linked-list.

other hand, programs requiring complicated memory layouts, and small allocations, show various weaknesses of the memory management system.

2.3.1. Allocation overheads. Consider the Haskell code for the standard accumulator-style list-reverse function given in Figure 2.7a. The strict version of this code must build the reversed list from back-to-front. When compiled to LoCal IR, as in Figure 2.7b, we find a `letregion` nested inside the `Cons` case. Essentially, every output cons cell is allocated in a separate region, linked together using indirection pointers (Figure 2.7c)—a traditional linked-list! Programs like `reverse` which allocate small regions at a very high rate show the overheads of region allocation and collection in Gibbon. In fact, the list-reverse

program compiled by Gibbon is 4× slower than its pointer-based version. This isn't surprising because the Gibbon program performs the same number of region-`malloc`'s as the pointer-based version does for objects, but also does additional work to track region metadata information such as the reference counts and outsets. Note that creating excessive number of `Cons` cells isn't the problem here, but rather it's that `reverse` leads to excessive region allocations. That is why `mkList` when compiled with Gibbon is very efficient but `reverse` is not, even though they create the same number of `Cons` cells.

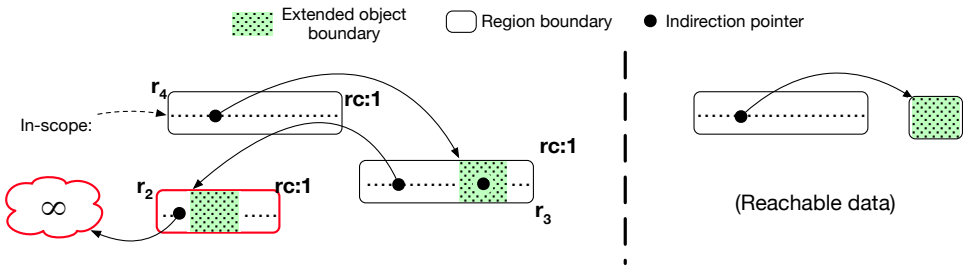
2.3.2. Fragmentation. Another problem that is highlighted in Figure 2.7c is that of fragmentation. First, reverting to a pointer-based representation means all subsequent traversals of this list would be slower because of poor data locality and pointer chasing. A little slower than in a traditional pointer-based heap, because of processing the extra tag-check on each indirection pointer. Second, the space usage is not efficient, with only one `Cons` cell per region, most space is left unoccupied.

2.3.3. Delayed collection. As we alluded earlier, programs in which objects in a region have significantly different lifetimes can be memory-inefficient due to the coarsening of reference counts, and thus can also leak memory: not only leaking the *unused* space (as in the one-Cons-cell-per-region example), but because of space used by formerly-reachable objects sharing the region. Consider a program which repeatedly inserts a random number into a binary search tree (given in Figure 2.8a in the Appendix). Being a *functional* tree-insert, it copies the $log(N)$ nodes along the path it updates. The output tree containing these fresh $log(N)$ nodes must be written to location ${k_1}^s$. This is the *only* output region, as you can see there are no `letregion` expressions in the function body. All the parts of the tree that are *not* recursed into and updated, are linked into the output tree via explicit indirection pointers. The additional `loop` function is included as an example client that repeatedly updates the tree, inserting or deleting random even numbers within the range $[0, 512)$. This artificial client is representative of a common class of programs that update some structure repeatedly, such as a web-server that maintains a data structure modified by successive incoming requests. With Gibbon's previous memory management strategy, such programs would leak memory over time, even though the live data is bounded by a constant in this example. Figure 2.8b demonstrates how old regions and deleted nodes are kept alive. In this heap, region r_4 is still in scope. All other regions have non-zero reference counts only because of indirection pointers. Only a small subset of this heap, however, is truly live. All of region r_4 is live because it is still in scope. Next, region r_4 contains a pointer to an object in region r_3 (highlighted in stippled green), thus keeping it live. Everything else in this heap is unreachable and should be freed! But the coarsened reference counting collector is unable to free

```
loop :: ∀ l^r k^s. RNG → Int → Tree@l^r → Tree@k^s
loop [l^r k^s] rng i tr = if i == 0 then Ind k^s (addrOf tr) else
    let (n,rng') = next rng 0 512 in
    letregion t in letloc j^t = start t in
    let tr':Tree@j^t = if (n%2)==0 then insert [l^r j^t] n tr else delete [l^r j^t] (n-1) tr
    in loop rng' (i-1) tr'

insert :: ∀ l_1^r k_1^s. Int → Tree@l_1^r → Tree@k_1^s
insert [l_1^r k_1^s] n tr = case tr of
  Node (m:Int@l_m^r) (x:Tree@l_x^r) (y:Tree@l_y^r)→
    if m < n then
      letloc k_2^s = k_1^s + 1 in
      let x':Tree@k_2^s = Ind k_2^s (addrOf x) in
      letloc k_3^s = after Tree@k_2^s in
      let y':Tree@k_3^s = insert [l_y^r k_3^s] n y in
      Node k_1^s m x' y'
    else ... -- Insert into the left subtree.
  ··· -- cases for Leaf and Null
```

(A) A LoCal program that repeatedly inserts a random number into a binary search tree.



(B) Heap representation of a long run of program that updates some structure frequently, like the program given in Figure 2.8a. The objects highlighted in red are unreachable but their collection can be arbitrarily delayed.

FIGURE 2.8. A serialized heap with arbitrarily delayed collection in Gibbon.

any dead space because a pointer from a dead part of region r_3 keeps region r_2 alive, thus creating a space leak—stretching the lifetime of objects to depend on a longer lived, even global, region. A supplemental collection is required to properly collect such heaps.

Reconciling Parallelism and Serialization

The lack of pointers makes running parallel computations over serialized representations challenging. In this chapter we introduce the necessary extensions to LoCal and the Gibbon compiler to enable parallelism. We propose a strategy where form follows function: where data representation retains pointers only insofar as parallelism is needed, and both data representation and control flow "bottom out" to sequential pieces of work that are pointer-free. That is, granularity-control in the data mirrors traditional granularity-control in parallel task scheduling. In this chapter:

- We introduce the first compiler that combines parallelism with automatic dense data representations for trees. While dense data [**65**] and efficient parallelism [**67, 48**] have been shown to independently yield large speedups on tree-traversing programs, our system is the first to combine these sources of speedup, yielding the fastest known performance generated by a compiler for this class of programs.
- We formalize the semantics of a *parallel location calculus* (Section 3.2) that underpins the compiler, including a proof of its type-safety (Section 3.2.6). To do so, we extend prior work on location calculi [**65**], which in turn builds on work in region calculi [**62**].
- We evaluate our implementation (Section 3.3) on several benchmarks from the literature (Section 3.4.4). On a single thread, our implementation is 1.93×, 2.53×, and 2.14× faster compared to MaPLe [**67**] (an extension of MLton), OCaml, and GHC, respectively. When utilizing 48 threads, our geomean speedup is 1.92×, 3.73× and 4.01×, meaning that the use of dense representations to improve sequential processing performance *coexists with scalable parallelism.* Most notably, the speedup on a five-pass compiler drawn from a university compiler course was 1.02×, 2.2× and 10.7× over those alternative languages.

3.1. Parallelism in Location-Calculus

In this section, we outline various *latent* opportunities for parallelism that exist in LoCal programs (irrespective of annotation with "‖"). The first kind of parallelism is available when programs access the store in a read-only fashion, such as in an interpreter, for example.

```
interp : ∀ l^r . Exp @ l^r → Int
interp [l^r] t = case t of
  Lit (i : Int @ l_i^r) → i
  Plus (e1 : Exp @ l_a^r) (e2 : Exp @ l_b^r) →
    (interp [l_a^r] a) + (interp [l_b^r] b)
  ...
```

Even though the recursive calls in the `Plus` case can safely evaluate in parallel, there is a subtlety: parallel evaluation is efficient only if the `Plus` constructor stores offset information for its right child node. If it does, then the address of `e2` can be calculated in constant time, thereby allowing the calls to proceed immediately in parallel. If there is no offset information, then the overall tree traversal is necessarily sequential, because the starting address of `e2` can be obtained only after a full traversal of `e1`. As such, there is a tradeoff between space and time, that is, the cost of the space to store the offsets, versus the time of the sequential traversal forced by the absence of offsets.

Programs that write to the store also provide opportunities for parallelism. The most immediate such opportunity exists when the program performs writes that affect different regions. Such writes can happen in parallel because different regions cannot overlap in memory. There is another kind of parallelism that is more challenging to exploit, but is at least as important as the others: intra-region parallelism that can be realized by allowing different fields of the same constructor to be filled in parallel. This is crucial in LoCal programs, where large, serialized data (large trees or DAGs) frequently occupy only a small number of regions, and yet there are opportunities to exploit parallelism in their construction.

Consider the case in Figure 2.1b which recursively calls `constFold` on the sub-expressions of `Plus`. If we want to access the parallelism between the recursive calls, we need to break the data dependency that the right branch has on the left. The starting address of the right branch, namely $l_4^{r_2}$, is assigned to be end witness of the left branch by the `after` expression. But the end witness of the left branch is, in general, known only after the left branch is completely filled, which would effectively sequentialize the computation. One non-starter would be to ask the programmer to specify the size of the left branch up

front, which would make it possible to calculate the starting address of the right branch. Unfortunately, this approach would introduce safety issues, such as incorrect size information, of exactly the kind that LoCal is designed to prevent. Instead, we explore an approach that is safe-by-construction and efficient, as we explain next.

3.2. Region-Parallel LoCal

To address the challenges of parallel evaluation—in concert with dense, mostly-serialized data representations —we start by presenting an execution model, LoCal$^{\text{par}}$, which can utilize *all* potential parallelism in LoCal programs. Parallelism in this formal model is generated *implicitly*, by allowing every let-bound expression to potentially evaluate in parallel with the body. Accordingly, the language omits explicit parallelism "hints" (‖). That is, you'll see in the next sections that implicitly parallel `let` has both a sequential and parallel evaluation rule. By modeling every possible parallelization, the formal model is general — it formalizes all possible valid parallel schedules, and all valid heap layouts. We return to the pragmatic issue of selecting *efficient* parallelizations, i.e. granularity control, in Section 3.3.1.

3.2.1. Region memory and parallel tasks. In the formal model, while parallelism is implicit, there is still a restriction that at most one task allocates in a given region at a time. To realize intra-region parallelism, the model introduces fresh, intermediate regions as needed, that is, when the schedule takes a parallel evaluation step for a given let-bound expression, and the body tries to allocate in the same region. To demonstrate this, let us consider a trace of the region-parallel evaluation of the program from Figure 2.1b, corresponding to the schedule shown in Figure 3.1, where the recursive calls to `constFold` on lines 12 and 14 run in parallel with each other. The parallel fork point for the first recursive call occurs on the 11^{th} step of the trace. At this point, the evaluation of the let-bound expression results in the creation of a new child task, and the continuation of the body of the let expression in the parent task.

Each task has its own private view of memory, which is realized by giving the child and parent task copies of the store S and location map M. These copies differ in one way: each sees a different mapping for the starting location of `e3`, namely ${l_3}^{r_2}$. The child task sees the mapping ${l_3}^{r_2} \mapsto \langle r_2, 1\rangle$, which is the ultimate starting address of `e3` in the heap. The parent task sees a different mapping for ${l_3}^{r_2}$, namely $\langle r_2, \texttt{ivar}\ e3\rangle$. This address is an *ivar*: it behaves exactly like an I-Var [**7**], and, in our example, stands in for the completion of the memory being filled for `e3`, by the child task. Any expression in the body of the let expression that tries to read from this location blocks on the completion of the child task.

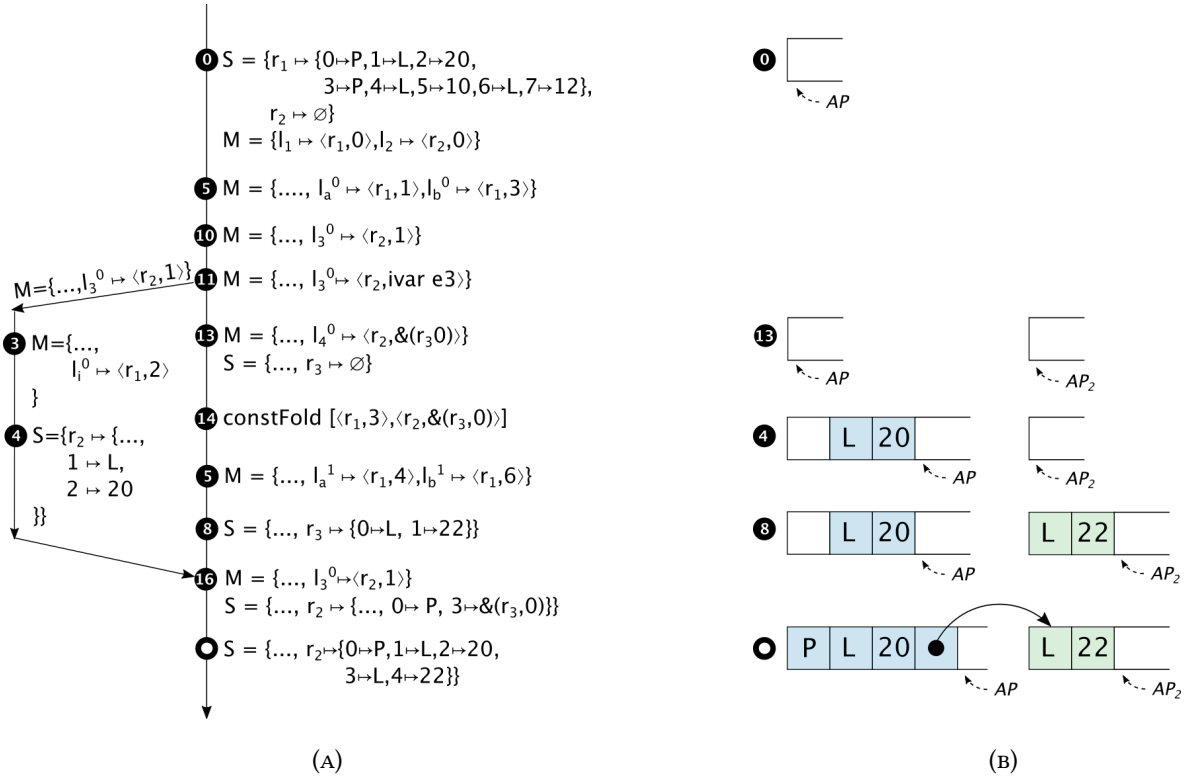


FIGURE 3.1. (a) Parallel, step-by-step execution of the program from Figure 2.1b such that parallel allocations happen only in separate regions, and (b) the heap operations corresponding to the output region r_2. Each step is named after its line number in the program and only shows the changes relative to the previous step. `P` is short for `Plus`, and `L` is short for `Lit`.

The only exception to this blocking-rule is a letloc `after` expression, which is handled differently. Such an expression occurs at line 12, just after the parent continues after the fork point. At this step, the parent task uses an `after` expression to assign an appropriate location for the starting address of `e4`, one past every byte occupied by `e3`. If we synchronize with the child task here, the computation will effectively be sequential. In order to avoid that, the starting address of `e4` is assigned to be $l_4{}^{r_2} \mapsto \langle r_2, \&(r_3, 0)\rangle$. This address is an *indirection* pointing to the start of fresh region r_3, and causes the parent task to allocate `e4` in the region r_3 instead of r_2, which is being allocated to by the child task, thus maintaining the single-threaded-per-region allocation invariant. The parent and child tasks have, in effect, two different allocation pointers for what will functionally be the same region (after joining). The use of `e3` on line 15 forces the parent task to join with its child task. In particular, $\langle r_2, \texttt{ivar}\ e3\rangle$ is substituted by $\langle r_2, 1\rangle$, the starting

address of `e3`, in the expression and the location map M. Also, all the new entries in the location map M and store S of the child are merged into the corresponding environments in the parent. Finally, the regions r_2 and r_3 are linked with a pointer, corresponding to the indirection pointer that was added for the starting address of `e4`.

3.2.2. Syntax and operational semantics. In this section, we present the formal semantics of our parallel location calculus, LoCal$^{\text{par}}$. This semantics has also been mechanically tested in PLT Redex [**24**]. The grammar for the language is given in Figure 3.2. Again, all parallelism in this model language is introduced implicitly, by evaluating `let` expressions. There is no explicit syntax for introducing parallelism in our language, and consequently the language is, from the perspective of a client, exactly the same as the sequential language [**65**].

The parallel operational semantics does, however, differ from the sequential semantics, most notably from the introduction of a richer form of indexing in regions. Whereas in sequential LoCal a region index consists simply of a non-negative integer, it is enriched to an extended region index $i\diamond$ in LoCal$^{\text{par}}$. It consists of either a concrete index i, an `ivar` x, or an indirection pointer $\&(r, i)$. A concrete index is a non-negative integer that specifies the final position of a value in a region. An ivar is a synchronization variable that is used to coordinate between parallel tasks. For example, the `ivar` $e3$ in the sample trace in Figure 3.1, is used to synchronize with the child task that is allocating `e3`. An indirection $\&(r, i)$ points to the address i in the region r, and is used to link together different *chunks* of the same logical region, which may have been introduced to enable intra-region parallel allocation. For example, in the sample trace in Figure 3.1, a pointer $\&(r_3, 0)$ written at the end of the value `e3` links it with the value `e4`, which is allocated to a separate region r_3. And a concrete location cl is enriched to a pair $\langle r, i\diamond\rangle$, of a region r, and an extended region index $i\diamond$. The state configurations of LoCal$^{\text{par}}$ appear in Figure 3.3. Just like in sequential LoCal, a sequential state of LoCal$^{\text{par}}$, t, contains a store S, location map M, and an expression e. But by using enriched concrete locations, the location map also has the ability to contain indirection pointers. A value that can be written to a heap, hv, is similarly enriched to allow indirection pointers.

Figures 3.5 and 3.6 show the complete dynamic semantics of LoCal$^{\text{par}}$. The driver which runs a LoCal$^{\text{par}}$ program initially loads all data types, functions, type checks them, and if successful, then seeds the *Func – tion*, *TypeOfCon*, and *TypeOfField* environments. Let e_0 be the main expression. If e_0 type checks with respect to the T-Program rule (given in Figure B.3), then the main program is safe to run. The initial configuration for the machine is a single task, $(\hat{\tau}, \langle r, 0\rangle, \emptyset; \{\, l \mapsto \langle r, 0\rangle \,\}; e_0)$, and the program can start

$K \in$ Data Constructors, $\tau_c \in$ Type Constructors,

$x, y, f \in$ Variables, $l, l^r \in$ Symbolic Locations,

$r \in$ Regions, $i, j \in$ Concrete Region Indices,

Top-Level Programs	top	$::= \overrightarrow{dd}\ ;\overrightarrow{fd}\ ; e$
Datatype Declarations	dd	$::= \texttt{data}\ \tau_c = \overrightarrow{K\ \overrightarrow{\tau}}$
Function Declarations	fd	$::= f : ts; f\ \overrightarrow{x} = e$
Located Types	$\hat{\tau}$	$::= \tau@l^r$
Types	τ	$::= \tau_c$
Type Scheme	ts	$::= \forall_{\overrightarrow{l^r}}.\ \overrightarrow{\hat{\tau}} \rightarrow \hat{\tau}$
Extended Region Indices	$i\diamond, j\diamond$	$::= i \mid \texttt{ivar}\ x \mid \&(r, i)$
Concrete Locations	cl	$::= \langle r, i\diamond \rangle^l$
Values	v	$::= x \mid cl$
Expressions	e	$::= v$
		$\mid f\ [\overrightarrow{l^r}]\ \overrightarrow{v}$
		$\mid K\ l^r\ \overrightarrow{v}$
		$\mid \texttt{let}\ x : \hat{\tau} = e\ \texttt{in}\ e$
		$\mid \texttt{letloc}\ l^r = le\ \texttt{in}\ e$
		$\mid \texttt{letregion}\ r\ \texttt{in}\ e$
		$\mid \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat}$
Pattern	pat	$::= K\ \ \overrightarrow{(x : \hat{\tau})}\ \ \rightarrow\ \ e$
Location Expressions	le	$::= \texttt{start}\ r$
		$\mid l^r + 1$
		$\mid \texttt{after}\ \hat{\tau}$

FIGURE 3.2. Grammar of LoCal$^{\text{par}}$.

$$
\begin{array}{rrl}
\text{Store} & S & ::= \{\, r_1 \mapsto h_1,\ \ldots\ , r_n \mapsto h_n \,\} \\
\text{Heap Values} & hv & ::= K \mid \&(r, i) \\
\text{Heap} & h & ::= \{\, i_1 \mapsto hv_1,\ \ldots\ , i_n \mapsto hv_n \,\} \\
\text{Location Map} & M & ::= \{\, l_1^{r_1} \mapsto cl_1,\ \ldots\ , l_n^{r_n} \mapsto cl_n \,\} \\
\text{Sequential States} & t & ::= S; M; e \\
\text{Parallel Tasks} & T & ::= (\hat{\tau}, cl, t) \\
\text{Task Set} & \mathbb{T} & ::= \{\, T_1, \ldots, T_n \,\}
\end{array}
$$

FIGURE 3.3. Extended grammar of LoCal$^{\text{par}}$ for dynamic semantics.

$$
\begin{array}{rrl}
\text{Typing Env.} & \Gamma & ::= \{\, x_1 \mapsto \hat{\tau_1},\ \ldots\ , x_n \mapsto \hat{\tau_n} \,\} \\
\text{Store Typing} & \Sigma & ::= \{\, l_1^{r_1} \mapsto \tau_1,\ \ldots\ , l_n^{r_n} \mapsto \tau_n \,\} \\
\text{Constraint Env.} & C & ::= \{\, l_1^{r_1} \mapsto le_1,\ \ldots\ , l_n^{r_n} \mapsto le_n \,\} \\
\text{Allocation Pointers} & A & ::= \{\, r_1 \mapsto ap_1,\ \ldots\ , r_n \mapsto ap_n \,\} \\
 & & \quad \text{where } ap = l^r \mid \emptyset \\
\text{Nursery} & N & ::= \{\, l_1^{r_1},\ \ldots\ , l_n^{r_n} \,\} \\
\text{Typing Env. Map} & \mathbb{\Gamma} & ::= \{\, cl_1 \mapsto \Gamma_1,\ \ldots\ , cl_n \mapsto \Gamma_n \,\} \\
\text{Store Typing Map} & \mathbb{\Sigma} & ::= \{\, cl_1 \mapsto \Sigma_1,\ \ldots\ , cl_n \mapsto \Sigma_n \,\} \\
\text{Constraint Env. Map} & \mathbb{C} & ::= \{\, cl_1 \mapsto C_1,\ \ldots\ , cl_n \mapsto C_n \,\} \\
\text{Allocation Pointers Map} & \mathbb{A} & ::= \{\, cl_1 \mapsto A_1,\ \ldots\ , cl_n \mapsto A_n \,\} \\
\text{Nursery Map} & \mathbb{N} & ::= \{\, cl_1 \mapsto N_1,\ \ldots\ , cl_n \mapsto N_n \,\}
\end{array}
$$

FIGURE 3.4. Extended grammar of LoCal$^{\text{par}}$ for static semantics.

taking evaluation steps from this initial configuration. This configuration can be constructed automatically in a straightforward way.

3.2.2.1. Sequential transitions. A subset of the sequential transition rules are given in Figure 3.5. The rules are close to the original sequential rules, except for some minor differences. For the rule D-DataConstructor, we need to handle the case where an indirection is assigned to the source symbolic

[D-DataConstructor]

$$S; M; K\ l^r\ \overrightarrow{v} \Rightarrow S'; M; \langle r', i' \rangle \quad \text{where}\ \ S' = S \cup \{\, r' \mapsto (i' \mapsto K) \,\};\ \langle r', i' \rangle = \hat{M}(l^r)$$

[D-LetLoc-Tag]

$$S; M; \texttt{letloc}\ l^r = {l'}^r + 1\ \texttt{in}\ e \Rightarrow S; M'; e \quad \text{where}\ \ M' = M \cup \{\, l^r \mapsto \langle r, i+1 \rangle \,\};\ ({l'}^r \mapsto \langle r, i \rangle) \in M$$

[D-LetLoc-After]

$$S; M; \texttt{letloc}\ l^r = \texttt{after}\ \tau@{l_0}^r\ \texttt{in}\ e \Rightarrow S; M'; e \quad \text{where}\ \ \langle r, i \rangle = \hat{M}({l_0}^r);\ \tau; \langle r, i \rangle; S \vdash_{ew} \langle r, j \rangle,\ M' = M \cup \{\, l^r \mapsto \langle r, j \rangle \,\}$$

[D-LetLoc-After-NewReg]

$$S; M; \texttt{letloc}\ l^r = \texttt{after}\ \tau@{l_0}^r\ \texttt{in}\ e \Rightarrow S'; M'; e \quad \text{where}\ \ \langle r, \texttt{ivar}\ x \rangle^{l_0} = \hat{M}({l_0}^r);\ r' \text{ fresh},\ S' = S \cup \{\, r' \mapsto \emptyset \,\};\ M' = M \cup \{\, l^r \mapsto \langle r, \&(r', 0) \rangle \,\}$$

[D-Case]

$$S; M; \texttt{case}\ \langle r, i \rangle^{l^r}\ \texttt{of}\ [\ldots, K\ \ \overrightarrow{(x : \tau@l^r)}\ \ \rightarrow\ \ e, \ldots] \Rightarrow S; M'; e' \quad \text{where}\ \ e' = e[\langle r, \overrightarrow{w} \rangle^{\overrightarrow{l^r}} / \overrightarrow{x}]$$

$$M' = M \cup \{\, \overrightarrow{l_1^r} \mapsto \langle r, i+1 \rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}} \rangle \,\}$$

$$\overrightarrow{\tau_1}; \langle r, i+1 \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle$$

$$\overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle$$

$$K = S(r)(i);\ j \in \{\, 1, \ldots, n-1 \,\};\ n = |\overrightarrow{x : \hat{\tau}}|$$

[D-Let-Expr]

$$\frac{S; M; e_1 \Rightarrow S'; M'; e_1' \qquad e_1' \neq v}{S; M; \texttt{let}\ x : \hat{\tau} = e_1\ \texttt{in}\ e_2 \Rightarrow S'; M'; \texttt{let}\ x : \hat{\tau} = e_1'\ \texttt{in}\ e_2}$$

[D-Let-Val]

$$S; M; \texttt{let}\ x : \hat{\tau} = v_1\ \texttt{in}\ e_2 \Rightarrow S; M; e_2[v_1/x]$$

Figure 3.5. Dynamic semantics rules (sequential transitions).

location l^r. For this purpose, we use a metafunction $\hat{M}$ (formally defined in Section 3.2.4) that can dereference indirection pointers when looking up its address in the location map M. With respect to the rule D-LetLoc-After-NewReg, we now allow the concrete location assigned to the source location ${l_0}^r$ to hold an ivar. The purpose of this relaxation is to allow an expression downstream from a parallelized `let` binding to continue evaluating in parallel with the task that is evaluating the let-bound expression. The task evaluating the `after` expression continues by using an indirection pointing to the start of a fresh region r'. The effect is to make $\langle r', 0\rangle$ the setting for the allocation pointer for the task. If the source location ${l_0}^r$ is assigned to hold a concrete index i, the rule D-LetLoc-After yields an address by using the the end-witness judgment.

3.2.2.2. Parallel transitions. We generalize a sequential state to a parallel task T by adding two more fields: a located type and a concrete location, which together describe the type and location of the final result allocated by the task. A parallel transition in LoCal$^{\text{par}}$ takes the form of the following rule, where any number of tasks in a task set $\mathbb{T}$ may step together.

$$\mathbb{T} \Longrightarrow_{rp} \mathbb{T}'$$

In each step, a given task may make a sequential transition, it may fork a new parallel task, it may join with another parallel task, or it may remain unchanged.

The parallel transition rules are given in Figure 3.6. In these rules, we model parallelism by an interleaving semantics. Any of the tasks that are ready to take a sequential step may make a transition in rule D-Par-Step. A parallel task can be spawned by the D-Par-Let rule, from which an in-flight `let` expression breaks into two tasks. The child task handles the evaluation of the `let`-bound expression e_1, and the parent task handles the body e_2. To represent the future location of the let-bound expression, and to create a data dependency on it, the rule creates a fresh ivar, which is passed to the body of the `let` expression. This same ivar is also the target concrete location of the child task, thereby indicating that it produces this value.

A task can satisfy a data dependency in a rule such as D-Par-Case-Join, where a `case` expression is blocked on the value located at `ivar` x_c, by joining with the task producing the value. Because each task has a private copy of the store and location map, the process of joining two tasks involves merging environments. The merging of the task memories is performed by the metafunctions *MergeS* and *MergeM*, defined formally in Section 3.2.4. We merge two stores by merging the heaps of all the regions that are shared in common by the two stores, and then by combining with all regions that are not shared. We merge two heaps by taking the set of all the heap values at indices that are equal, and all the heap values at indices

[D-Par-Step]

$$\frac{S; M; e \Rightarrow S'; M'; e'}{T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow_{rp} T_1, \ldots, (\hat{\tau}, cl, S'; M'; e'), \ldots T_n}$$

[D-Par-Let-Fork]

$T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow_{rp} T_1, \ldots, (\hat{\tau}_1, cl'_1, S; M; e_1), \ldots T_n, (\hat{\tau}, cl, S; M_2; e'_2)$

where $e = (\texttt{let}\ x : \hat{\tau}_1 = e_1\ \texttt{in}\ e_2);\ \hat{\tau}_1 = \tau_1 @ {l_1}^{r_1}$

x_1 fresh; $cl'_1 = \langle r_1, \texttt{ivar}\ x_1 \rangle;\ e'_2 = e_2[cl'_1/x]$

$M = \{\ {l_1}^{r_1} \mapsto cl_1\ \} \cup M';\ M_2 = \{\ {l_1}^{r_1} \mapsto cl'_1\ \} \cup M'$

[D-Par-Case-Join]

$T_1, \ldots, T_c, \ldots, T_n \Longrightarrow_{rp} T_1, \ldots, T'_c, \ldots T_n,$

where

$T_c = (\hat{\tau}_c, cl_c, S_c; M_c; e_c);\ e_c = \texttt{case}\ \langle r, \texttt{ivar}\ x_c \rangle^{l_c}\ \texttt{of}\ \overrightarrow{pat}$

$T_p \in \{\ T_1, \ldots, T_n\ \} = (\tau_p @ {l_p}^r, \langle r, \texttt{ivar}\ x_c \rangle, S_p; M_p; \langle r, i_p \rangle)$

$M_3 = MergeM(M_p, M_c);\ S_3 = MergeS(S_p, S_c)$

$e'_c = \texttt{case}\ \langle r, i_p \rangle^{l_p}\ \texttt{of}\ \overrightarrow{pat}[i_p/\texttt{ivar}\ x_c];\ T'_c = (\hat{\tau}_c, cl_c, S_3; M_3; e'_c)$

[D-Par-DataConstructor-Join]

$T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots, T_n \Longrightarrow_{rp} T_1, \ldots, T', \ldots, T_n$

where $e = K\ {l'}^r\ \overrightarrow{v};\ \langle r, \texttt{ivar}\ x_j \rangle = \overrightarrow{v_j};\ T_c \in \{\ T_1, \ldots, T_n\ \}$

$T_c = (\tau_c @ {l_c}^r, \langle r, \texttt{ivar}\ x_j \rangle, S_c; M_c; \langle r, i_c \rangle^{l_c})$

$M' = MergeM(M_c, M);\ S' = MergeS(S_c, S)$

$n = |\overrightarrow{v}|;\ \overrightarrow{v'} = [\overrightarrow{v_1}, \ldots, \overrightarrow{v_{j-1}}, \langle r, i_c \rangle^{l_c}, \overrightarrow{v_{j+1}}, \ldots, \overrightarrow{v_n}]$

$\tau_j = TypeOfField(K, j);$

$S'' = LinkFields(S', M, \tau_j, \langle r, i_c \rangle^{l_c})$ if $j \neq n$ else S'

$e' = K\ {l'}^r\ \overrightarrow{v'};\ T' = (\hat{\tau}, cl, S''; M'; e')$

Figure 3.6. Dynamic semantics rules (parallel transitions).

in only the first and only the second heap. The merging of location maps follows a similar pattern, but is slightly complicated by its handling of locations that map to ivars. In particular, for any location where one of the two location maps holds an ivar and the other one holds a concrete index, we assign to the resulting

location map the concrete index, because the concrete index contains the more recent information. After merging the environments, all occurrences of `ivar` x_c are eliminated in the continuation, and are replaced by the index i_p, that represents the starting index of the value produced by the task T_p. Join points in LoCal$^{\text{par}}$ are, in general, deterministic, because they only *increase* the information held by the parent task.

The rule D-Par-DataConstructor-Join handles the case where a data constructor is blocked on the value of its j^{th} field, and it joins with the task producing that value. It is similar to D-Par-Case-Join, and also requires merging environments. But depending on the schedule of execution, if the $(j+1)^{th}$ field of this constructor was computed in parallel with the j^{th} field, they will both be allocated to separate regions, due to the way the rule D-LetLoc-After-NewReg works. These fields have to be reconciled to simulate a single region. For this purpose, we use a metafunction *LinkFields* —defined formally in Section 3.2.4— which stitches together these fields by writing an indirection pointing to the start of the region containing the $(j+1)^{th}$ field at an address one past the end of the j^{th} field. Thus, when all fields of a data constructor are synchronized with, all fields allocated to different regions are linked together by indirection pointers, forming a linked-list.

3.2.3. Type system. Our type system for LoCal$^{\text{par}}$ requires some substantial extensions to the original type system given by [**65**]. These extensions address the need to handle multi-task configurations, which require a number of new typing environments and rules. Before we present these extensions, we recall the typing rule for the configuration of a single task, which is mostly unchanged from the original.

$$\Gamma; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}$$

The context for this judgement includes five different environments. First, Γ is a standard typing environment. Second, Σ is a store-typing environment, mapping *materialized* symbolic locations to their types. That is, every location in Σ *has been written* and contains a value of type $\Sigma(l^r)$. Third, C is a constraint environment, keeping track of how symbolic locations relate to each other. Fourth, A maps each region in scope to a location, and is used to symbolically track the allocation and incremental construction of data structures; Finally, N is a nursery of all symbolic locations that have been allocated, but not yet written to. Both A and N are threaded through the typing rules, also occurring in the output of the judgement, to the right of the turnstile.

Let us first consider the rule T-DataConstructor given in Figure 3.7. It starts by ensuring that the tag being written, and all the fields have the correct type. Along with that, the locations of all the fields of the constructor must also match the expected constraints. That is, the location of the first field should be

[T-DataConstructor]

$$\frac{\begin{array}{l} \mathit{TypeOfCon}(K) = \tau \qquad \mathit{TypeOfField}(K, i) = \overrightarrow{\tau'_i} \\ l^r \in N \qquad A(r) = \overrightarrow{l_n}^r \quad \text{if } n \neq 0 \quad \text{else } l^r \\ C(\overrightarrow{l_1}^r) = l^r + 1 \qquad C(\overrightarrow{l_{j+1}}^r) = \texttt{after } (\overrightarrow{\tau'_j}@\overrightarrow{l'_j}^r) \\ \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau'_i@l_i}^r \end{array}}{\begin{array}{l} \Gamma; \Sigma; C; A; N \vdash A'; N'; K\ l^r\ \overrightarrow{v} : \tau@l^r \\ \text{where } \ A' = A \cup \{\, r \mapsto l^r \,\};\ N' = N - \{\, l^r \,\} \\ \qquad\qquad n = |\overrightarrow{v}|;\ i \in I = \{\, 1, \ \ldots\ , n \,\};\ j \in I - \{\, n \,\} \end{array}}$$

Figure 3.7. A copy of the typing rule for LoCal data constructor given in [**65**].

[T-DataConstructor-Ivars]

$$\frac{\begin{array}{c} \mathit{TypeOfCon}(K) = \tau \qquad \mathit{TypeOfField}(K, i) = \overrightarrow{\tau'_i} \\ l^r \in N \qquad A(r) = \overrightarrow{l_n}^r \quad \text{if } n \neq 0 \quad \text{else } l^r \\ C(\overrightarrow{l_1}^r) = l^r + 1 \qquad C(\overrightarrow{l_{j+1}}^r) = \texttt{after } (\overrightarrow{\tau'_j}@\overrightarrow{l'_j}^r) \\ \exists_{k \in I}.\langle r, \texttt{ivar } x_k \rangle = \overrightarrow{v_k} \qquad \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau'_i@l_i}^r \end{array}}{\begin{array}{l} \Gamma; \Sigma; C; A; N \vdash A'; N'; K\ l^r\ \overrightarrow{v} : \tau@l^r \\ \text{where } \ n = |\overrightarrow{v}|;\ i \in I = \{\, 1, \ \ldots\ , n \,\};\ j \in I - \{\, n \,\} \end{array}}$$

Figure 3.8. Additional typing rule for type checking an in-flight data constructor.

[T-Task]

$$\frac{\Gamma; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}}{\Gamma; \Sigma; C; A; N \vdash_{task} A'; N'; (\hat{\tau}, cl, S; M; e)}$$

[T-Taskset-Empty]

$$\mathbb{\Gamma}; \mathbb{\Sigma}; \mathbb{C}; \mathbb{A}; \mathbb{N} \vdash_{taskset} \mathbb{A}; \mathbb{N}; \emptyset$$

[T-Taskset]

$$\frac{\begin{array}{c} (\hat{\tau}, cl, S; M; e) = T_i \quad \Gamma = \mathbb{\Gamma}(cl) \quad \Sigma = \mathbb{\Sigma}(cl) \quad C = \mathbb{C}(cl) \quad A = \mathbb{A}(cl) \quad N = \mathbb{N}(cl) \\ \Gamma; \Sigma; C; A; N \vdash_{task} A'; N'; T_i : \hat{\tau} \quad \mathbb{A}' = \mathbb{A} \cup \{\, cl \mapsto A' \,\} \quad \mathbb{N}' = \mathbb{N} \cup \{\, cl \mapsto N' \,\} \\ \mathbb{\Gamma}; \mathbb{\Sigma}; \mathbb{C}; \mathbb{A}'; \mathbb{N}' \vdash_{taskset} \mathbb{A}''; \mathbb{N}''; \{\, T_1, \ldots, T_n \,\} \end{array}}{\mathbb{\Gamma}; \mathbb{\Sigma}; \mathbb{C}; \mathbb{A}; \mathbb{N} \vdash_{taskset} \mathbb{A}''; \mathbb{N}''; \{\, T_1, \ldots, T_i, \ldots, T_n \,\}}$$

Figure 3.9. Typing rules for a parallel task T, and a set of parallel tasks $\mathbb{T}$.

immediately after the constructor tag, and there should be appropriate *after* constraints for other fields in the location constraint environment. To indicate that the tag has been written, the allocation environment is extended and the location l is removed from the nursery to prevent multiple writes to a location. If any fields of the constructor have evaluated to an `ivar`, however, the tag *is not* written to the store (D-Par-DataConstructor-Join). The rule T-DataConstructor-Ivars handles this case. It does not extend the allocation environment, and also keeps the location l in the nursery, so that a constructor tag may be written at this location in the future, once this task has synchronized with those allocating the fields. This additional typing rule is necessary to satisfy a requirement of the Top Level Preservation lemma (B.2.4).

To generalize our typing rules to handle multi-task configurations, we introduce new environments for variables $\mathbb{\Gamma}$, store typing $\mathbb{\Sigma}$, allocation constraints $\mathbb{C}$, allocation pointers $\mathbb{A}$, and nurseries $\mathbb{N}$. These environments—given in Figure 3.4—extend their counterparts in the sequential LoCal type system, and are needed to track state on a per-task basis. The precise typing rules to type check a parallel task T, and a set of parallel tasks $\mathbb{T}$ are given in the Figure 3.9. A parallel task T is well-typed if its target expression e is well-typed, using the original LoCal typing rules, and a task set $\mathbb{T}$ is well-typed if all tasks in it are well-typed.

3.2.4. Definitions of metafunctions. In the following, we define various metafunctions used in the formalism.

3.2.4.1. Merging task memories. These metafunctions are given in Figure 3.10. We merge two stores by merging the heaps of all the regions that are shared in common by the two stores, and then by combining with all regions that are not shared. We merge two heaps by taking the set of the all the heap values at indices that are equal, and all the heap values at indices in only the first and only the second heap. The merging of location maps follows a similar pattern, but is slightly complicated by its handling of locations that map to ivars. In particular, for any location where one of the two location maps holds an ivar and the other one holds a concrete index, we assign to the resulting location map the concrete index, because the concrete index contains the more recent information.

3.2.4.2. End-Witness judgement. The end-witness rule is given in Figure 3.11. This rule provides a naive computational interpretation of the process for finding the index one past the end of a given concrete location, with its given type. It is mostly the same as the one given for the original, sequential LoCal, but includes an additional case for handling indirection pointers. To compute the end-witness of an indirection

$$
\begin{array}{lcl}
\mathit{MergeS}(S_1, S_2) & = & \{\, r \mapsto \mathit{MergeH}(h_1, h_2) \mid (r \mapsto h_1) \in S_1, (r \mapsto h_2) \in S_2 \,\} \\
& \cup & \{\, r \mapsto h \mid (r \mapsto h) \in S_1, r \notin \mathit{dom}(S_2) \,\} \\
& \cup & \{\, r \mapsto h \mid (r \mapsto h) \in S_2, r \notin \mathit{dom}(S_1) \,\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{MergeH}(h_1, h_2) & = & \{\, i_1 \mapsto hv \mid (i_1 \mapsto hv) \in h_1, (i_2 \mapsto hv) \in h_2, i_1 = i_2 \,\} \\
& \cup & \{\, i \mapsto hv \mid (i \mapsto hv) \in h_1, i \notin \{\, i' | i' \in \mathit{dom}(h_2) \,\} \,\} \\
& \cup & \{\, i \mapsto hv \mid (i \mapsto hv) \in h_2, i \notin \{\, i' | i' \in \mathit{dom}(h_1) \,\} \,\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{MergeM}(M_1, M_2) & = & \{\, l^r \mapsto \langle r, i_1 \rangle \mid (l^r \mapsto \langle r, i_1 \rangle) \in M_1, (l^r \mapsto \langle r, i_2 \rangle) \in M_2, i_1 = i_2 \,\} \\
& \cup & \{\, l^r \mapsto cl \mid (l^r \mapsto cl) \in M_1, l^r \notin M_2 \,\} \\
& \cup & \{\, l^r \mapsto cl \mid l^r \notin M_1, (l^r \mapsto cl) \in M_2 \,\} \\
& \cup & \{\, l^r \mapsto \langle r, j \rangle \mid (l^r \mapsto \langle r, \texttt{ivar}\ x \rangle) \in M_1, (l^r \mapsto \langle r, j \rangle) \in M_2 \,\} \\
& \cup & \{\, l^r \mapsto \langle r, j \rangle \mid (l^r \mapsto \langle r, j \rangle) \in M_1, (l^r \mapsto \langle r, \texttt{ivar}\ x \rangle) \in M_2 \,\}
\end{array}
$$

FIGURE 3.10. Metafunctions for merging task memories.

Case (A). $\tau_c; \langle r, i_s \rangle; S \vdash_{ew} \langle r, i_e \rangle$:

(1) $S(r)(i_s) = K'$ such that
$\texttt{data}\ \tau_c = \overrightarrow{K_1\ \overrightarrow{\tau}_1} \mid \ldots \mid K'\ \overrightarrow{\tau}' \mid \ldots \mid \overrightarrow{K_m\ \overrightarrow{\tau}_m}$

(2) $\overrightarrow{w_0} = i_s + 1$

(3) $\overrightarrow{\tau'_1}; \langle r, \overrightarrow{w_0} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle \wedge$
$\overrightarrow{\tau'_{j+1}}; \langle r, \overrightarrow{w_j} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle$
where $j \in \{\, 1, \ldots, n-1 \,\}; n = |\overrightarrow{\tau'}|$

(4) $i_e = \overrightarrow{w_n}$

Case (B). $\tau_c; \langle r, i_s \rangle; S \vdash_{ew} \langle r', i'_e \rangle$:

(1) $S(r)(i_s) = \&(r', i'_s)$

(2) $\tau_c; \langle r', i'_s \rangle; S \vdash_{ew} \langle r', i'_e \rangle$

FIGURE 3.11. The end-witness rule.

pointer, the judgement first reads the address $\langle r', i'_s \rangle$ which is written at $\langle r, i_s \rangle$, and returns the end-witness of the value allocated at $\langle r', i'_s \rangle$.

3.2.4.3. Linking fields of a data constructor. Given a store S, a location map M, and the addresses of the n^{th} and $(n+1)^{th}$ fields of a data constructor K, this metafunction conditionally establishes a link between these fields. If the location map M maps the symbolic location of the $(n+1)^{th}$ field, ${l_2}^{r_1}$, to an indirection pointer $\&(r_2, i_2)$, it means that these fields were computed in parallel with each other, and thus would have been allocated into separate regions. To link such fields, we use the end-witness judgement to compute an address $\langle r_1, i_e \rangle$ which is one past the end of the n^{th} field, and we then write an indirection pointer pointing to the starting address of the $(n+1)^{th}$ field at $\langle r_1, i_e \rangle$. This establishes the desired link. If the fields have been allocated into the same region, we do not have to link them, and the store S is returned unchanged in this case. A precondition for using this metafunction is that the n^{th} field must be fully allocated. The layout of the heap corresponding to different schedules after establishing links between the fields is shown in Figure 3.14.

$$LinkFields(S, M, \tau_1, \langle r_1, i_1 \rangle^{l_1}, cl^{l_2}) = S \cup \{\, r_1 \mapsto (i_e \mapsto \&(r_2, i_2)) \,\}$$
$$\text{where } \; (l_2{}^{r_1} \mapsto \langle r_1, \&(r_2, i_2) \rangle) \in M \text{ and } \tau_1; \langle r_1, i_1 \rangle; S \vdash_{ew} \langle r_1, i_e \rangle$$

$$LinkFields(S, M, \tau_1, \langle r_1, i_1 \rangle^{l_1}, cl^{l_2}) = S$$
$$\text{where } \; (l_2{}^{r_1} \mapsto \langle r_1, \&(r_2, i_2) \rangle) \notin M$$

FIGURE 3.12. Metafunction for linking fields of a data constructor.

3.2.4.4. Derefrencing indirections in M. If a location l maps to an indirection pointer $\&(r', i)$ in M, it is derefrenced by returning the address $\langle r', i \rangle$. Otherwise, the mapping contained in M is returned unchanged.

$$\hat{M}(l) = \langle r, j\diamond \rangle \;\; \text{where} \;\; \begin{array}{l} (l \mapsto cl) \in M(l) \\ \langle r, j\diamond \rangle = Deref(M, cl) \end{array}$$

$$Deref(M, \langle r, \&(r', i) \rangle) = \langle r', i \rangle$$
$$Deref(M, \langle r, i \rangle) = \langle r, i \rangle$$
$$Deref(M, \langle r, \texttt{ivar}\ x \rangle) = \langle r, \texttt{ivar}\ x \rangle$$

FIGURE 3.13. Derefrencing indirections in M.

3.2.4.5. Other global environments and metafunctions.

- $Function(f)$: An environment that maps a function f to its definition fd.
- $Freshen(fd)$: A metafunction that freshens all bound variables in function definition fd and returns the resulting function definition.
- $TypeOfCon(K)$: An environment that maps a data constructor to its type.
- $TypeOfField(K, i)$: A metafunction that returns the type of the `i`'th field of data constructor K.
- $ArgTysOfConstructor(K)$: An environment that maps a data constructor to its field types.
- $MaxIdx(r, S) = \max(\{-1\} \cup \{ j \mid (r \mapsto (j \mapsto K)) \in S \})$: A metafunction that returns the highest allocated address in the store, or -1 if nothing has been allocated yet.
- $IsVal(e)$: A metafunction that checks if an expression is a value or not.
- $Ivars(e)$: A metafunction that yields the set of ivars that occur in the term e.
- $HasSingleWriter(\mathbb{T}, \hat{\tau}, \texttt{ivar } x) = |\{ (\hat{\tau}, \langle r, \texttt{ivar } x\rangle^l, S; M; e) \mid \exists_{S,M,e}.(\hat{\tau}, \langle r, \texttt{ivar } x\rangle^l, S; M; e) \in \mathbb{T} \}| = 1$
  A metafunction that checks if there is exactly one task $T \in \mathbb{T}$ which can supply a value of type $\hat{\tau}$ for `ivar` x.
- $GetSingleWriter(\mathbb{T}, \hat{\tau}, \texttt{ivar } x)$: A metafunction that returns a unique task $T \in \mathbb{T}$ which can supply a value of type $\hat{\tau}$ for `ivar` x if it exists, or returns -1 otherwise.
- $TaskComplete((\hat{\tau}, cl, S; M; e)) = IsVal(e)$: A metafunction that checks whether a task has evaluated to a value.
- $deepSupersetEqS(\mathbb{\Sigma}_1, \mathbb{\Sigma}_2) = \mathbb{\Sigma}_1 \supseteq \mathbb{\Sigma}_2 \vee ((cl \mapsto \Sigma_2) \in \mathbb{\Sigma}_2 \implies (cl \mapsto \Sigma_1) \in \mathbb{\Sigma}_1 \wedge \Sigma_1 \supseteq \Sigma_2)$: A metafunction that checks if the first argument is a *deep* superset of the second. In other words, the first argument can contain a mapping $(cl \mapsto \Sigma)$ which is not present in the second one and thus be a super set at the outer level. Or a specific mapping within the first argument can be a super set of the corresponding mapping within the second, and be a super set at an inner level. Note that it uses an inclusive-or, and both these conditions may be simultaneously true.
- $deepSupersetEqC(\mathbb{C}_1, \mathbb{C}_2)$: A metafunction identical to $deepSupersetEqS$, but for $\mathbb{C}$.
- $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$: A metafunction for the exclusive-or logical operator.

3.2.5. Well-formedness Judgments. In this section we present certain well-formedness criteria for various elements of the formal model. Some of these are similar to the corresponding judgments for sequential LoCal given in [**65**], but they are extended to handle ivars, and indirections in the location map

and store. We give an additional judgement to check the well-formedness of the set of tasks executing in a parallel machine. Because there are many requirements specified inside the various well-formedness judgments, we introduce notation for referring to requirements individually. For example, the notation WF 3.2.5.4;2 refers to the judgement $A; N \vdash_{wf_{ca}} M; S$, specified in Section 3.2.5.4, and in that judgement, rule number 2.

3.2.5.1. Well-formedness of a task set.

Judgement form. $\mathbb{\Sigma}; \mathbb{C}; \mathbb{A}; \mathbb{N} \vdash_{wf_{tasks}} \mathbb{T}$

The following well-formedness rule applies to a set of tasks executing in the parallel machine. This judgment specifies two new invariants that must hold for all tasks $T \in \mathbb{T}$. The first enforces that all ivars get filled with an appropriate value. In particular, if an expression being evaluated by a task references an ivar, then there must be exactly one other task in the task set which supplies a *well-typed* value for it. Rule 1 specifies that for all ivars referenced in an expression being computed by a certain task, there is a corresponding ivar in the location map, and there is exactly one other task in the task set which supplies a well-typed value for that ivar. The second invariant consists of the well-formedness judgment that verifies certain properties hold for each store of a given task. Rule 2 references a separate judgement for well-formedness of the store with respect to the location map of a parallel task. This judgment generalizes a similar rule used in the original proof by its use of the overall task set $\mathbb{T}$.

Definition.

$\forall(\hat{\tau}, cl, S; M; e) \in \mathbb{T}.$

$\Sigma = \mathbb{\Sigma}(cl);\ C = \mathbb{C}(cl);\ A = \mathbb{A}(cl);\ N = \mathbb{N}(cl)$

(1) $\langle r, \texttt{ivar}\ x\rangle^l \in Ivars(e) \wedge \emptyset; \Sigma; C; A; N \vdash A'; N'; \langle r, \texttt{ivar}\ x\rangle^l : \hat{\tau}' \Rightarrow$
$(l^r \mapsto \langle r, \texttt{ivar}\ x\rangle^l) \in M \wedge HasSingleWriter(\mathbb{T}, \hat{\tau}', \texttt{ivar}\ x)$

(2) $\Sigma; C; A; N; \mathbb{T} \vdash_{wf} M; S$

3.2.5.2. Well-formedness of a store.

Judgement form. $\Sigma; C; A; N; \mathbb{T} \vdash_{wf} M; S$

The well-formedness judgement for a parallel task's store extends the judgment of the sequential LoCal typing system to establish a global criterion for well-formedness, checking in particular that parts of regions that are distributed across each task-private store, given by $M; S$, are well-formed. This judgement is one of the most challenging parts of our extension, because it must be strong enough to ensure safe merging of stores when tasks meet at join points. Like in sequential LoCal, it specifies three categories of invariants.

The first category enforces that allocations occurring across the task-private stores are accounted for. In particular, for each symbolic location in the store-typing environment, $(l^r \mapsto \tau) \in \Sigma$, a value must be allocated to the appropriate store. There are two possible ways in which this allocation may occur: (1) sequentially, in the current task, or (2) in parallel, in a different task. In the sequential case, l^r's address in the location map M must be a concrete index, and it must have an end-witness. This technical point ensures that the store never contains partially allocated values. In the parallel case, l^r's address in M must be an ivar, and there must be exactly one other task, $T_{oth} \in \mathbb{T}$, that supplies a well-typed value for it. Moreover, l^r's address in T_{oth}'s location map must be a concrete index, and if T_{oth} has finished evaluating, this value must have an end-witness. This property ensures that when these tasks merge, the resulting store has *complete* values allocated at the expected addresses and the expected types.

The second category enforces that allocations occur in the sequence specified by the constraint environment C. Rules 2 and 3 reference the judgments for well-formedness concerning in-flight constructor applications (Section 3.2.5.3) and correct allocation in regions (Section 3.2.5.4), respectively. In particular, if there is some location l in the domain of C, then the location map and store must have the expected allocations at the expected types. The most interesting rule here is that for the `after` constraint, since it involves potential parallel allocations. For instance, if $(l \mapsto (\texttt{after } \tau@l')) \in C$, then the values at locations l and l' may be allocated sequentially, in the same task, or in parallel, in different tasks. The sequential case is straightforward. For the parallel case, there are two possibilities — (1) the task allocating the value at location l' may be still in-flight, or (2) it may have already synchronized with the current (parent) task. In the first case, we ensure the presence of an appropriate indirection in the location map $(l \mapsto \langle r, \&(r_{\text{fresh}}, 0)\rangle)$ and of a fresh region in the store ($r_{\text{fresh}} \in S$). Otherwise, we ensure that a link between the values at locations l' and l exists, which is accomplished by the metafunction *LinkFields*.

The final category enforces that each location is written to only once. This is done by checking that the nursery and store-typing environments reference no common locations ($dom(\Sigma) \cap N = \emptyset$), which is a way of reflecting that each location is either in the process of being constructed and in the nursery, or allocated and in the store-typing environment, but never both.

Definition.

(1) $(l^r \mapsto \tau) \in \Sigma \Rightarrow$
$(\langle r, i_1\rangle = \hat{M}(l^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle) \oplus$
$(\langle r, \texttt{ivar } x\rangle = \hat{M}(l^r) \wedge$
$\exists_{S', M', e'}.(\tau, \langle r, \texttt{ivar } x\rangle, S'; M'; e') = GetSingleWriter(\mathbb{T}, \hat{\tau}, \texttt{ivar } x) \wedge$

$\langle r, i_1 \rangle = \hat{M'}(l^r) \wedge$

$(IsVal(e') \implies \tau; \langle r, i_1 \rangle; S' \vdash_{ew} \langle r, i_2 \rangle))$

(2) $C \vdash_{wf_{cfc}} M; S$

(3) $A; N \vdash_{wf_{ca}} M; S$

(4) $dom(\Sigma) \cap N = \emptyset$

3.2.5.3. Well-formedness of constructor application.

Judgement form. $C \vdash_{wf_{cfc}} M; S$

The well-formedness judgement for constructor application specifies the various constraints that are necessary for ensuring correct formation of constructors, dealing with constructor application being an incremental process that spans multiple LoCal$^{\text{par}}$ instructions. Rule 1 specifies that, if a location corresponding to the first address in a region is in the constraint environment, then there is a corresponding entry for that location in the location map. Rule 2 specifies that, if a location corresponding to the address one past a constructor tag is in the constraint environment, then there are corresponding locations for the address of the tag and the address after it in the location map. Rule 3 specifies that, if a location corresponding to the address one past after a fully allocated constructor application is in the constraint environment, then there are corresponding locations for the address of the start of that constructor application, and for the address one past the constructor application in the location map. There are two possible ways in which the values at locations l'^r and l^r may be allocated — (1) sequentially, by a single task, and thus in a single region, or (2) in parallel with each other, by separate tasks, and thus in separate regions. The first disjunct holds when the values are allocated sequentially by a single task. In this case, the starting address of the constructor application is a concrete index, and an end-witness for it also exists. The address one past the constructor application is this end-witness. The second disjunct holds when the values are allocated in parallel by separate tasks. In this case, the location map M and store S belong to the task that allocates the value at location l^r. The starting address of the constructor application is an ivar, and the address one past the constructor application is an indirection pointer pointing to a separate region which has been allocated in the store. The third disjunct holds when the values are allocated in parallel by separate tasks, but after those tasks have synchronized with each other, and their memories merged (3.2.4.1). The merge resets the starting address of the constructor application back to a concrete index, and an end-witness for it now exists. The store contains an indirection pointer at this end-witness which points to the start of the region that contains the value at location l^r. In other words, a link between the values at locations l'^r and

l^r exists. The address one past the constructor application is the corresponding indirection pointer. Note that these disjuncts are connected with an *exclusive-or*, and only one of them can hold at a time.

Definition.

(1) $(l^r \mapsto \texttt{start}\ r) \in C \Rightarrow (l^r \mapsto \langle r, 0\rangle) \in M$

(2) $l^r \mapsto (l'^r + 1)) \in C \Rightarrow \langle r, i_l\rangle = \hat{M}(l'^r) \wedge \langle r, i_l + 1\rangle = \hat{M}(l^r)$

(3) $(l^r \mapsto \texttt{after}\ \tau@l'^r) \in C \Rightarrow$

$(\langle r, i_1\rangle = \hat{M}(l'^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle \wedge \langle r, i_2\rangle = \hat{M}(l^r)) \oplus$

$(\langle r, \texttt{ivar}\ x_1\rangle = \hat{M}(l'^r) \wedge (l^r \mapsto \langle r, \&(r_2, 0)\rangle) \in M \wedge \{\ r_2 \mapsto h\ \} \in S) \oplus$

$(\langle r, i_1\rangle = \hat{M}(l'^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle \wedge (l^r \mapsto \langle r, \&(r_2, 0)\rangle) \in M \wedge S(r)(i_2) = \&(r_2, 0))$

3.2.5.4. Well-formedness concerning allocation.

Judgement form. $A; N \vdash_{wf_{ca}} M; S$

The well-formedness judgement for safe allocation specifies the various properties of the location map and store that enable continued safe allocation, avoiding in particular overwriting cells, which could, if possible, invalidate overall type-safety. Rule 1 requires that, if a location l^r is in both the allocation and nursery environments, i.e., that address represents an in-flight constructor application, then there is a corresponding location in the location map and the address of that location is the highest address in the store. Alternatively, the value at location l^r is allocated in parallel by a separate task, in a separate region, and its address is the corresponding indirection pointer. Rule 2 requires that, if there is an address in the allocation environment and that address is fully allocated, then the address of that location is the highest address in the store. Rule 3 requires that, if there is an address in the nursery, then there is a corresponding location in the location map, but nothing at the corresponding address in the store. Finally, Rule 4 requires that, if there is a region that has been created but for which nothing has yet been allocated, then there can be no addresses for that region in the store.

Definition.

(1) $((r \mapsto l^r) \in A \wedge l^r \in N) \Rightarrow$

$((l^r \mapsto \langle r, i\rangle) \in M \wedge i > \mathit{MaxIdx}(r, S)) \oplus (l^r \mapsto \&(r_2, i_2)) \in M$

(2) $((r \mapsto l^r) \in A \wedge \langle r, i_s\rangle = \hat{M}(l^r) \wedge l^r \notin N \wedge \tau; \langle r, i_s\rangle; S \vdash_{ew} \langle r, i_e\rangle) \Rightarrow i_e > \mathit{MaxIdx}(r, S)$

(3) $l^r \in N \Rightarrow \langle r, i\rangle = \hat{M}(l^r) \wedge (r \mapsto (i \mapsto K)) \notin S$

(4) $(r \mapsto \emptyset) \in A \Rightarrow (r \mapsto \emptyset) \in S$

3.2.6. Type safety. Compared to the original type-safety result proved for single-task LoCal, ours generalizes to parallel evaluation by requiring that, for any given multi-task configuration, either the program has fully evaluated or at least one task can take a step. As usual, we prove this theorem by showing progress and preservation. The main complication relates to the property that parts of the overall store are now spread across the individual stores of the tasks, whereas in the original proof there is only one store. In particular, our proof must establish that the store of each task remains well formed, even while that task waits on a data dependency, and moreover after the task joins with another task and their stores are merged. The complete proof is available in Appendix B.2.

With the well-formedness and typing judgments in hand, we can now state the type-safety theorem, shown below. This theorem states that, if a given task set $\mathbb{T}$ is well typed and its overall store is well formed, and if $\mathbb{T}$ makes a transition to some task set $\mathbb{T}'$ in n steps, then either all tasks in $\mathbb{T}'$ are fully evaluated or $\mathbb{T}'$ can take a step to some task set $\mathbb{T}''$.

Theorem 3.2.1 (Type Safety).

$$\begin{aligned} &\textit{If } \emptyset;\mathbb{\Sigma};\mathbb{C};\mathbb{A};\mathbb{N} \vdash_{taskset} \mathbb{A}';\mathbb{N}';\mathbb{T} \ \wedge\ \mathbb{\Sigma};\mathbb{C};\mathbb{A};\mathbb{N} \vdash_{wf_{tasks}} \mathbb{T} \\ &\textit{and } \mathbb{T} \Longrightarrow_{rp}^{n} \mathbb{T}' \\ &\textit{then, either } \forall T \in \mathbb{T}'.\ TaskComplete(T) \\ &\textit{or } \exists\, \mathbb{T}''.\ \mathbb{T}' \Longrightarrow_{rp} \mathbb{T}''. \end{aligned}$$

3.2.7. Controlling fragmentation. A consequence of LoCal$^{\text{par}}$ introducing fresh regions is that the schedule of evaluation dictates the way a value is laid out on the heap, as shown in Figure 3.14. Every choice to parallelize an intra-region allocation implies the creation of a new region and a new indirection, thereby introducing fragmentation. Thus, in addition to the usual task-scheduling overheads, in our system, a schedule that parallelizes too many allocations also leads to fragmentation. Conversely, effort at amortizing the overhead of parallelism simultaneously amortizes the overhead of indirections and region fragmentation. We return to this topic and address fragmentation along with parallelism granularity management in Section 3.3.1.

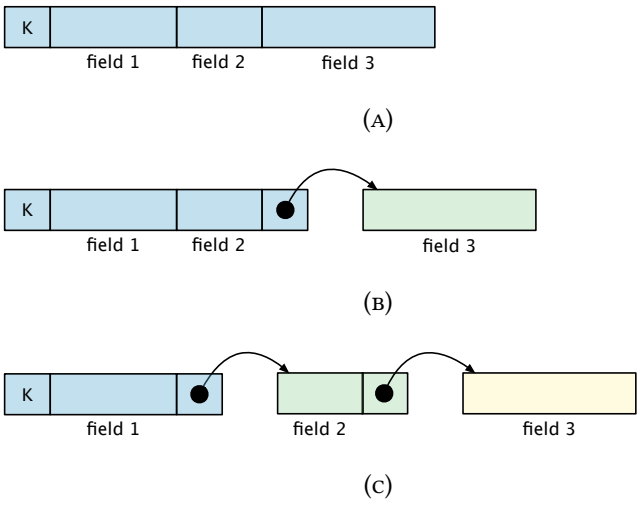


FIGURE 3.14. The heap layout for a data constructor if: (a) all fields are allocated sequentially, (b) only the second and third fields are allocated in parallel with each other, and (c) all fields are allocated in parallel.

3.3. Implementation

Gibbon is a whole-program[1] micropass compiler that compiles a polymorphic, higher-order subset of (strict) Haskell[2]. The Gibbon front-end uses standard whole-program compilation and monomorphization techniques [**18**] to lower input programs into a first-order, monomorphic representation. On this representation, Gibbon performs *location inference* to convert it into a LoCal program, which has region and location annotations. Then a big middle section of the compiler is a series of LoCal⇒LoCal compiler passes that perform various transformations. Finally, it generates C code.

Our parallelism extension operates in the middle section, with minor additions to the backend code generator and the runtime system. We add a collection of LoCal⇒LoCal compiler passes that transform the program so that reads and allocations can run in parallel. At run time, we make use of the Intel

[1]Gibbon's automatic selection of data representation works best if it can see the whole program, much like the data-representation optimizations in MLton. One way to get around this issue would be to make the programmer responsible for choosing the representation, by using appropriate annotations in datatype definitions. Another option is to conservatively insert random-access information in all datatypes that flow into code within other compilation units. Our current implementation does not offer these options, and only supports whole program compilation.

[2]Note that we are not the first to propose a strict variant of Haskell, not only do many of its cousins like Idris take a strict approach, but GHC itself supports a module-level strict mode.

Cilk Plus language extension [**14**] (and its work-stealing scheduler) to realize parallel execution. Our implementation closely follows the formal model described in Section 3.2.2, but with explicit parallelism annotations.

3.3.1. Granularity control. Before going further into the details of what we implemented, we first explain our choices in what we do and *do not* implement. As we saw in Figure 2.1a, we use manual annotations for the programmer to mark parallelism opportunities. This is the norm in both current and past parallel programming practice: from MultiLisp [**29**] to OpenMP [**43**], Cilk [**14**], Java fork-join [**36**], etc. Recall also that with a purely functional source language, parallel-tuple annotations change *performance only*, not program semantics, so a programmer need not worry about safety when inserting annotations[3]. Task granularity thresholds can be fine-tuned by using the same reasoning as in other parallel systems — switch to sequential for small problem sizes. But there's also the issue of fragmentation (Section 3.2.7), i.e. amortizing the overhead of pointers in the representation as well as parallel tasks in the control flow. One might wonder how these interact.

3.3.1.1. How to optimize granularity in Gibbon? Relatively small chunks of sequential data can effectively amortize the cost of creating regions and indirections. For instance, serializing just the bottom two levels in our binary tree examples eliminates 75% of pointers and ensures pointers use only 11% of memory. The task size to amortize parallel scheduling overheads, on the other hand, is usually much larger. Therefore, it's best the Parallel Gibbon programmer thinks about parallelism granularity exclusively, and the data representation can comfortably follow from that. In other words, a manual (or automatic) solution to task granularity, also gives "for free", an efficient, mostly-serialized data representation with amortized indirections.

3.3.1.2. Why not automatic granularity? *Automatic* task-parallel granularity control is an active research area [**4, 3**]. Combining Parallel Gibbon's automatic control over data representation, *together* with automatic granularity control, is a promising avenue for future work. This goes doubly so if approaches like *Heartbeat scheduling* [**4**] mature to the point of offering robust backends and runtime systems that compilers like Parallel Gibbon may target, and very recent work offers a step in that direction [**49**].

Here, however, it would be confounding to address automating granularity *simultaneously* with compacting data representation and assigning regions. In our experiments (Section 3.4), we hold task granularity constant across different implementations of the same benchmarks, focusing only on the impact

[3]This same property holds for inserting parallel annotations in pure GHC Haskell code, which has been used to modest benefit in past experimental work [**30**], but is not commonplace practice.

of each compiler's code generation and data representation choices. Parallel Gibbon, as well as all of its competitors, use explicit parallelism annotations, and schedule the same set of tasks at runtime for the same program inputs, unless mentioned otherwise.

3.3.2. Desugaring parallel tuples. As shown in Figure 2.1a, in the front-end language, we use the standard parallel tuples to express parallelism, like other eager, parallel functional languages [**50, 67**]. A parallel tuple `(e1 || e2)` marks the expressions `e1` and `e2` to evaluate in parallel with each other. To more closely match Cilk, we desugar these parallel tuples into a spawn/sync representation in the compiler IR:

```
let x = spawn e1 in
let y = e2 in
let _ = sync in
(x,y)
```

Using this representation simplifies the subsequent conversion to LoCal, in which additional steps like allocating regions or binding locations may be required before getting to `e1` or `e2`. Generating the corresponding `letregion`/`letloc` bindings, such that they have the correct scope, is easier with a spawn/sync representation. Also, we preserve these parallelism annotations in the LoCal code we generate. In contrast with the formal model (with implicit parallelism, Section 3.2), `let y = e2` is always sequential, whereas `let y = spawn e2` essentially corresponds to a *potentially* parallel let binding, though the decision is ultimately dynamic.

We do not support first-class futures, or tasks that communicate through channels or other mutable data structures, and thus the task-parallelism opportunities available in Parallel Gibbon remain effectively *series-parallel*. But this is sufficiently expressive for writing a large number of parallel algorithms. Note that the formal model can express some *local*, non-escaping futures that are not strictly series-parallel by using parallel lets that are forced out of order. This pattern of parallelism does not provide much additional expressive power over that provided by parallel tuples, so we do not give up much by not exposing this capability in the front-end language.

3.3.3. Indirection pointers. In the implementation, we need a runtime representation of optional pointers to include in the data that corresponds to the indirections in the semantics (Section 3.2.1). Fortunately, in the Gibbon compiler there is already a pointer mechanism that is sufficient for our purposes. This exists because of how Gibbon's regions grow—rather than copying data into a larger buffer, Gibbon accumulates a linked list of contiguous chunks, doubling the size on each extension. The last

filled cell in a chunk is an indirection into the next chunk. Indirections also enable Gibbon to allocate a value that is *shared* between multiple locations (within the same region or across regions) without requiring a full copy, which is crucial for ensuring asymptotic complexity conservation of programs. Variable aliases indicate this sharing to the compiler. For example, the aliased variable `x` in the expression `(let x = mkBigExpr in Plus x x)` indicates that a single, shared value must be allocated for the left and right subtrees of `Plus`. Gibbon rewrites this expression to `(let x = mkBigExpr in Plus x (IndPtr x))`, where the right subtree is an indirection pointing to the data allocated for the left. Similarly, the identity function `(id x = x)` becomes `(id x = IndPtr x)`. The shared scalar values such as numbers and booleans are always copied, because it is more efficient to do so. In Parallel Gibbon, we reuse this indirection pointer mechanism to implement intra-region parallel allocations.

3.3.4. Parallel reads. Using static analysis, Gibbon can infer if a dataype requires offsets, and it can transform the program to add offsets to datatypes that need them. In sequential programs, these are used to preserve asymptotic complexity of certain functions. In Parallel Gibbon, we use these offsets to enable parallel reads. We update that static analysis, and add offsets if a program performs parallel reads, i.e. via a clause in a case expression that accesses a data-constructor's fields in parallel.

3.3.5. Parallel allocations. The implementation of intra-region parallel allocations closely follows the design described in Section 3.2. A program transformation pass generates code that allocates fresh regions and writes indirection pointers at appropriate places. But the metafunctions *MergeS* and *MergeM* which merge task memories at join points have a different run time behavior compared to their formal definition. The implementation does not have a direct notion of a store. At run time, a region variable r is only a structure containing a pointer to the start of a memory buffer and some metadata necessary for garbage collection. Two memory buffers are merged (linked) simply by writing a single indirection pointer in one of them. This operation is relatively cheap compared to the set union used in the formal definition. Similarly, the implementation does not have a direct notion of a location map, and therefore there is no run time operation equivalent to *MergeM*. At run time, all location variables become absolute pointers into the heap.

But there still exists an issue with fragmentation. With granularity control — in the form of judicious use of parallel tuples — we can restrict excessive creation of fresh regions, but the number of regions created will still always be equal to the number of parallel tasks spawned by the program. This can still cause fragmentation because all spawned tasks might not actually *run* in parallel.

The key insight is to make the number of fresh region allocations equal to the number of *steals*, not spawns. That is, because our implementation uses a work-stealing scheduler, but the general idea applies to other schedulers as well: fragmentation should be proportional to parallelism in the dynamic schedule, not the static potential for parallelism. Our implementation creates fresh regions for intra-region parallel allocations only if they really run in parallel. We accomplish this by using the Cilk Plus API to implement a hook to detect when steals occur. Before reaching a parallel fork point (spawn), the runtime system stores the ID of the worker executing the current code. Next, the corresponding ID is immediately fetched in the continuation of the fork point. If the IDs match, it indicates that a steal did not occur. This optimization enables parallel allocations with minimal fragmentation.

3.3.6. Parallel arrays. Programs need arrays as well as trees. We extend Gibbon with array primitives such as `alloc`, `length`, `nth`, `slice`, and `inplaceUpdate`, and use them to build a small library of parallel array operations with good work and span bounds. The ability to *safely* mutate an array in-place allows us to implement optimizations that go beyond what is commonly allowed in a purely functional language. This is enforced using the new Linear Haskell extensions [**9**], for example, the signature of an $O(1)$ array mutation is:

```
inplaceUpdate :: Int → a → Array a ⊸ Array a
```

Using these primitive operations, collective operations on arrays are implemented as recursive divide-and-conquer functions in Parallel Gibbon that use parallelism annotations[4]. For example, our parallel `map` first allocates an array to store the output, and then updates it in parallel with `inplaceUpdate`. But all such potentially-racy operations are hidden behind a pure interface. Also, an `Array` in Parallel Gibbon can only store primitive values such as numbers, booleans, and n-ary tuples of such values. In the future, we plan to explore ways to support data-parallel operations on serialized algebraic data.

3.4. Evaluation

In this section, we evaluate our implementation using a variety of benchmarks from the existing literature, as well as a new compiler benchmark. To measure the latent overheads of adding parallelism, we compare our single-thread performance against the original, sequential LoCal, as implemented by the Gibbon compiler. Sequential Gibbon is also a good baseline for performing speedup calculations since

[4]These combinators offer variants to explicitly control sequential chunk size, or to use the common heuristic of splitting into a number of tasks that is a multiple of the number of cores (provided as a global constant).

its programs operate on serialized heaps, and as shown in prior work, are significantly faster than their pointer-based counterparts. Note that prior work [**66**] compared sequential constant factor performance against a range of compilers including GCC and Java. Since Sequential Gibbon outperformed those compilers in sequential tree-traversal workloads, we focus here on comparing against Sequential Gibbon for sequential performance.

We also compare the performance of our implementation to other languages and systems that support efficient parallelism for recursive, functional programs — MaPLe [**67**], Multicore OCaml [**37, 56**], and GHC. MaPLe (an extension of MLton) is a whole-program, optimizing compiler for Standard ML [**45**]; it supports nested fork/join parallelism, and generates extremely efficient code.

The experiments in this section are performed on a 48 core machine (96 hyper-threads) made up of 2×2.9 GHz 24 core Intel Xeon Platinum 8268 processors, with 1.5TB of memory, and running Ubuntu 18.04. The shared memory on this machine is divided into two NUMA nodes such that CPUs `0-23` and `48-71` use `node-0` as their local memory node, and `24-47` and `72-95` use `node-1`. In our experiments we only use 48 threads (no SMT), evenly distributed across both NUMA nodes (`numactl --physcpubind=48-95`). All experiments are performed using the default memory allocation policy which always allocates memory on the current NUMA node. We observed that using a round-robin memory allocation policy (option `--interleave=0,1`) did not affect performance, and therefore we do not report those results.

Each benchmark sample is the median of 9 runs. To compile the C programs generated by our implementation we use GCC 7.4.0 with all optimizations enabled (option `-O3`), and the Intel Cilk Plus language extension [**14**] (option `-fcilkplus`) to realize parallelism. To compile sequential LoCal programs we use the open-source Sequential Gibbon compiler, but we modify it to include arrays with in-place mutation using linear types, just like Parallel Gibbon. For MaPLe, we use version `20200220.150446-g16af66d05` compiled from its source code. For OCaml, we use the Multicore OCaml compiler [**56**] (version 4.10 with options `-O3`), along with the `domainslib` [5] library for parallelism. We use GHC 8.6.5, with options `-threaded -O2`, along with the monad-par [**40**] library for parallelism.

3.4.1. Benchmarks. We use the following set of 10 benchmarks. For GHC, we use *strict datatypes* in benchmarks, which generally offers the same or better performance, and avoids problematic interactions between laziness and parallelism. All programs use the same algorithms and datatypes (including mutable arrays, which are provably race-free in Gibbon and GHC), have identical granularity control thresholds,

[5] https://github.com/ocaml-multicore/domainslib

and are run with the same inputs. This way, each pairing of program and input creates a deterministic task graph — which does not change when varying the number of threads — and the evaluation focuses on data representation and code generation, rather than on decomposing and scheduling parallel tasks.

- **fib**: Compute the 48th fibonacci number with a sequential cutoff after depth=18: a simple baseline for scaling.
- **buildtreeHvyLf**: This is an artificial benchmark that is included here to measure parallel allocation under ideal conditions. It constructs a balanced binary tree of height 18, and computes the 20th fibonacci number at each leaf, with sequential cutoff after depth=12.
- **buildKdTree** and **countCorrelation** and **allNearest**: `buildKDTree` constructs a kd-tree [**25**] containing 1M 3-d points in the Plummer distribution. The sequential cutoff is at a node containing less than 32K points. `countCorrelation` takes as input a kd-tree and a list of 100 3-d points, and counts the number of points which are correlated to each one. The chunk-size for the parallel-map is 4, and the sequential cutoff for `countCorrelation` is at a node containing less than 8K points. `allNearest` computes the nearest neighbors of all 1M 3-d points. The chunk-size for the parallel-map is 1024.
- **barnesHut**: Use a quad tree to run an nbody simulation over 1M 2-d point-masses distributed uniformly within a square. The chunk-size for the parallel-map is 4096.
- **coins** This benchmark is taken from GHC's NoFib [6] benchmark suite. It is a combinatorial search problem that computes the number of ways in which a certain amount of money can be paid by using the given set of coins. The input set of coins and their quantities are `[(250,55),(100,88),(25,88),(10,99),(5,122),(1,177)]`, and the amount to be paid is 999. The sequential cutoff is after depth=3.
- **countNodes**: This operates on ASTs gathered from the Racket compiler when processing large, real programs. The benchmark simply counts the number of nodes in a tree. For our implementation, we store the ASTs on disk in a serialized format which is read using a single `mmap` call. All others parse the text files before operating on them. To ensure an apples-to-apples comparison, we do not measure the time required to parse the text files. The size of the text file is 1.2G, and that same file when serialized for our implementation is 356M. The AST has around 100M nodes in it. The sequential cutoff is after depth=9.

[6] https://gitlab.haskell.org/ghc/nofib

- **constFold**: Run the `constFold` function shown in Figure 2.1 on an artificially generated syntax-tree, which is a balanced binary tree of `Plus` expressions, with a `Lit` as a leaf. The height of the syntax-tree is 26, the sequential cutoff is after depth=8.
- **mergeSort**: An in-place parallel merge sort, which bottoms out to a sequential quick sort when the array contains less than 8192 elements. For our implementation, we use the `qsort` function from the C standard library to sort small arrays. The Haskell implementation is taken from Kuper et al's artifact accompanying their paper [**35**], and it makes an FFI call to a sequential quick sort written in C. MaPLe and OCaml bottom out to a sequential quick sort implemented in their source language. The input array contains 8M randomly generated floating point numbers.

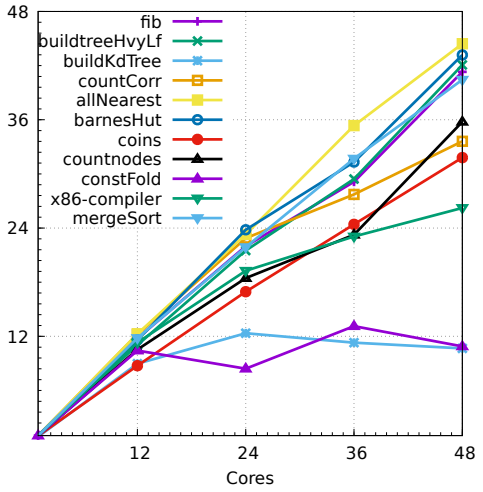
3.4.2. Results: Parallel versus Sequential Gibbon. Figures 3.15a and 3.15b show the results of comparing performance of benchmarks compiled using our parallel implementation, labeled "Ours", relative to Sequential Gibbon. The quantities in the table can be interpreted as follows. Column T_s shows the run time of a sequential program. which serves the purpose of a sequential baseline. T_1 is the run time of a parallel program on a single thread, and O the percentage overhead relative to T_s, calculated as $((T_1 - T_s)/T_s) * 100$. T_{48} is the run time of a parallel program on 48 threads and S is the speedup relative to T_s, calculated as T_s/T_{48}. R is the number of additional regions created to enable parallel allocations, calculated as $R_{48} - R_s$. For a majority of benchmarks, the overhead is under 3%, and the speedups range between 31.7× and 43.5×. These speedups match, or in cases such as `barnesHut` and `allNearest`, exceed those of optimized implementations that have been analyzed on similar machines [**54, 67, 4**].

With respect to the difference in speedups between different benchmarks, we see the expected relationship among them which reflects their memory access patterns. Compute-bound benchmarks such as `fib` scale very well, whereas benchmarks such as `constFold` and `buildKdTree` can become memory bound, and do not scale over a certain number of cores[7]. With respect to `buildKdTree`, a significant portion of its total running time is spent in sorting the points at each node. We observed that our `mergeSort` doesn't scale well on small inputs, and since `buildKdTree` performs a series of smaller and smaller sorts, it eventually runs into this, leading to lower scalability. But its high overhead (14.6%) and low speedup (10.6×) are in the same ballpark as an optimized C implementation which [**20**] analysed on a 32-core machine. Tables 3.2 and 3.3 show that MaPLe, GHC and OCaml also scale similarly. Overall, these results show that our technique is able to handle parallelism in a mostly-serialized data representation effectively.

[7]For example, a simple parallel dot-product computation (in Cilk) has a similar linear access pattern and low arithmetic intensity to `constFold`, and it achieves only a 6X speedup on this same machine.

	Gibbon	Ours			
Benchmark	T_s	T_1	O	T_{48}	S
fib	12.8	12.8	0	0.31	41.3
buildtreeHvyLf	4.69	4.69	0	0.11	42.6
buildKdTree	2.33	2.67	14.6	0.22	10.6
countCorr	1.46	1.47	0.68	0.044	33.2
allNearest	1.0	1.01	1	0.023	43.5
barnesHut	3.21	3.21	0	0.074	43.4
coins	3.04	3.13	3	0.096	31.7
countnodes	0.21	0.21	0	0.006	35.0
constFold	1.78	1.78	0	0.16	11.1
x86-compiler	1.08	1.08	0	0.041	26.3
mergeSort	1.58	1.60	1.27	0.039	40.5
average	-	-	1.28	-	29.5

(A) T_s is the run time of Sequential Gibbon. T_1 and T_{48} are the run times of Parallel Gibbon on 1 thread and on 48 threads respectively. O is the single-thread percentage overhead: $O = (T_1 - T_s)/T_s * 100$. S is the 48-thread speedup: $S = T_s/T_{48}$.



(B) Speedups relative to Sequential Gibbon.

FIGURE 3.15. Parallel Versus Sequential Gibbon.

Fragmentation. Traversals on a serialized heap are efficient because serialization minimizes pointer-chasing and maximizes data locality. But the heap produced by running intra-region allocations in parallel is fragmented, which can affect the performance of subsequent traversals that consume this heap, due to additional pointer dereferences and worse locality. To measure this downstream effect, we compare the run time of a single-threaded traversal operating on a sequentially allocated value to that traversal operating on a value allocated in parallel, which will be fragmented. Thus, the thing whose run time is being compared—the traversal—stays the same but it is given inputs that have different levels of fragmentation. We also measure the amount of fragmentation introduced for parallelism by counting the number of regions created solely to enable parallel allocations. Table 3.1 shows the results.

We use a subset of the benchmarks from Section 3.4.1 whose output is a serialized value (the other benchmarks do not measure the construction of new values), and measure the time required to *traverse* the output. For example, the benchmark `trav(constFold)` constructs an input expression, runs constant folding over it, and then sequentially traverses the resulting expression (counts the number of leaves in it), but the number reported is only the run time of the traversal and it does not include the time taken to run `constFold` itself. Each benchmark sample is the median of 9 runs, such that each run allocates a value and then traverses it N times, where N is set high enough to get the total run time over one second, and the run time of a *single* traversal is reported. T_s is the time required to sequentially traverse a sequentially allocated value, which is the baseline.

We compare against this baseline the performance of traversing a value allocated in parallel with two levels of fragmentation: *optimum* and *maximum*. In the *optimum* setting, the allocators are identical to those used for measurements reported in Figure 3.15a. That is, they control fragmentation by controlling the granularity of parallelism (using the thresholds given in Section 3.4.1) and are compiled using the *region-upon-steal* allocation policy (Section 3.3.5). R_{opt} is the number of *additional* regions created to allocate a value in parallel using 48 threads. For example, the sequential `constFold` uses a single region and its parallel version requires 132 additional regions (133 regions total). T_{opt} is the time required to sequentially traverse the allocated value, and O_{opt} is the percentage overhead relative to T_s, calculated as $O_{opt} = (T_{opt} - T_s)/T_s * 100$. In the *maximum* setting, the allocators do not control fragmentation at all (no granularity control), and they are compiled using the *region-upon-spawn* allocation policy, which creates a fresh region for every intra-region allocation task that is spawned. This setting thus represents the *upper bound* on the amount of fragmentation that can be introduced due to parallelism, where the heap essentially degenerates to a full pointer-based representation. R_{max}, T_{max} and O_{max} are the corresponding

TABLE 3.1. T_s is the time required to sequentially traverse a sequentially allocated value. R_{opt} is the number of *additional* regions created to allocate a value in parallel using 48 threads, T_{opt} is the time required to sequentially traverse it, and O_{opt} is the percentage overhead of the traversal: $O_{opt} = (T_{opt} - T_s)/T_s * 100$. R_{max}, T_{max} and O_{max} are the corresponding numbers for a value allocated in parallel using 48 threads with maximum fragmentation. O_{max} is the percentage overhead relative to T_s calculated as $O_{max} = (T_{max} - T_s)/T_s * 100$.

	Seq Alloc.	Optimum Fragmentation			Maximum Fragmentation		
Benchmark	T_s	R_{opt}	T_{opt}	O_{opt}	R_{max}	T_{max}	O_{max}
trav(buildtreeHvyLf)	0.99ms	341	1.02ms	3.03	262K	11.45ms	1056.6
trav(buildKdTree)	6.22ms	31	6.64ms	6.75	262K	55ms	784.2
trav(coins)	0.35s	9K	0.37s	5.71	75M	5.21s	1388.6
trav(constFold)	0.30s	132	0.32s	6.67	67M	2.70s	800
trav(x86-compiler)	366.2μs	1K	377.5μs	3.09	14K	464.3μs	26.8
geomean	-	-	-	4.74	-	-	476.9

numbers for this fragmentation setting when using 48 threads. O_{max} is the percentage overhead relative to T_s, calculated as $O_{max} = (T_{max} - T_s)/T_s * 100$.

Comparing R_{opt} and R_{max}, we see that in the optimum setting most of the allocated heap is still serialized and uses only a small number of additional regions—less than 0.13% of the maximum— in most cases. For `x86-compiler`, this percentage is higher (7.14%) compared to the other benchmarks because even in the optimum setting this benchmark does not control the granularity of parallelism, which is controlled only by the structure of the input as we discuss in Section 3.4.5. In this case, the *region-upon-steal* allocation policy is responsible for trimming the number of regions from 14K to 1K. With respect to the run time of the traversal, the overhead with optimum fragmentation is between 3.03% to 6.75%, with a geomean of 4.74%. In addition to fragmentation, the NUMA memory policy [8] has a significant impact on the overall overhead because in the parallel version potentially 50% of memory accesses are at a non-local NUMA memory node. If we run the experiment using a single NUMA node (`numactl --membind=0 --physcpubind=1-24,49-71`), the geomean overhead drops to 1.44%, with a significant reduction in the overheads for `buildKdTree` (0.96%)

[8] As described in the experimental setup, the shared memory on this machine is divided into two NUMA nodes and we use 48 threads evenly distributed across both nodes with the default *local* memory allocation policy.

and `constFold` (2.95%). In the presence of maximum fragmentation, we see the expected result: since heap degenerates to a full pointer-based representation, the traversals are several times slower and the benefits of using a serialized representation are lost. This slowdown in traversing a pointer-based representation compared to a serialized one is consistent with the results given in previous work [**65, 66**]. Overall, these results show that using the granularity of parallelism to guide the data representation works well in practice, and gives us an efficient, *mostly-serialized* representation.

3.4.3. Results: Gibbon versus other compilers. Table 3.2 shows the results of comparing performance of our implementation to MaPLe, OCaml, and GHC. For each compiler, Column T_s is the run time of a sequential program, Column T_{48} is the run time of a parallel program on 48 threads, and an adjacent column to each shows the corresponding speedup (or slowdown) of our implementation relative to this compiler. For example, on 48 threads, `allNearest` is 3.95× faster with our implementation compared to OCaml. Figures 3.16 and 3.17 show how these benchmarks scale on 48 threads. With respect to self-relative comparisons, on average, we scale similarly to MaPLe, and *better* than OCaml or GHC.

Across all benchmarks, on a single thread our Parallel Gibbon offers a 1.93×, 2.53×, and 2.14× geomean speedup compared to MaPLe, OCaml, and GHC, respectively. When utilizing 48 cores, our geomean speedup is 1.92×, 3.73× and 4.01×. Overall, these results show that we start with a faster baseline in the sequential world, and we're able to preserve the speedups in the parallel as well, meaning that the use of dense representations to improve sequential processing performance *coexists with scalable parallelism*. The only benchmark for which our implementation is slower compared to others is `coins`. This benchmark makes heavy use of linked-list operations such as `cons`, `head`, and `tail`, and our implementation uses `malloc` to allocate memory for every `cons`, which is inefficient. Also, our dense representation currently offers no benefit when building linked-lists by cons'ing onto existing lists. All others, MaPLe, OCaml and GHC, use a copying garbage collector [**57, 67, 39**] allowing them to use a bump-allocator, making them more efficient than our implementation. Table 3.3 gives the *self-relative* performance results for MaPLe, OCaml and GHC.

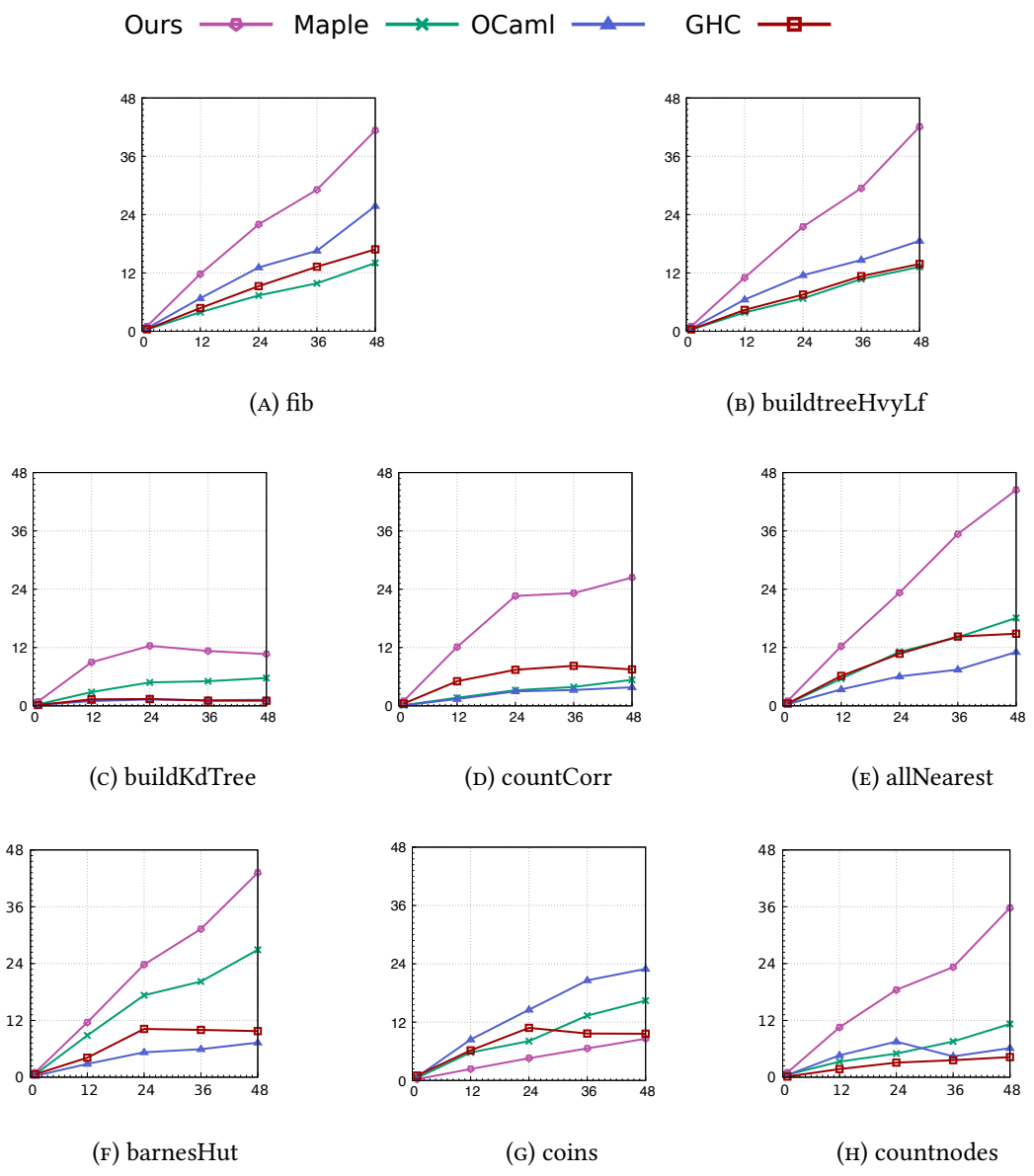
3.4.4. Results: Full evaluation details for other compilers. In Section 3.4, we presented the results for MaPLe, OCaml, and GHC by showing speedups/slowdowns of our implementation relative to them. In this section, we give the full evaluation results. Figure 3.3 shows the self-relative comparisions for MaPLe, OCaml and GHC. The quantities in the table can be interpreted as follows. Column T_s shows

TABLE 3.2. Comparison of Ours with MaPLe, OCaml, and GHC — execution time in seconds, and ratios to Ours. T_s is the run time of a sequential program, and T_{48} is the run time of a parallel program on 48 threads.

	MaPLe				OCaml				GHC			
Benchmark	T_s	$\frac{\text{MaPLe}}{\text{Ours}}s$	T_{48}	$\frac{\text{MaPLe}}{\text{Ours}}48$	T_s	$\frac{\text{OCaml}}{\text{Ours}}s$	T_{48}	$\frac{\text{OCaml}}{\text{Ours}}48$	T_s	$\frac{\text{GHC}}{\text{Ours}}s$	T_{48}	$\frac{\text{GHC}}{\text{Ours}}48$
fib	37.4	2.92	1.06	2.93	21.1	1.65	0.50	1.61	31.9	2.5	0.76	2.45
buildtreeHvyLf	14.5	3.09	0.35	3.18	8.60	1.83	0.25	2.27	12.4	2.64	0.34	3.09
buildKdTree	7.26	3.11	0.41	1.86	10.9	4.68	1.84	8.36	13.4	5.75	2.21	10.0
countCorr	10.5	7.19	0.27	6.14	13.9	9.52	0.37	8.41	3.54	2.42	0.15	3.41
allNearest	2.38	2.38	0.06	2.60	3.01	3.01	0.091	3.95	2.07	2.07	0.068	2.96
barnesHut	5.05	1.57	0.12	1.62	10.9	3.40	0.44	5.94	4.97	1.55	0.33	4.46
coins	1.71	0.56	0.05	0.52	1.05	0.34	0.036	0.37	0.82	0.27	0.085	0.88
countnodes	0.37	1.76	0.019	3.16	0.46	2.19	0.034	5.67	1.45	6.90	0.049	8.16
constFold	2.36	1.32	0.23	1.44	17.7	9.94	2.23	13.9	3.71	2.08	0.64	4.00
x86-compiler	1.34	1.24	0.042	1.02	1.20	1.11	0.09	2.20	2.34	2.16	0.44	10.7
mergeSort	1.74	1.10	0.047	1.20	3.83	2.42	0.19	4.87	2.74	1.73	0.16	4.10
geomean	-	1.93×	-	1.92×	-	2.53×	-	3.73×	-	2.14×	-	4.01×

the run-time of a sequential program, which serves the purpose of a sequential baseline. T_1 is the run-time of a parallel program on a single thread, and O the percentage overhead relative to T_s, calculated as $((T_1 - T_s)/T_s) * 100$. T_{48} is the run-time of a parallel program on 48 threads and S is the speedup relative to T_s, calculated as T_s/T_{48}. These comparisions are *self-relative*, meaning that they compare sequential MaPLe to parallel MaPLe, sequential OCaml to parallel OCaml, and sequential GHC to parallel GHC. With respect to self-relative comparisons, on average, we scale similarly to MaPLe, and *better* than OCaml or GHC, and the single-thread overhead across all four compilers is comparable to each other. Figure 3.16 shows speedups on 1-48 threads relative to the fastest sequential baseline, which is Sequential Gibbon for all benchmarks except `coins`, for which GHC is fastest.

3.4.5. Results: x86-compiler case study. As an example of a complex benchmark which performs multiple traversals over different datatypes, we implement a subset of a compiler drawn from a university



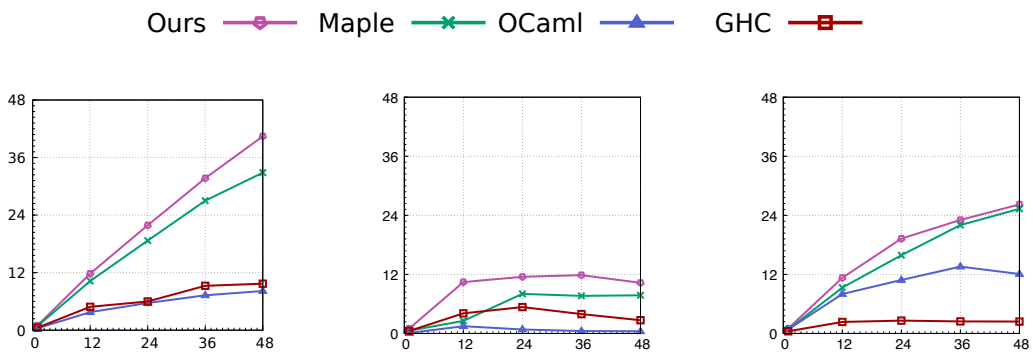
(A) fib (B) buildtreeHvyLf

(C) buildKdTree (D) countCorr (E) allNearest

(F) barnesHut (G) coins (H) countnodes

FIGURE 3.16. Speedups relative to the fastest sequential baseline, which is Sequential Gibbon for all benchmarks except `coins`, in which case sequential GHC is fastest.

course [**55**]. Our version compiles to x86, from a source language that supports integers and arithmetic and comparison operations on them, booleans and operations such as `and` and `or`, and a conditional expression, `if`. To compile this high-level language, we first translate it to an intermediate language similar to C, in which the order of evaluation is explicit in its syntax. The compiler is written in a nanopass style [**51**],



(A) mergeSort (B) constFold (C) x86-compiler

FIGURE 3.17. Remaining plots showing speedups relative to the fastest sequential baseline.

and is made up of five passes: (1) `typecheck` type checks the source program, (2) `uniqify` freshens all bound variables to handle shadowing, (3) `explicateControl` translates to an intermediate language similar to C, (4) `selectInstructions` generates x86 code which has variables in it, (5) and `assignHomes` maps each variable to a location on the stack. The input to this benchmark is a synthetically generated, balanced syntax-tree with conditional expressions at the top, followed by a sequence of let bindings. The structure of the input program is used to control the granularity of parallelism. The first three passes process the branches of conditionals in parallel, and subsequent ones process every *block* of instructions in parallel.

On this compiler benchmark, Parallel Gibbon offers a 1.24×, 1.11×, and 2.16× speedup on a single thread, and 1.02×, 2.20× and 10.7× speedup when utilizing 48 threads, compared to MaPLe, OCaml, and GHC, respectively. Note that most of the run time and also the self-relative parallel speedup of this benchmark is due to `assignHomes`. The first four passes of the compiler have a linear memory access pattern and low arithmetic intensity, much like `constFold`. The passes inspect the input expression, perform inserts or lookups on shallow environments (which only contain entries for variables that are in scope at a point), and then allocate the output. But `assignHomes` works in a different way. Since it needs to assign a unique stack location to every variable occurring in an expression, it constructs an environment containing *all* variables in the expression (as opposed to just those in scope at a point), and then performs repeated lookups on it to rewrite variable occurrences with stack location references. This environment is significantly larger than those used by other passes, making `assignHomes` much more work-intensive and suitable for parallel execution.

TABLE 3.3. Execution time in seconds, self-relative overheads on a single thread (Columns 3, 8 and 13) and self-relative speedups on 48 threads (Columns 5, 10 and 15). T_s is the run-time of a sequential program. T_1 and T_{48} are the run-times of a parallel program on 1 thread and on 48 threads respectively. O is the single-thread percentage overhead: $O = (T_1 - T_s)/T_s * 100$. S is the 48-thread speedup: $S = T_s/T_{48}$.

	MaPLe					OCaml					GHC				
Benchmark	T_s	T_1	O	T_{48}	S	T_s	T_1	O	T_{48}	S	T_s	T_1	O	T_{48}	S
fib	37.4	38.6	3.2	0.91	41.1	21.1	21.1	0	0.5	42.2	31.9	31.8	-0.3	0.76	42
buildtreeHvyLf	14.5	14.6	0.69	0.35	41.4	8.6	8.6	0	0.25	34.4	12.4	12.4	0	0.34	36.5
buildKdTree	7.26	7.65	5.37	0.41	17.7	10.9	11.7	7.34	1.84	5.92	13.4	13.6	1.5	2.21	6.06
countCorr	10.5	10.5	0	0.27	38.9	13.9	15.0	7.9	0.37	37.6	3.54	3.57	0.84	0.15	23.6
allNearest	2.38	2.4	0.84	0.06	39.6	3.01	3.09	2.65	0.091	33.1	2.07	2.04	-1.4	0.068	30.4
barnesHut	5.05	5.33	5.5	0.12	42	10.9	11	0.91	0.44	24.7	4.97	5.29	6.4	0.33	15.1
coins	1.71	1.51	-11%	0.05	34.2	1.05	1.07	1.9	0.036	29.1	0.82	0.82	0	0.085	9.6
countnodes	0.37	0.38	2.7	0.019	19.5	0.46	0.45	-2.17	0.034	13.5	1.45	1.46	0.68	0.049	29.6
constFold	2.36	3.29	39.4	0.23	10.3	17.7	19.2	8.5	2.23	7.94	3.71	3.99	7.55	0.64	5.80
x86-compiler	1.34	1.33	-0.74	0.042	31.9	1.2	1.3	8.33	0.09	13.3	2.34	2.38	1.7	0.44	5.31
mergeSort	1.74	1.84	5.75	0.048	36.3	3.83	3.94	2.87	0.19	20.16	2.74	2.95	7.66	0.16	17.1
average	-	-	3.29	-	33.7	-	-	4.14	-	15.3	-	-	1.35	-	15.2

Here we have only taken the first step towards developing an efficient parallel compiler, and there is ample opportunity for further investigation in this area. In the future, we plan to expand the compiler's source language to include constructs such as top-level function definitions and modules which are a source of parallelism in many real compilers.

Memory Management for Mostly-Serialized Heaps

In this dissertation, we propose a garbage collector that takes Gibbon from a domain-specific tool for tree traversals to a *general-purpose functional language implementation*. We adopt the tried-and-true design of the tracing, generational collector, which is used by many state-of-the-art implementations of functional languages. Gibbon's denser data representations violate many of the usual assumptions baked into many memory management systems, making them unsuitable to be directly used. In this setting, the solution we propose is a generational collector with a copying (and compacting, and pointer-eliminating) collector for its younger-generation, while keeping Gibbon's reference-counted-regions for the older generation, which now become partial/deferred reference counts[1], similar to Ulterior Reference Counting [**11**]. The new system, GC-Gibbon, includes several novel features to deal with the mostly-serialized setting. We must introduce new regions during collection time, to keep all roots valid after collection (Section 4.1.1). Because values continue growing after promotion to the old-generation, we choose to allow allocation into old-generation regions (Section 4.1.2), which in turn incurs a need for a remembered set. we allow sharing, in spite of the fact that many objects are smaller than a word-sized pointer, and thus develop a new approach to forwarding pointers (Section 4.1.3). In this chapter:

- We introduce a new garbage collection algorithm that deals with dense data, where the presence of pointers or header objects cannot be counted on. The hybrid approach of reference-counted regions and copying collection enables fast bump-allocation of new objects and regions, while retaining efficient handling of large and growing regions.
- We present a suite of solutions to the above challenges (Section 4.1) that enable the first practical, complete, and correct solution to automatic memory management in mostly-serialized heaps.
- We evaluate the resulting system (Section 4.3) on both tree-traversals, and benchmarks that stress small object allocation and collection. This new approach retains Gibbon's strong performance on tree-traversals (where the serialized data approach is most effective) and improves Gibbon to

[1]Heap cycles are not possible in a pure, strict setting, but we still need backup tracing collection of the old-generation, for defragmentation.

achieve reasonable performance on out-of-order small allocations where the approach is weakest, more closely resembling mature compilers and runtime systems that have been heavily optimized for such programs, using traditional memory representations. For small allocations, our system is 3.79×, 0.46×, and 1.09× geomean faster than Legacy-Gibbon, GHC, and Java, respectively. For bulk-tree-traversals, our geomean speedup is 1.02×, 2.19× and 1.5×.

4.1. Design

The main challenge we address is that of efficiently copying mostly-serialized objects, while maintaining sharing. To maintain sharing efficiently, our collector has to cope with many objects being smaller than a word-sized pointer, there being partially written objects, and there being pointers into the middle of objects at arbitrary byte offsets.

A collection starts when the mutator[2] requests a new chunk and the resulting heap size would exceed a certain threshold[3]. All new chunks are allocated in the *young generation* (or, nursery) by bumping the allocation pointer. When it becomes full, the young-generation is collected by a copying collector. Although it is common to divide the young-generation into equal *semispaces* [**17**], our design does not. Instead, we use single slab of memory to serve as the young-generation, and we have the old-generation serve directly as the old-generation (i.e., the destination for copying data during collection).

In the next few sections, we concern ourselves with how we treat a *minor collection*, tracing the young-generation, and copying to a different region representation in the old-generation where regions are granted their own growing, memory allocations, and equipped with extra metadata. At the end of minor collection, old-generation objects may be freed due to (deferred) reference count changes, yet old-generation tracing does not occur until a *major collection* (Section 4.1.5).

4.1.1. GC roots and evacuation. We use a *shadow-stack* [**31**] to maintain a root set of live objects for collection. Following the convention, addresses of objects (locations in LoCal) that are live after an allocation point (`letregion`) are spilled to the shadow-stack, and are restored later, potentially having been updated by an intervening collection. Spilling is also necessary across function call sites, as the function body might perform allocations. Along with the spilled location, we also spill and restore the end-of-chunk

[2]The user program, which *mutates* the heap.

[3]There is only one expression whose evaluation may trigger a collection—`letregion`—which allocates the initial chunk for a new region. Perhaps unintuitively, the application of a data constructor never triggers a collection, although it may grow a region, which involves adding chunks directly to the old-generation Section 4.1.2.

address for the chunk that the spilled location points into. There are two reasons: (1) if the location points into a young-generation chunk, it will be promoted to the old-generation and its end address will change, and (2) if it points into an old-generation chunk, the collector would require its end address to access the region-level metadata (reference count, outset etc.) whose address is stored at the footer. Also, the *type* of the value residing at the spilled location is stored on the shadow-stack. It serves as a numeric index into an *info-table* (described in Section 4.2) that stores layout information required to guide the evacuation routine. Each frame on the shadow-stack is represented as follows:

```
struct shadowstack_frame { char *ptr; char *endptr; uint32_t type; };
```

We now present the algorithm—given in Figures 4.1, 4.2, 4.3—to evacuate each root in the root set. Here we focus our attention on evacuating *complete* objects *without* maintaining sharing. We will discuss how we handle incomplete objects and maintain sharing extensively in subsequent sections. The main entrypoint of the algorithm is `evacuate_root` (given on line 5 in Figure 4.1). It first checks whether a young-generation object has already been evacuated, in which case it writes to the start and end addresses of the root the corresponding addresses of the relocated data in the old-generation (using the forwarding pointer mechanism described in Section 4.1.3). Otherwise, it allocates a fresh region in the old-generation to relocate this object. This fresh region can grow during collection, using the normal policy of doubling the size of each additional chunk in the linked series of chunks. Then, it learns the kind of object it is evacuating by inspecting the first tag in the object, after which it carries out the evacuation accordingly. After completing evacuation, the start and end addresses of the root are updated the same as before.

To evacuate an object (line 3 in Figure 4.2) it is copied from the source address, namely `from_start`, to the destination address, namely `to_start`. Recall that an object has two subcomponents, a fixed portion consisting of a tag and constant-sized fields, and an extended portion containing variable number of bytes due to the child objects. The fixed portion is copied directly using `memcpy`, and the child objects are processed by recursively inlining them into the destination region. We use a worklist (line 18 in Figure 4.2) instead of call-stack based recursion since that is more efficient in our experience.

The first tag of the object can be one of the following, which informs how it will be evacuated:

- **Tagged indirection pointer:** If the target of the pointer is in the young-generation, it is *inlined* by copying its data. Otherwise, the indirection pointer is copied as it is. In the latter case, the indirection pointer written by the collector is an *old-to-old* pointer, and thus the reference count

```
global info_table[] -- store type layouts per data constructor (static)
global fwd_table[]  -- map spans of young-generation memory to old-generation
global skip_table[] -- map address of value start to its end

fun evacuate_root(from_start, ty):
  if in_oldgen?(from): -- skip root
  else if already_marked?(from): update_root(fwd_addr(from_start))
  else: (to_start, to_end) := alloc_oldgen_region()
        evacuate_object(from_start, ty, to_start)
        update_root(to_start, to_end)

-- Returns a pointer into the old-generation, after the given value
fun skip_over(from):
  assert(not(zero_location?(from)))
  return skip_table[from]

-- Returns a pointer in the old-generation
fun fwd_addr(from):
  -- For the first location in the region, the region metadata lets us forward.
  if zero_location?(from):  return footer_fwd_ptr(region(from))
  else:  while offset < max_scan:
           let next = deref(from + offset)
           match(next):
             Forwarded(addr): return (addr - offset)
             Burned: offset += 1
         -- The interval-map maps source to target bytes by internally mapping entire
         -- (src..src+k) ⇒ (dst..dst+k) ranges efficiently:
         return fwd_table[from] -- side metadata lookup = slow path
```

FIGURE 4.1. Part of the evacuation algorithm. Global definitions, helpers, and entrypoint to begin evacuation. See also Figures 4.2 and 4.3.

```
-- Recursively evacuate the value at a given location, with the given type. Because of the acyclic heap,
-- this will never recur back to the same location.
fun evacuate_object(from_start, ty, to_start):
  let ty_stk = [ty]
  let to = to_start
  let from = from_start
  let span_start = from_start
  let span_bytes = 0

  fun end_span():
    if span_bytes > max_span:
      let sz = from - span_start
      fwd_table[span_start .. sz] := to-sz .. to
    span_bytes := 0
    span_start = from

  -- One evacuate-and-burn session on one contiguous interval:
  while (from_ty = pop(ty_stk)):
     while (chunk_redirection?(from)):
       forward_obj(from, to)
       from := deref(from)
       end_span()
     if indirection?(from):
       -- start a separate interval in a new chunk:
       to' = evacuate_object(deref(from), from_ty, to)
       forward_obj(from,to)
       from += TAGGED_INDIRECTION_SIZE
       to := to'
       end_span()
```

FIGURE 4.2. Continued from figure 4.1, core burning and forwarding algorithm for sharing maintenance.

```
    else if already_marked?(from):
      to := write_indirection(to, fwd_addr(from))
      if non_empty?(ty_stk):
        from := skip_over(from)
      end_span()
    else: -- regular data-copying codepath
      -- advance to,from cursors past the data written:
      (to',from') =
        write_tag_and_scalars(to, from)
      -- look up types of 0 or more non-scalar data fields (children):
      push_children(stk, info_table[from])
      fwded = burn_or_forward_obj(from)
      to   := to'
      from := from'
      if fwded: end_span()
      else: span_bytes += from' - from

  if not(zero_location?(from_start)):
    -- Populate for future slowpath lookups one byte after the end
    skip_table[from_start] := to

return to
```

FIGURE 4.3. Continued from figure 4.2, rest of the forwarding algorithm.

of the region containing the target object must be updated, as our (deferred) reference counts include only old-generation pointers.

Updating reference counts requires accessing the target region's metadata, which must be findable given only the contents of the tagged indirection pointer being copied. To accomplish this we use the 64-bit indirection pointers to pack in both (1) the address of the target object, and (2) the offset from there to the target chunk's footer—which is where the region metadata resides. We give details of this pointer encoding in Section 4.2.

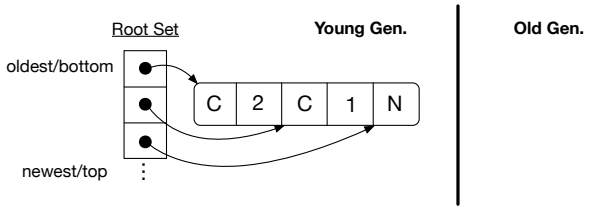
Recall that the mutator uses indirection pointers to share objects among different regions. Thus, there is additional work required here to carry forward this sharing into the old generation. We postpone this discussion to Section 4.1.3.

- **End-of-chunk pointer:** Due to the eager promotion policy (Section 4.1.2) the target of an end-of-chunk pointer is always in the old-generation, and thus an object is considered completely evacuated upon reaching this tag. But, we still need to *combine* metadata information for two regions: (1) the fresh region that was created to copy the object that ends in this end-of-chunk pointer, and (2) the old-generation region that was created earlier due to eager promotion to store the remainder of this object. For this reason, all end-of-chunk pointers also need to encode an offset from their target to the target chunk's footer. After this step, one of the region metadata objects is deleted since an end-of-chunk pointer always links two chunks of the same *logical* region.
- **Burned or forwarded object:** Objects which have already been evacuated before are marked as *burned* or *forwarded*, depending on whether there's enough room for a forwarding pointer. We discuss these extensively in Section 4.1.3.
- **Regular data constructor:** If the tag is not among the reserved tags listed above, it corresponds to an allocation of a regular data constructor and is copied by referring to the info-table (Section 4.2). Its scalar data (fixed portion) is copied using `memcpy` and its child objects (data constructor fields) are pushed on the worklist for evacuation. Any untagged shortcut pointers stored in this constructor have to re-created to point to the new addresses of the relocated child objects. The creation of new shortcut pointers is *deferred* until its corresponding child object begins evacuation, since the child's new address cannot be known in advance. In the implementation, the worklist stores a pair `(Type, Option<*mut i8>)`, where the `(Option<*mut i8>)` corresponds to the address of the potential[4] shortcut pointer that needs to be created when this object begins evacuation.

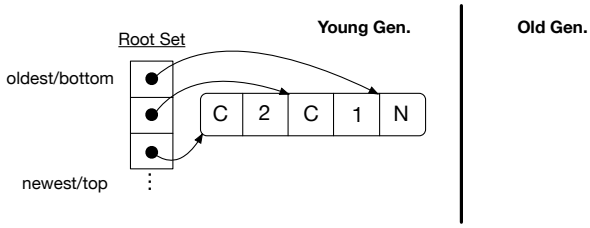
After all GC roots have been evacuated, any old-generation regions that are dead as per the deferred reference counting mechanism are freed. The minor collection is now considered complete and the mutator resumes execution. We discuss the major collection strategy in Section 4.1.5.

Evacuation order of GC roots. Like other tracing collectors, the layout and locality of the old-generation heap produced by our collector is sensitive to the order in which roots are processed [**2**]. But

[4]Not all data constructors store shortcut pointers.



(A) Evacuating GC roots from oldest to newest would create a compact old-generation object with no indirection pointers.



(B) Evacuating GC roots from oldest to newest would create an old-generation object with unnecessary indirections.

FIGURE 4.4. Figure (a) shows a young-generation object, and a root set representative of in-order allocation workloads. Figure (b) shows the same young-generation object, but a root set representative of reverse-order allocation workloads. These root sets have different traversal orders which are *efficient*.

our collector is even more sensitive to this ordering. Certain traversal orders produce a compact heap with no indirection pointers, while some others produce significantly pointer-heavy heaps. Consider the young-generation and the root set in Figure 4.4a and Figure 4.4b. The young-generation object is identical in both cases, but the order of GC roots corresponding to the child objects is different. In Figure 4.4a, the oldest root corresponds to the parent object, and roots toward the top correspond to child objects deeper in the graph. In Figure 4.4b, the roots are ordered in the opposite order. This ordering of roots in the root set is a consequence of different workloads: in-order allocators like `mkList` tend to create root sets like Figure 4.4a, while reverse-order allocators like `reverse` tend to create root sets like Figure 4.4b. If a collection is triggered at this point, there is no fixed order of root set traversal (e.g. newest-to-oldest versus oldest-to-newest) that would handle both examples efficiently[5]. For these list examples, an efficient traversal starts evacuation with the *head* of the list, promoting the entire list into a single old-generation region

[5]We say that a traversal order is efficient if it produces compact heaps and introduces minimum fragmentation.

(Figure 4.4a). An inefficient traversal would evacuate the tail of the list first, and then subsequently evacuate earlier parts of the list which would be linked by (unnecessary) indirections to the already-evacuated portions (Figure 4.4b). How can we reconcile different efficient traversal orders?

The key insight here is to pick an order that consistently evacuates *upstream* data earlier, irrespective of the order of allocation. Specifically, we want a traversal order that (1) evacuates roots for the newer regions before the older regions, and (2) within each region, evacuates roots that are towards the beginning of the region before roots that are towards the end.

We get such an ordering as follows. First, we bump the allocation pointer of the young-generation *backwards*. That is, the allocation pointer starts at the end of the young-generation (the high address), and moves towards the start (the low address). Next, we sort the root set such that roots corresponding to objects at lower addresses appear before those at higher addresses. This policy gives us both the desired properties because objects towards the beginning of the region already occupy addresses lower than those of objects towards the end, and the reversed bump allocation pointer puts newer regions at addresses lower than those of older regions.

For a desirable post-collection heap, indirections should only be proportional to actual *sharing* in the data. To quantify this notion more precisely, we define an optimal heap with minimum indirection count:

Definition 4.1.1: Minimal indirections post-collection
The minimum number of indirections post-collection includes: up to one per root, plus $N - 1$ indirections for every object which has $N > 1$ references to it.

If a live object has only a single reference to it, then it must be placed in the same region as the object from which it is reachable. The per-root indirections are there because the roots themselves represent pointers into the heap data. Furthermore, we currently place these pointers in fresh regions only if the object has not been copied already, otherwise, the root is directly updated to point to the object's new address. Our post-collection heaps always achieve minimal, sharing-only indirections, as in Definition 4.1.1. Of course, this compactness comes at the cost of sorting the roots. In practice, root sets are small enough that the time is worth it for this optimization. Also, the root set is bounded by the program stack size. If we wanted to further bound sorting time to a constant, we could use a *partial sorting* algorithm to limit sorting time, trading it off against having fewer of the young-generation indirection-edges respected by the order of the root-set traversal.

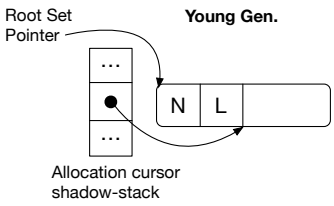
4.1.2. Growing partially-written objects. Certain objects encountered by GC-Gibbon's copying collector might be only *partially written*. For example, the mutator could be in the middle of allocating a tree structure when it triggers a collection, which would cause the young-generation to contain a region with a tree node having a *left* field but no *right* field (yet), as shown in Figure 4.5a. When such a tree node is promoted to the old-generation, the collector must *stop copying* after the left field, otherwise it will keep reading uninitialized data. Thus the collector must be able to detect the ends of such partially-written objects. Furthermore, once the minor collection is complete the collector must decide *where* to grow this object, that is, where to restart construction of the remainder of the object (the right field)—in the young or in the old-generation.

We use a region's *allocation cursor* to detect partially-written objects. LoCal always allocates objects in a region *in order*, and, the allocation cursor is the address where the next object would be written. As a corollary, the allocation cursor is the frontier of all data written within the region, or in other words, the first byte address in a region that is uninitialized. This is the address where we want the copying to stop.

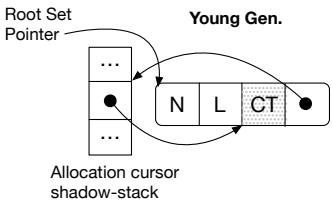
Before beginning copying, the collector writes a special reserved tag at all live allocation cursors, effectively *cauterizing* the regions to mark the end of initialized data, as shown in Figure 4.5b. The copying routine described in Section 4.1.1 stops copying upon reaching this tag, so as to not read any uninitialized data. To support this, the mutator spills all live writeable locations to a separate shadow-stack before starting a collection, and restores their updated addresses after the collection is complete. Regions that contain a fully constructed value do not have an allocation cursor, as they do not have any writeable locations in them. Correspondingly, such regions do not undergo cauterization and the live objects within them are promoted in the standard way.

The design choice of where to restart construction of the remainder of partially-written values is a tradeoff between (1) requiring a write barrier for new indirection pointers written into the old-generation, and (2) sacrificing the benefits of pretunuring large and growing regions into the old-generation.

4.1.2.1. Design choice 1: Restarting construction in the young-generation. We would allocate one young-generation chunk for each partially-written object that was promoted to the old-generation, and, to continue its construction, update its allocation cursor to point to the beginning of the new chunk. Next, we would use an end-of-chunk pointer to link the end of the promoted object to the beginning of this fresh young-generation chunk. These end-of-chunk pointers (pointing from old-generation to young-generation) would serve as a *remembered set* of roots for the subsequent minor collection. Crucially, however, all writes would now always happen in the young-generation. As a result, and since LoCal is a pure



(A) A partially-written tree node in the young-generation; `N` is short for `Node` and `L` is short for `Leaf`.



(B) The tree node from Figure 4.5a *cauterized* by writing the reserved tag `CT`. A pointer to the allocation shadow-stack frame is saved here so that the allocation pointer can be restored after collection.

FIGURE 4.5. Partially-written values in the young-generation.

language, this remembered set would remain constant until the subsequent collection. Moreover, all new indirection pointers would be young-to-old pointers, and could be created *without a write barrier* (for maintaining a separate remembered set), making them fairly cheap to create. With this policy, we would use a remembered set of end-of-chunk pointers that is updated once per collection, instead of a remembered set of indirection pointers that is maintained by the mutator using a write barrier.

While this policy reduces the cost of indirection pointers, it precludes the collector from performing pretenuring, which requires that the mutator be able to allocate certain long-lived objects directly in the old-generation. This would have a significant impact on the performance of programs that allocate large values, often using a small number of regions. Such programs would exhaust the nursery frequently and trigger a collection. Moreover, programs that have a large number of regions under construction simultaneously could cause an exceptional situation where after the minor collection, a large portion of the nursery is populated by these new chunks for older promoted values, also increasing the number of collections (unless the nursery is allowed to grow).

4.1.2.2. Design choice 2: Restarting construction in the old-generation. In GC-Gibbon we use the dual of the previous choice, namely to continue the construction of partially-written objects in the old-generation. After an object is promoted, its allocation cursor is updated to point to the frontier of its old-generation chunk, and no new chunks are created in the nursery. As discussed above, the choice to allow allocations directly in the old-generation has two consequences: the collector can perform pretenuring, but pointer creation needs to be protected by a write barrier since the remembered set of indirections pointing

from old to young-generation can dynamically grow[6]. While such a write barrier is expensive, we already amortize its overhead by minimizing the number of indirection pointers. Also, this write barrier is no worse than what already exists in Gibbon, which has to potentially update reference counts and outsets when creating indirection pointers. On the plus side, pretenuring is vastly beneficial for programs that allocate large data structures—exactly the kind of bulk-data-processing programs which are Gibbon's speciality.

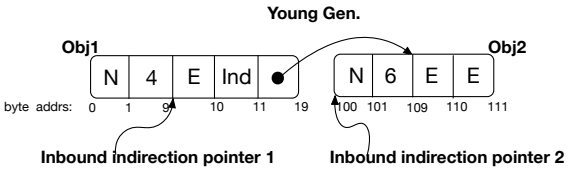
We adopt the following pretenuring policy: the first chunk of every region is allocated in the young generation, but *all subsequent chunks* directly start in the old-generation. Thus, after a small prefix, the remainder of a large structure would be written only once and *never* copied by GC, similar to Gibbon. The reasoning behind this policy is that the lifetime of a structure is at least the time required to construct it, which could be quite large for large structures. Analogous to the accepted wisdom that old objects tend to live even longer, large regions are more likely to grow even bigger. Also, in our experience typical Gibbon programs tend to not require many indirection pointers in their representation, thus we try to optimize the more common case. This design choice has a huge impact on benchmarks evaluated in Table 4.3.

4.1.3. Copying object graphs while maintaining sharing. Of course, an object can have more than one inbound pointer, and thus tracing GCs must maintain *sharing* as they relocate data—for example, a subtree might be shared by two separate tree values. The rootset itself can contain stack variables with multiple pointers into different parts of the same object, as is the case when the mutator recurs through multiple levels of a tree (illustrated in Figure 4.4a). Thus, if we failed to detect sharing while copying live data, all the local variables on the stack could end up with their own completely separate copy of the data[7]!

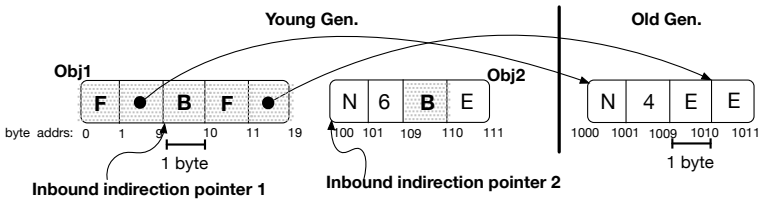
Alas, there are several challenges to maintaining sharing given a dense, mostly-serialized heap. First, there is insufficient space for forwarding pointers inside many objects' layouts. Second, when the collector is mid-way through copying an object, and finds a sub-portion of it *has* already been copied, it needs a shortcut pointer to "skip over" the already-copied portion and resume copying after it. Shortcut-pointers created by the mutator cannot be relied upon since they are added on a per-data-constructor, per-field basis. The simplest solution to both these challenges is to maintain two side-metadata tables during collection: (1) to store all forwarding information, i.e. a table that maps (start,end) intervals in the young-generation to their corresponding addresses in the old-generation, and (2) another table to store shortcut pointers

[6]The remembered set can never *shrink*, because pointers in the old generation cannot be deleted by the mutator as LoCal is a pure language.

[7]Strategies that introduce a *bounded* amount of data duplication, however, could still be considered as reasonable optimizations.



(A) A young-generation heap with two objects, namely `Obj1` and `Obj2`. `Obj1` has one inbound indirection pointer from an object not pictured here. `Obj2` has two inbound indirection pointers: from `Obj1` and from another object not pictured here. `N` is short for `Node` and `E` is short for `Empty` (a binary search tree constructor).



(B) The young-generation heap from (A) after `Obj1` has been evacuated. The collector has been able to add two forwarding pointers, but one object (`E`) has been burned without a forwarding pointer.

FIGURE 4.6. An in-progress evacuation that illustrates how objects are *forwarded* and *burned*.

(skip-over addresses). Unfortunately, using these tables naively is prohibitively expensive and makes the collector several times slower. Instead, we explore an approach that stores this metadata in the copied portion of objects where possible and only uses side-metadata tables as a fallback (the slow path), as we explain next.

4.1.3.1. Forwarding. The forwarding strategy we use follows the principle of: (1) precisely marking each byte that is copied, while (2) *opportunistically* including forwarding pointers anywhere in the bytestream where there is room: including wherever indirection pointers exist, and any data-constructors with more than a pointer-sized quantity of scalar data. We say that data marked in this manner is either *forwarded*, by writing a forwarding pointer into its payload, or, when too small for forwarding, *burned*, with each status corresponding to another reserved tag value. We denote these tags `B` and `F`. A "`B`" behaves like a single byte object, whereas a 9 byte "`F addr`" object consists of the forwarded tag followed by the new address of the object which previously lived at the same byte location as the forwarded tag in the nursery.

Before proceeding further, let us recap, and define some terminology:

- An *evacuation* copies a complete value from the nursery to the old-generation.

- In doing so, it *marks* all copied objects (with burned or forwarded tags).
- Only the first *chunk* within each *region* can reside within the nursery.
- An evacuation marks an *interval* of bytes within a nursery chunk. Intervals starting at location zero are treated differently[8].
- An interval consists of one or more indivisible *spans*, which are sequences of packed data that are free of indirections before collection begins. These spans are relocated to *layout-equivalent* spans of the same size in the old-generation.

When the collector needs to compute the forwarded address of a given tag, it either reads it directly (if the object was forwardable), or it reads a burned tag and *scans to the right* looking for a forwarding entry within the same span. Once such an entry is found, the address of the original tag byte in question can also be computed via subtraction—as its location relative to the forwarded object in the young-generation and old-generation will be conserved. This conservation holds because: (1) inlining of data due to the presence of indirection pointers is the only reason why an object could have different sizes in the young-generation and the old-generation, and (2) each indirection itself has enough space to store a forwarding pointer after being evacuated. Thus, the forwarding address of any object occurring in the *span* of bytes serialized *before* an indirection pointer can be computed in a straightforward manner: we use the forwarding pointer that will be written in place of the old indirection.

Consider a scenario where the young-generation heap is as illustrated in Figure 4.6b and the collector follows "inbound indirection pointer1" and reaches the burned tag in `Obj1` at byte-address `9`. It will now scan to the right and immediately discover a forwarding pointer at byte-address `10`. This forwarding pointer points to the old-generation byte-address `1010`. Thus, the collector will compute the forwarded address of the object at byte-address `9` as: `1010 - (10 - 9) = 1009`.

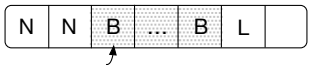
Our goal in designing the sharing-preservation aspect of our collection algorithm is thus to bound the amount of scanning time necessary to resolve a forwarded address for *any* tag residing in the young-generation before collection. Nevertheless, for completeness we need to also introduce a global table as the place of last resort to store forwarding information. The table maps (start,end) intervals in the young-generation to their corresponding start locations in the old-generation. The collector may enter in the middle of a burned interval, so the table needs to be an *interval map* allowing forwarding-address lookups keyed on locations anywhere within the forwarded interval.

[8]If we increased the number of regions in a Gibbon program until every object had its own region, then they would all be placed at location 0 and resemble object memory allocations in a traditional heap.

We fall back to the table when we fail to find forwarding information by scanning. We can fail by hitting the end of the span, which we recognize based on encountering an already-burned tag of the next span. The collector also exposes a *max_scan* parameter for the maximum number of burned bytes that should be traversed in search of a forwarding pointer. By default this is set to the nursery chunk size. After scanning this many bytes, we fall back to the global table to lookup forwarding information.

Symmetrically, with writing as with reading, after reaching the end of the span or burning *max_scan* bytes, without successfully forwarding, we populate an entry in the table. Because the bound must hold from any starting location, after successfully forwarding an object, we begin counting again, up to *max_scan*. If we hit an indirection, it is itself a forwardable object. In this case, we also mark the object downstream from it as start of a new span (in addition to resetting the *max_scan* counter).

4.1.3.2. Skipping-over. The worst-case scenario is pictured below: a tree value with *shape* only, containing no scalars, but whose subtree was already copied and burned. This value consists *exclusively* of small, non-forwardable objects.



Even scanning to the end of this interval will fail to turn up a forwarding pointer. Fortunately, the size of the problematic interval is bounded by the chunk size of the starting region (in the nursery). The reason is that, otherwise, the interval would contain a chunk redirection pointer to another chunk, which *itself* is forwardable. Unfortunately, the landlocked, evacuated value needs to not only have its new forwarding address resolved (for writing an indirection in the old-generation), but the collector *also* needs to know where the burned value *ends* in the from space, so that it can subsequently continue collection.

To this end, our algorithm, given in Figure 4.1, introduces a second table for storing skip-over addresses. The table provides a fallback, requiring a slow path for evacuation, just like the table for forwarding pointers. Our collector creates a table entry *only* when it enters a region for the first time at a *non-zero location*[9]. A zero-location tested at runtime with the `zero_location?` predicate, is simply the first location within each region. The reason is that values which begin at *location zero* in a region *never* need to be skipped over in this way. Either they are top-level values, or if they are referred to, it is via an indirection, which itself is trivial to skip over. Furthermore, such location-zero values are *always* forwardable, because we leave

[9]In future work, static analysis may assist in ruling out sharing and lessening this obligation.

enough room in the footer of each region chunk to store a forwarding pointer. Conversely, when the collector stumbles on a transition to burned data in the middle of a chunk, it immediately switches to the slow-path, performing a table lookup.

To support skipping ahead, the table stores only the end of the entire burned interval, i.e. the end of the value rooted in the first burned byte. This information is sufficient because the collector can only jump into the middle of the burned interval by following an indirection there. In that case, forwarding information is needed but skip-over information is not, because it is trivial to skip-over the value by skipping over the indirection itself. Adjacent burned intervals are not ambiguous with a single interval, because they correspond to two logical values, and as such are copied by separate evacuations resulting in separate entries in the table. In Figure 4.6b, our collector would create an entry in the skip-over table for the object at byte-address `109`, since this object lives at a non-zero location and is burned by following an indirection pointer. The collector will *look up* this entry in the table if it evacuates the object starting at byte-address 100.

In conclusion, this subsection introduced a sharing-preservation strategy, which, for completeness, includes separate tables for slow-path lookups of forwarding and skip-ahead information. We expect that programs take these slow paths rarely because: (1) heaps generally have a sufficient density of primitive data (ints, floats, strings) or indirections such that there is a high density of *forwardable* objects, and (2) skipping over already-evacuated values is necessary only when the nursery contains sharing. The latter case generally happens in programs that perform fine-grained allocations of small regions, with a high percentage of objects occupying location zero of their respective regions.

4.1.4. Write barrier. As mentioned in Section 4.1.2.2, because we allow old-to-young pointers, we need a write barrier on indirections written to the old generation. Thus on every write of an indirection, we test the target address to see if it's in the nursery, and if not we add it to a remembered set. Our write barrier currently compares the pointer against a $(start, end)$ range of addresses for the nursery. Further optimizations to the "is in nursery?" predicate are possible in the future[10].

One important optimization we do perform is to prevent redundant chains of indirections, **short circuiting** them, in the write-barrier. We perform an additional load to peek at the tag of the target to which the new indirection points, and if it is an indirection, we keep following it until we find a non-indirection

[10]For example, we could allocate the nursery in a fixed portion of the virtual address space, so that a test on the pointer is sufficient for the is-in-nursery test, without any additional loads for (dynamic) nursery bounds.

tag as the target. As with other aspects of the design, this leverages the immutability of our heap to maintain the invariant.

4.1.5. Major collection. A region in the old-generation can be freed either by having its reference count drop to zero, or by a backup major tracing collection. We discuss each strategy in the following.

4.1.5.1. Deferred reference counting. On the one hand, reference counting at the region-level is extremely effective for large values, because a single reference count is shared by many objects—such as an entire tree packed into a region. We thereby avoid tracing these large values, achieving the same benefits as GHC's *Compact Normal Forms* [**68**]. Young objects, on the other hand, are more *dispersed*, occupying more distinct regions-per-megabyte—for example when a linked list is allocated with each node in a different region. As always, the generational approach allows us to take advantage of the difference in expected lifetime between young and old objects. But there is a further advantage in the mostly-serialized setting: a copying minor collection *also* coalesces objects into fewer regions in the old-generation, placing those objects *in order* to take full advantage of the serialized representation, *eliminating* pointers from the representation rather than just moving and redirecting them.

4.1.5.2. Backup tracing collection. In the introduction, we described how grouping objects into regions can increase their lifetime. One live object in a region can prevent its reclamation, wasting space on other, dead objects inside the region. Further, because Gibbon's reference counting is region-level, keeping a region alive means keeping downstream regions alive, even if they are pointed to only by now-dead objects within the region. It is because of these fragmented regions and overapproximated reference counts that we need to include a backup collection strategy for the old-generation as well.

Our major collection uses the same copying strategy as minor collection—evacuating live objects into in-order serialized sequences inside fresh, dense regions. It is triggered in the standard way, when the old-generation size exceeds a threshold. What is unusual, however, is that the combination of region-reference-counting in the old-generation, with the backup collector, means that the old-generation size does not monotonically increase. It *shrinks* when reference counting deallocates regions. We perform major collection in a stop-the-world fashion, collecting all regions into a small number of output regions based on the number of roots.

Natural Incrementality. Because all the data in regions is immutable, it would be straightforward to incrementalize the major collection, which we could address in future work. That is, given the roots for a particular region, we can evacuate it, and compute an updated set of outbound pointers, either evacuating

them recursively, or using that winnowed *outset* to decrement the reference counts of downstream regions. For example, if we stop evacuating after copying K bytes, the (reduced) outbound pointers keep fewer downstream regions alive. Thus evacuating a region can free memory not just in that region, but other regions that are released.

4.2. Implementation Details

We implement our memory management system in the open-source Gibbon compiler[11]. Our implementation mainly changes Gibbon's runtime system (with only a few additions to certain LoCal-to-LoCal compiler passes). The region creation routine is updated to create a chunk in the young-generation using bump pointer allocation. A collection is triggered when the young-generation becomes full. Our garbage collector is implemented in the Rust programming language, primarily because of the memory safety guarantees it can provide, along with access to a rich collection of data structures in the standard library. The Rust code is compiled as a dynamic system library (using `crate-type =["cdylib"]`) and then linked with the GC-Gibbon-generated C code. This choice does have a side effect: we lose potential compile-time or link-time optimization opportunities between the C and Rust code[12]. We limit the interaction between our C and Rust code to just one function call, `garbage_collect`, which reduces any potential slowdowns caused due to missed optimizations.

Info-table. We use a statically allocated info-table to store the layout information required to evacuate objects of different types. This table is populated by the program when it starts executing. For each user-defined datatype in the Gibbon source program, the info-table has an entry of type `DatatypeInfo` given in Figure 4.7. The main evacuation loop operates like an interpreter consuming a stream of byte-codes; when the object being evacuated starts with a tag corresponding to a regular data constructor, (as opposed to a reserved tag described earlier) it retrieves the necessary layout information from the info-table.

Pointer encoding. At various points during collection, the collector needs to know metadata information of a region which houses an object that is the target of a pointer (indirection pointer or end-of-chunk pointer). If writing an old-to-old indirection pointer, the target region's reference count needs to change. If promoting a chunk that ends with a link to a pretenured chunk, the target region's set of chunks needs

[11] https://github.com/iu-parfunc/gibbon/

[12] Since we use GCC to compile the generated C programs because it usually produces more efficient code than Clang in our experience, especially for the switch-heavy tree traversal programs. Besides, getting meaningful link-time optimizations between code compiled using Clang and Rust is not trivial.

```
type InfoTable = Vec<DatatypeInfo>;

enum DatatypeInfo {
    Scalar(usize),
    Packed(Vec<DataconInfo>),
}

struct DataconInfo {
    scalar_bytes: usize,  // Bytes before the first packed field.
    num_shortcut: usize,  // Number of shortcut pointer fields.
    field_tys: Vec<u32>,  // Field types of packed fields.
}
```

FIGURE 4.7. The representation of GC-Gibbon's info-table in the Rust runtime system.

to be updated. This metadata information can be accessed via the footer of the target chunk. To enable access to the footer, we use a 64 bit pointer to store both: (1) the address of the target object, and (2) the offset from there to the target chunk's footer. The 16 high-order reserved bits are used to store the offset information. As a consequence, the maximum chunk size that can be allocated is bound at 65K bytes (2^{16}).

Reordering tag allocations. Even though allocations in LoCal happen in order, its formalism requires that a data constructor tag be written *after* all its fields are. That's because writing the tag indicates that a particular LoCal value is fully written, and the type-system ensures certain invariants are maintained by the fields. This, however, creates a problem for our collector which might need to copy a value while it is still under-construction(Section 4.1.2). Without the tag present at the beginning of an object, the collector cannot infer what kind of an object it is copying. We bypass this by reordering the writes such that a data constructor tag always gets written before any of its fields. Not only does this help the collector, it also makes the mutator slightly more efficient.

Bounding region size. A performance anti-pattern with previous versions of Gibbon was to allocate a sizeable region of the default size, typically at least 1K, and then write only a single constant-sized object to it, such as one cell of a linked list. This wastes a lot of memory, and in GC-Gibbon, can also cause many more collections to occur. It is therefore profitable to identify certain regions with *statically bounded maximum size*. We add such a static analysis on Gibbon's LoCal intermediate language. When the compiler

backend generates code for `letregion` constructs, it overrides the default size with the static bound if it is smaller. Implementing this requires analyzing all the locations that allocate to a particular region, and then inferring the sizes of objects written to these locations. The size of primitive types such as ints and floats is known a priori. Expressions that allocate a variably-sized serialized value (for example, using recursion) are inferred to have an *unbounded* size.

4.3. Evaluation

In this section we evaluate our memory management system using a variety of benchmarks taken from previous literature, and two additional benchmarks—`reverse` and `treeUpdate`—that stress the worse-case scenarios for our implementation. Besides prior Gibbon (referred to as Legacy-Gibbon in the rest of this section), we compare the performance of our implementation to GHC[13], which is especially optimized to run functional programs which allocate lots of small objects, and Java, which has a highly optimized and mature garbage collector. For our experiments, we use a single-socket Intel E5-2699 18 core machine with 64GB of memory and running Ubuntu 18.04. We compile the C programs generated by our implementation using GCC 7.5.0 with all optimizations enabled (option `-O3`). For comparing against Legacy-Gibbon, we use its version `0.2` compiled from source. To ensure an apples-to-apples comparison, we port our bounding-region-size optimization (Section 4.2) to Legacy-Gibbon. For GHC, we use GHC 9.0.2, with options `-threaded -O2`. We use GHC's default collector [**39**] and control the size of its young-generation with the run-time option `+RTS -A <SIZE> -RTS`. For Java, we use OpenJDK 17.0.1 with its default G1 collector [**21**] and control the size of its young-generation with the option `-XX:NewSize=<SIZE>`. Each reported measurement is the mean of 10 runs, where each run records the wall-clock time required to run a benchmark. For Java, we do two additional runs to warm up the JVM but don't count their run time when computing the mean. We oberved low variance in all our measurements and therefore do not report it separately.

Benchmarks. We consider the following benchmarks for our evaluation. For GHC, we use *strict datatypes* in benchmarks, which generally offers the same or better performance, but avoids performance complications due to laziness. All programs use the same algorithms[14] and datatypes, and are run with the same inputs. For GHC and Legacy-Gibbon, we hold the size of the young-generation constant at 4M. For Java, the young-generation starts with a size of 4M, but is allowed to grow if desired by the collector.

[13] https://www.haskell.org/ghc/

[14] To workaround a stackoverflow error, we use `for` loops instead of recursion for the Java implementation of `reverse` and `treeUpdate`.

- **reverse**: This is the standard accumulator style list-reverse program shown in Figure 2.7a; it reverses a list containing 8M integers. The Java implementation is defined using a `while` loop rather than a recursive function.
- **treeUpdate**: This is the complete version of the program given in Figure 2.8a. It starts with a very small search tree and repeatedly inserts and deletes numbers in it. The numbers are chosen from small range (0-512) to keep the size of the tree more or less constant, and the tree is updated 5M times. The Java implementation encodes the outer "update" loop as an actual `while` loop, but `insert` and `delete` are defined using recursion in the standard way.
- **coins**: This benchmark is taken from GHC's NoFib[15] benchmark suite. It computes the number of ways in which a certain amount of money can be paid by using the given set of coins. The input set of coins and their quantities are `[(250,55),(100,88),(25,88),(10,99), (5,122),(1,177)]`, and the amount to be paid is 999.
- **lcss**: This benchmark computes the longest-common-substring using Hirschberg's algorithm. Our implementation is taken from GHC's NoFib benchmark suite. We provide as input two strings of length 3100 and 3000, respectively, such that the result has length 2100.
- **power**: This benchmark [**42**] computes 20 elements of the power series `(ts = 1 :+: ts^2)`, which is shown here assuming lazy evaluation. We use a slightly modified implementation that is suitable for a strict language.
- **buildKdTree** and **countCorr** and **allNearest**: `buildKDTree` constructs a kd-tree [**25**] containing 1M 3-d points in the Plummer distribution. `countCorr` takes as input a kd-tree and counts the number of correlated (within a distance of 100 units) points for all 1M 3-d points. `allNearest` computes the nearest neighbors of all 1M 3-d points.
- **barnesHut**: Uses a quad tree to run an nbody simulation over 1M 2-d point-masses distributed uniformly within a square.
- **constFold**: This benchmark is taken from [**34**] and implements constant folding for a language that supports integer arithmetic. It is run on synthetic syntax-tree which is a balanced binary tree of depth 26.
- **evacSharedTree**: This is a synthetic benchmark designed to stress the sharing-preservation aspect of our collection algorithm. It allocates a large balanced binary tree with maximum sharing and releases it almost immediately, and then triggers a major collection. Only the time required for

[15] https://gitlab.haskell.org/ghc/nofib

TABLE 4.1. Run times in seconds of benchmarks run with different GC configurations (explained in Section 4.3).

Benchmark	Default	NoBurn	NoCompact	Simple-Barrier	NoBurn +SB	NoCompact +SB
reverse	0.49	0.43	11.8	0.49	0.44	11.9
treeUpdate	0.77	0.75	0.88	2.30	1.20	12.1
coins	4.34	4.32	4.31	10.45	10.3	10.4
lcss	0.51	0.53	0.54	0.52	0.52	0.53
power	1.40	1.34	1.36	1.38	1.40	1.44
evacSharedTree	2.53e-05	2.44	1.33e-04	3.25e-05	2.43	1.31e-04

the major collection is measured, to demonstrate the performance impact of sharing-preservation. Maximum sharing in the allocated tree is achieved as follows: for each interior node at height `h`, a single tree with height `h-1` is constructed and is used as the left *and* the right subtree of this node. In GC-Gibbon and Legacy-Gibbon, this tree with height `h-1` will be allocated in a separate region and two indirections will be written to construct the node with height `h`. This benchmark is run on tree of height 25.

These benchmarks are roughly divided into two sets: (1) those that perform many out-of-order and small-allocations (`reverse`, `treeUpdate`, `coins`, `lcss`, `power`), where the mostly-serialized approach is weakest, and (2) those that allocate or traverse a large data structure (`buildKdTree`, `countCorr`, `allNearest`, `barnesHut`, `constFold`), where the mostly-serialized approach shines.

Evaluating GC design choices. To evaluate the effects of the design choices we made, we run the benchmarks that stress the collector in six different modes, each of which toggles a specific choice:

- **Default:** The default configuration follows the design described in Section 4.1, with all optimizations enabled.
- **NoBurn:** In this mode we disable the forwarding pointer mechanism described in Section 4.1.3. Thus, every shared value is copied multiple times. A benefit of this is that writing forwarding pointers and burning data, and maintaing side-metadata tables is no longer required.

- **NoCompact:** In this mode we disable compaction (pointer elimination). For each indirection pointer encountered during evacuation, if the target object is not already copied, it is put into a fresh region and a new indirection pointing to this fresh region is created.
- **SimpleBarrier:** In this mode we disable the optimization that eliminates redundant chains of indirections (Section 4.1.4). This makes the write-barrier more efficient by reducing the number of memory loads it performs, but makes the collection more expensive because of the overheads associated with evacuating indirection-heavy heaps (due to forwarding, maintaining side-metadata tables, etc.).
- **NoBurn+SB**: This mode is a combination of `NoBurn` and `SimpleBarrier`. We disable forwarding pointers *and* the indirection-chain-elimination optimization.
- **NoCompact+SB**: This mode is a combination of `NoCompact` and `SimpleBarrier`. We disable compaction (pointer elimination) *and* the indirection-chain-elimination optimization.

All of these modes (except `SimpleBarrier`) are configuration flags provided to the collector. `SimpleBarrier` requires recompiling the mutator since the write-barrier is inlined into the mutator at compile time.

The results are given in Table 4.1. The choices made by our collector (column `Default`) perform well across all benchmarks. For all other modes, there is at least one benchmark which performs poorly. `evacSharedTree` has maximum sharing and is therefore a pathological worst-case for the `NoBurn` mode, which disables sharing-preservation during copying. In this case the collector copies an exponential amount of data—335MB versus 14KB in `Default` mode! Even though this is a worst-case scenario, it's quite easy to run into this sort of copying behavior with sharing disabled in the collector. With respect to `NoCompact` mode, `reverse` is 24× times slower in this configuration since it has the highest number of indirections among these benchmarks—8M, one for each cons-cell. The effectiveness of the indirection-chain-eliminating write-barrier is demonstrated by `treeUpdate` and `coins`, both of which create long indirection chains and are 2-3× slower in this mode. The performance of `treeUpdate` in `NoBurn+SB` mode is peculiar. On one hand, the collector handles a large number of indirections, but because sharing is disabled, it saves time by not having to maintain side-metadata tables. For instance, in `SimpleBarrier` mode the skip-over table contains 18K elements on average, versus 0 in `NoBurn+SB`. `treeUpdate` performs poorly (15× slower) in `NoCompact+SB` mode due to the overhead of side-metadata management and excessive old-generation allocations.

Comparison to other systems. Tables 4.2 and 4.3 show the results of comparing performance of our system to Legacy-Gibbon, GHC, and Java. For small and out-of-order allocation benchmarks (Table 4.2), GC-Gibbon benefits from its fast bump-allocated young-generation, whereas Legacy-Gibbon shows the

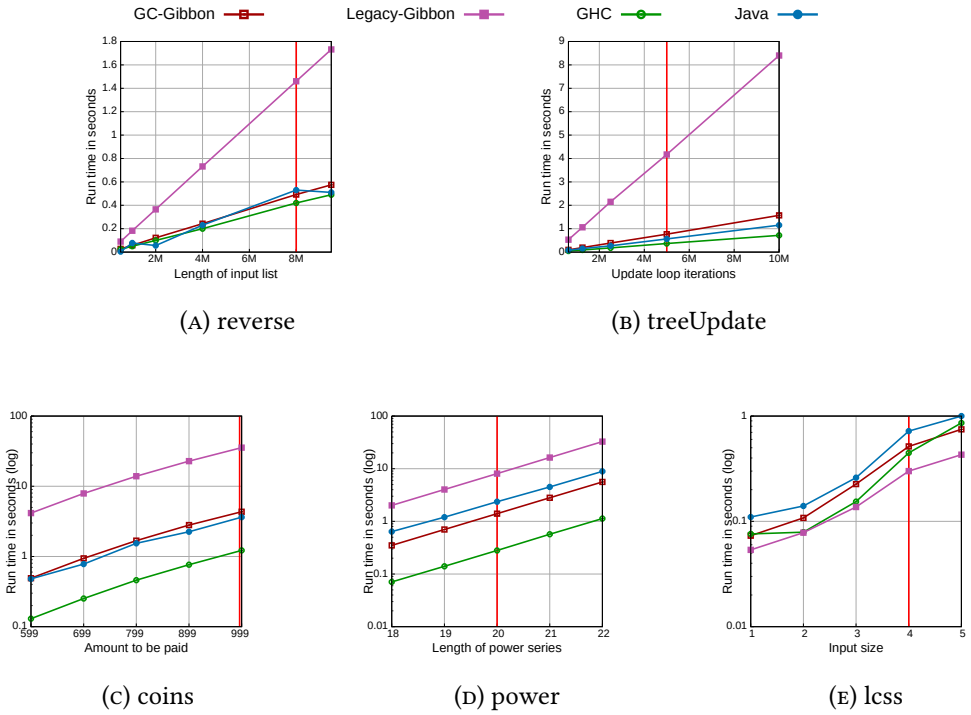
TABLE 4.2. Run times of out-of-order and small-allocation benchmarks in seconds.

	GC-Gibbon	Legacy-Gibbon		GHC		Java	
Benchmark	T_{gcgib}	T_{oldgib}	$T_{\text{oldgib}}/T_{\text{gcgib}}$	T_{ghc}	$T_{\text{ghc}}/T_{\text{gcgib}}$	T_{java}	$T_{\text{java}}/T_{\text{gcgib}}$
reverse	0.49	1.46	2.98	0.42	0.86	0.53	1.08
treeUpdate	0.77	4.17	5.41	0.37	0.48	0.56	0.73
coins	4.34	35.5	8.18	1.21	0.28	3.63	0.84
lcss	0.51	0.30	0.59	0.45	0.88	0.72	1.41
power	1.40	8.07	5.76	0.28	0.20	2.36	1.68
geomean	-	-	3.39×	-	0.46×	-	1.09×

TABLE 4.3. Run times of in-order allocation and bulk-traversal benchmarks in seconds.

	GC-Gibbon	Legacy-Gibbon		GHC		Java	
Benchmark	T_{gcgib}	T_{oldgib}	$T_{\text{oldgib}}/T_{\text{gcgib}}$	T_{ghc}	$T_{\text{ghc}}/T_{\text{gcgib}}$	T_{java}	$T_{\text{java}}/T_{\text{gcgib}}$
buildKdTree	2.67	2.53	0.95	7.78	2.91	4.48	1.68
countCorr	1.77	1.77	1.00	3.00	1.7	4.47	2.52
allNearest	0.71	0.80	1.13	1.46	2.06	1.00	1.41
barnesHut	3.54	3.40	0.96	5.83	1.65	2.40	0.68
constFold	1.38	1.50	1.09	4.12	2.98	2.56	1.85
geomean	-	-	1.02×	-	2.19×	-	1.5×

overheads of `malloc`-based region allocations. In the case of `reverse`, both Gibbon versions need to allocate a new region per input element, thus, 8M regions are allocated in this instance. But despite this very high rate region allocation, GC-Gibbon is 7% faster than Java and only 14% slower than GHC. The `lcss` benchmark is surprisingly fast with Legacy-Gibbon. According to our initial observations, `lcss`' allocation pattern seems to naturally have a stack-like behavior and thus benefits from Legacy-Gibbon's region based memory management. While we have achieved significant performance improvements compared to Legacy-Gibbon, some benchmarks do not perform as well as we might hope. Without the years of optimizations in these competing systems, since our collector is new there is much room improvement.



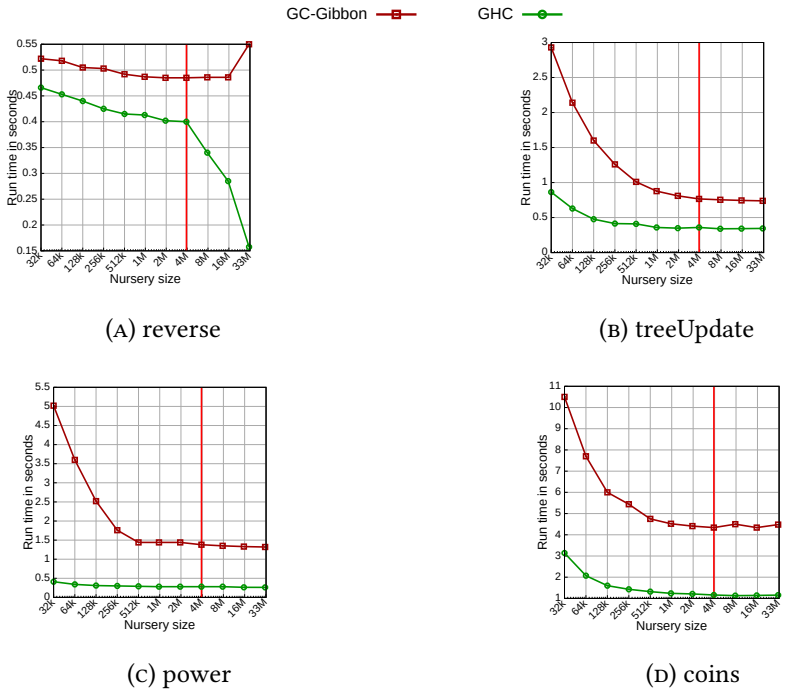
(A) reverse (B) treeUpdate

(C) coins (D) power (E) lcss

FIGURE 4.8. Run times of benchmarks using inputs of various sizes, the young-generation size is held constant at 4M. The red line marks the input used for measurements reported in Table 4.2.

Legacy-Gibbon has a home-turf advantage on the in-order allocation and bulk-traversal benchmarks (Table 4.3). As the results show, GC-Gibbon does not degrade their performance. The pretenuring optimization described in Section 4.1.2 is key to this. The slowdowns observed here are primarily because GC-Gibbon's pointer encoding mechanism, which puts an upper bound on the largest chunk that it can allocate, namely 65K, unlike Legacy-Gibbon which sets this upper bound to 1GB. Both GC-Gibbon and Legacy-Gibbon, however, outperforms GHC and Java on these benchmarks. Java performs exceptionally well on `barnesHut`.

For small allocations, our system is 3.79×, 0.46×, and 1.09× geomean faster than Legacy-Gibbon, GHC, and Java, respectively. For bulk tree-traversals, our geomean speedup is 1.02×, 2.19× and 1.5×. Overall, these results show that GC-Gibbon offers significant performance improvements compared to Legacy-Gibbon on small and out-of-order allocation benchmarks, *without* degrading the performance on bulk-traversal and allocation benchmarks. Legacy-Gibbon is prohibitively slow on certain workloads, thereby discounting its use entirely if any part of a big application has an allocation pattern like `coins`, `power` or



(A) reverse

(B) treeUpdate

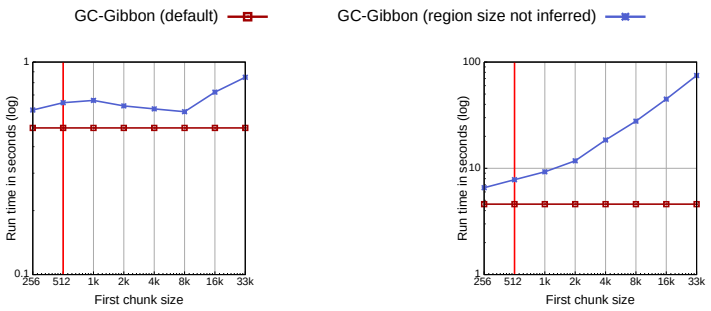
(C) power

(D) coins

FIGURE 4.9. Run times of benchmarks using young-generations of various sizes. The red line marks the nursery size used for measurements reported in Table 4.2, 4M.

`treeUpdate` as an example. For these benchmarks, GHC is 29×, 28× and 11× faster than Legacy-Gibbon, respectively. Overall, GC-Gibbon shows that general-purpose language runtimes powered by (mostly) serialized data representations can support efficient tree traversals, as in Legacy-Gibbon, and perform well in the setting of general-purpose functional programming, with a comprehensive solution for garbage collection.

Parameter sweeps. In this section we discuss the results of three parameter sweeps for small out-of-order allocation benchmarks that stress the collector. Figure 4.8 shows the run times of benchmarks using inputs of various sizes, and the young-generation size is held constant at 4M. All the variants have similar behavior, with Legacy-Gibbon being the slowest in most cases. Figure 4.9 shows the run times of benchmarks using different young-generation sizes. This shows the expected tradeoff between space and time; the wall-clock time gets better as the young-generation gets bigger, due to fewer collections. Figure 4.10 shows the run times of benchmarks using different initial chunk sizes for GC-Gibbon. Using bigger initial chunks, and therefore growing the overall region at a faster rate, causes more collections



(A) reverse

(B) coins

(C) power

(D) treeUpdate

FIGURE 4.10. Run times of benchmarks using initial chunks of varying sizes. Our region size inference analysis (Section 4.2) causes GC-Gibbon to allocate constant-sized regions in the main workoad of `reverse` and `coins`, thus their run times are constant. We report the measurements after disabling this optimization as "GC-Gibbon (region size not inferred)".

to occur since the nursery fills up faster. It also leads to inefficient space usage since many of the larger chunks will be mostly empty.

Accelerating Haskell tree-traversals: from Gibbon to GHC

Presently, Gibbon is a standalone whole-program compiler for a small, strict subset of Haskell. But this is not how we envision using it in the long term. We would instead like Gibbon to be an *accelerator* that can be used *with* GHC. That is, Gibbon and its serialized data representations could be used to optimize some portion of a bigger application that is compiled and run using GHC. This approach has two primary benefits:

- It would give Gibbon programmers access to the wider Haskell library ecosystem and many Haskell features that Gibbon does not yet support in its front-end source language. This would enable writing a wide variety of programs that are currently not possible.
- It would significantly lower the barrier to Gibbon's adoption and make it a practical tool that could be used in many different scenarios.

To accomplish this goal, we implemented a proof-of-concept library named `gibbon-plugin` that can be included in a Haskell application in the standard way[1] and that can be used in a regular Haskell program compiled with GHC. With `gibbon-plugin`, users can "mark" certain Haskell expressions to be compiled by Gibbon, and most importantly, these Gibbon-compiled expressions can be interspersed and used in the overall program easily.

`gibbon-plugin` uses GHC's plugin mechanism to integrate Gibbon with GHC. A GHC plugin allows one to edit GHC's usual compilation pipeline. Among other things, it lets one inspect and modify a program being compiled. `gibbon-plugin` defines a plugin that works as follows: (1) it walks over the Haskell module being compiled and collects expressions that are marked to be compiled using Gibbon, (2) it then uses the existing Gibbon compiler to generate a new versions of these expressions that will operate on serialized representations, and (3) sets things up so that the broader Haskell program can use these new Gibbon-compiled expressions. Programmers need to pass a compile time flag to GHC, `-fplugin=GibbonPlugin`, to use this plugin. While this overall strategy is straightforward, there are several details to get right so that the resulting system is easy to use and performs well:

[1] Using Haskell package managers like Cabal or Stack.

- Gibbon is a whole-program compiler whereas GHC is not. How to bridge this gap?

  We *generate* a whole program for each expression that is marked for Gibbon-compilation. Essentially, we collect all functions transitively reachable from the marked expression, crossing module and library boundaries if required, and compile this *transitive closure* as a "whole-program" using Gibbon.
- `gibbon-plugin`'s operates on GHC's intermediate language called Core [**22**], which is a variant of System FC [**58**]. Can we translate Core programs into Gibbon's frontend language?

  We can translate most Core programs into Gibbon's frontend language. Certain Core expressions, however, have no equivalent Gibbon counterpart—for example, coercions. Our implementation aborts compilation upon encountering such an expression. We plan to find a graceful way to handle this situation in the future.
- At the boundary between Haskell-compiled code and Gibbon-compiled code, we will need to convert values between the Haskell representation and the Gibbon representation. How would this work and can we accomplish this easily for values of new user-defined datatypes?

  We essentially need to serialize and deserialize values to convert them to and from the Gibbon representation. Our approach here is very similar to the existing `binary`[2] library. Similar to this library, we can use `GHC.Generics` to automate this process for user-defined datatypes (keep programmers from writing boilerplate code), but our implementation doesn't do this yet.
- How would memory management work in this setting? Can we teach GHC's collector about serialized values, or, do we have to keep Gibbon's collector around and make the two collectors cooperate?

  Memory management is a crucial component for making this system complete, but our implementation doesn't handle it yet. But we think that we'll need to keep Gibbon's collector around, given the complexity of collecting serialized values as shown in Chapter 4, and devise a cooperative collection strategy.

Next, there are some design choices to be made:

- At what granularity should programmers mark things for Gibbon-compilation: at the level of a complete Haskell module or top-level expressions or *any* expression?

  `gibbon-plugin` allows programmers can mark any expression for Gibbon-compilation.

[2]Binary serialisation for Haskell values using lazy ByteStrings: https://hackage.haskell.org/package/binary

- Should we use the existing Gibbon backend that generates C code or should we develop a new backend that can generate Core code? This question boils down to whether we want Gibbon primitives such as "allocate a region", "write a number", "read a number" etc. to be implemented in C or in Haskell.

  We use Gibbon's existing C backend. Our initial experiments showed that Gibbon primitives implemented in Haskell are slower compared to C. Thus, we generate C code for each expression marked for Gibbon-compilation, compile this C code to object code (`.o`) using GCC, link this object code with GHC compiled code, and use functions in this object code using Haskell's Foreign Function Interface (FFI).

5.1. Design and Implementation

In the remainder of this section, we describe `gibbon-plugin`'s components in turn.

5.1.1. API and performance evaluation. The API we designed for using `gibbon-plugin` in a Haskell program is given in Figure 5.1. A value of type `a` when converted to use Gibbon's serialized data representation has the type `Packed a`. The typeclass `Packable` provides functions that can convert a value from the Haskell representation to the Gibbon representation (`toPacked`) and vice versa (`fromPacked`). Programmers need to define instances of this typeclass for datatypes that cross the boundary between Haskell-compiled code and Gibbon-compiled code. `gibbon-plugin` can help here by providing a `Generic` instance for `Packable`, but it doesn't do this yet. Next, `liftPacked` can be used in a program to *mark* expressions that should be compiled using Gibbon. `liftPacked` takes an expression of type `(a → b)` and returns an expression of type `(Packed a → Packed b)`. Thus, it returns an optimized version of the expression which will operate on serialized values.

A simple program that uses this API is given in Figure 5.3. Here we're using Gibbon to optimize the performance of the expression named `buildsum` which is defined to be `(λn → sumTree (buildTree n))`. We define `fastbuildsum` as `(liftPacked buildsum)`. In this case, `gibbon-plugin` will essentially generate a new version of the function `buildsum`, and correspondingly `buildTree` and `sumTree` as well, and then update the definition of `fastbuildsum` to use this new version of `buildsum` via the C FFI. We will explain how this works next. Before that we present some preliminary performance evaluation results given in Figure 5.2.

Column "GHC" shows the run time of a benchmark compiled with GHC using its standard pointer-based representation. Column "GHC+Gibbon" shows the run time of a benchmark compiled with GHC and optimized using `gibbon-plugin`'s `liftPacked`. Column "Standalone Gibbon" shows the run time of a

```haskell
-- Packed (serialized) representation of the value of type 'a'.
data Packed a

-- A typeclass to convert values between the Haskell representation and the Gibbon representation.
class Packable a where
  toPacked   :: a → Packed a
  fromPacked :: Packed a → a

-- Lift an expression to operate on packed representation.
liftPacked :: (a → b) → (Packed a → Packed b)
```

FIGURE 5.1. `gibbon-plugin`'s API to use Gibbon as an accelerator in a Haskell program.

Benchmark	GHC	GHC+Gibbon	Standalone Gibbon
buildsum	1.27	0.85	0.71

FIGURE 5.2. Run time of benchmarks in seconds.

benchmark compiled with standalone Gibbon compiler. `buildsum` compiled using GHC+Gibbon is 1.5× faster compared to the same program compiled using just GHC and has comparable run time to Standalone Gibbon. While we evaluated `gibbon-plugin` using just one benchmark, it still shows the promise of this approach. We plan to work on expanding the performance evaluation in the future.

5.1.2. Fetching the transitive closure of an expression. There is a mismatch between Gibbon's and GHC's compilation methodology: Gibbon is a whole-program compiler, whereas GHC compiles one module at a time. Thus, if the marked expression (the expression given as argument to `liftPacked`) uses a function from a different module, Gibbon needs access to its definition but GHC does not. In GHC, a function's definition is available across modules only if it is *inlineable*—that is, if it is marked with an `INLINE` or an `INLINEABLE` pragma or if GHC decides to inline it on its own. To work around this, we require programmers to compile their application and all its dependencies using the flag `-fexpose-all-unfoldings`. This can be accomplished by specifying the following in a `cabal.project` file:

```
package *
  ghc-options: -fexpose-all-unfoldings
```

```
module BuildSum where

-- Import the Gibbon library.
import Gibbon ( liftPacked, Packed )

data Tree = Leaf Int | Node Tree Tree

buildTree :: Int → Tree
buildTree 0 = Leaf 1
buildTree n = Node (buildTree (n-1)) (buildTree (n-1))

sumTree :: Tree → Int
sumTree (Leaf n)   = n
sumTree (Node l r) = (sumTree l) + (sumTree r)

buildsum :: Int → Int
buildsum n = sumTree (buildTree n)

fastbuildsum :: Packed Int → Packed Int
fastbuildsum = liftPacked buildsum

main = do
  time1 ← measureRunTime buildsum 25
  time2 ← measureRunTime fastbuildsum (toPacked 25)
  print (time1, time2)
```

FIGURE 5.3. A simple Haskell program that uses Gibbon as an accelerator via `liftPacked`.

This ensures that all function definitions are accessible across module and library boundaries. There is one caveat. This flag does not apply to GHC's standard library, `base`, because it'll essentially require recompiling `base` which is not straightforward. Because of this limitation, if a marked expression uses a function from `base` that is not inlineable, our current implementation will fail to process it. In the future, we can provide a new version of `base` that is compiled with the required flag.

To satisfy Gibbon's whole-program constraint, we collect all functions transitively reachable from the marked expression, crossing module and library boundaries if required, and compile this *transitive closure* as a "whole-program" using Gibbon. Collecting the transitive closure of an expression can be accomplished by using the usual GHC API (functions from the `ghc` library) that is available for use in a GHC plugin.

5.1.3. Translating Core programs to Gibbon's frontend language. Gibbon's frontend language, L0, is a pure higher-order functional language with support for parametric polymorphism, algebraic data types, and pattern matching. It is very similar to Core except the following. Core has both recursive and non-recursive let bindings, whereas L0 only has non-recursive let bindings. Next, Core and its pattern matching implementation support a default *wildcard* pattern that matches any value, but L0 does not have this feature. These differences can be overcome by various program transformations.

Some differences, however, between Core and L0 are incompatible. Haskell, and therefore Core, has a much more advanced type system than L0 and some of these type system features also bleed into the expression language. For example, Core has some expressions that L0 has no match for, such as type applications, casts, and coercions. If `gibbon-plugin` encounters any of these expressions, it aborts compilation all together. In the future, we plan to find a graceful way to handle these.

While this is clearly a limitation of `gibbon-plugin`, it might not be a major roadblock to its adoption in practice. In our experience, high performance Haskell code often does not use advanced type system features and this is the kind of code that Gibbon will likely be used to optimize. Moreover, since programmers can mark expressions using `liftPacked` in a piecemeal fashion, `gibbon-plugin`'s limitations will only apply to a (possibly small) portion of the overall Haskell application. The other portion of the application (which is not reachable via the marked expression) is free to use any and all Haskell features that are required.

5.1.4. Tying everything together using FFI. For the program given in Figure 5.3, `gibbon-plugin` compiles `buildsum`, `buildTree`, and `sumTree` using Gibbon, specifically it's C backend. As a result, we get an object file (`.o`) which exports the following functions:

```
$ nm buildsum.o
c_buildTree
c_sumTree
c_buildsum
```

These functions operate on serialized representations of values, namely a serialized tree in this case. To use these functions in the Haskell program, `gibbon-plugin` will generate Core code equivalent to the following

Haskell code:

```
foreign import ccall unsafe "c_buildsum"
  c_fastbuildsum :: CInt → CInt

fastbuildsum :: Int → Int
fastbuildsum = fromPacked . c_fastbuildsum . toPacked
```

First, it has added an FFI binding for the `c_buildsum` function. The type of the function is `CInt → CInt` since `CInt` is the Haskell type representing the C `int` type. Second, the definition of `fastbuildsum` has been updated to use this new FFI binding. This is a fairly mechanical translation, the only interesting part here is the use of `fromPacked` and `toPacked`. Any types that the cross the boundary between the Haskell-compiled code and Gibbon-compiled code have to be encoded and decoded into an appropriate representation.

Converting values between representations will usually be slow (except for scalars such as numbers) and should be avoided as much as possible. It's more efficient to use `liftPacked` to lift a big computation, rather than lifting small pieces of work. In the latter case, the overhead of converting values between representations might outweigh the benefits of using serialized representations. Much like parallelism, the granularity of `liftPacked` must be controlled. Also, our current implementation doesn't offer a way for the Gibbon-compiled code to call back into Haskell-compiled code, the FFI bindings that are generated only go one way. This is something we plan to investigate in the future.

5.2. Future Work

We have only taken the first step towards developing an efficient way of using Gibbon as an accelerator in a Haskell program. There is ample opportunity for further investigation in this area. In particular, the following parts of `gibbon-plugin` could be significantly improved:

- We need to provide an easy way to derive `Packable` for user-defined datatypes without having to write a lot of boilerplate code. This is something that can be accomplished with the help of `GHC.Generics` and we plan to work on it.
- Memory management isn't handled in the current implementation at all. This is important for it to be a complete solution.
- There's much work to be done to make the plugin more robust, the current implementation is still a proof-of-concept.

Related Work

The most closely related work to this dissertation is, of course, Vollmer et al.'s Gibbon compiler [**66**] and LoCal [**65**], which was summarized in Chapter 2. As discussed there, Vollmer et al.'s treatment only provided sequential semantics and a simple reference counting based memory management system which is only suitable for processing bulk tree-like data. First, we extend those semantics to incorporate parallelism and present *Parallel Gibbon*, an implementation of the new parallel semantics whose performance exceeds the performance of the best existing parallel functional compilers. Second, we present *GC-Gibbon* that uses a much more sohisticated memory management scheme based on a generational garbage collector, which can handle a variety of workloads efficiently. It retains Gibbon's strong performance on tree-traversals where the serialized data approach is most effective *and* also achieves reasonable performance on out-of-order small allocations where the approach is weakest, more closely resembling mature compilers and runtime systems that have been heavily optimized for such programs, using traditional memory representations. In the following, we outline other prior research on serialized data structures, parallelism, and garbage collection, and how Parallel Gibbon and GC-Gibbon relate to them.

6.1. Data Processing and Layout Control

Earlier precursors to Gibbon include [**47**], which observed the relationship between an (ordered) extension of linear type theory, and the ability to stipulate data layout. But this work considered only fixed-sized types, not recursive types of unbounded size. It inspired later work, [**33**], which made the connection between a type system that could enforce data is consumed in left-to-right order, and the ability to convert tree-traversal programs into stream-processing programs (which is what Gibbon does). Gibbon is structured differently, taking unannotated programs as input and having to accept any program irrespective of data access order (instead changing the control flow and data representation both as free variables to create an efficient combination).

6.2. Serialized Data and Parallelism

Parallel Gibbon is related to several HPC approaches to serializing recursive trees into flat buffers for efficient traversal [**26, 44, 38**]. Notably, these approaches *must* maintain the ability to access the serialized trees in parallel, despite the elimination of pointers internal to the data structure, or risk sacrificing their performance goals. The key distinction that makes enabling parallelism in the HPC setting "easier" than in our setting is that these approaches are application-specific. The serialized layouts are tuned for trees whose structure and size are known prior to serialization, and the applications that consume these trees are specially-written to deal with the application-specific serialization strategies. Hence, offsets are either manually included in the necessary locations, or are not necessary as tree sizes can be inferred from application-specific information.

Work on more general approaches for packing recursive structures into buffers includes Cap'N Proto [**64**] and FlatBuffers [**27**], which attempt to unify on-disk and in-memory representations of data structures, and Compact Normal Form (CNF) [**68**]. Cap'N Proto, is designed to eliminate encoding/decoding by standardizing on a new binary format for use in memory as well on disk/network. Compact Normal Forms (CNF) is a feature provided by the Glasgow Haskell Compiler since version 8.2. The idea is that any purely functional value, once fully evaluated, can be compacted into its own region of the heap by capturing a transitive closure of its reachable data. The invariant maintained here is that all pointers inside the CNF must only point to other objects within the CNF, and outside it. After compaction, this heap can be stored externally and loaded back into the heap later.

Neither of these approaches have the same design goals as LoCal and LoCal$^{\text{par}}$: Cap'N Proto, FlatBuffers, and CNF preserve internal pointers in their representations, eliding the problem of parallel access by invariably paying the cost (in memory consumption and lost spatial locality) of maintaining those pointers. We note that Vollmer et al. showed that LoCal's representations enable faster sequential traversal than two of those approaches [**65**], and Section 3.4 shows that our approach is comparable in *sequential* performance to LoCal despite also supporting parallelism.

There is a long line of work on flattening and nested data parallelism, where parallel computations over irregular structures are *flattened* to operate over dense structures [**13, 32, 8**]. These projects do not, however, have the same goals as ours. They focus on array data, generating parallel code, and data layouts that promote data parallel access to the elements of the structure, rather than selectively trading off between parallel access to structures and efficient sequential access.

6.3. Region-based Memory Management

The main motivation behind *region-based memory management*[**62**] was to bring some of the benefits of *stack-based memory management* (a technique common in imperative languages like Pascal and Algol) to higher-order functional languages (primarily Standard ML). In this context, "region types" are a feature of a type system that tracks what region of memory a value is allocated into, with the goal of safely de-allocating all values in that region once it goes out of scope.

Often closely associated with region types is region inference [**10, 59**], a technique for taking an ordinary call-by-value program and inferring the region bindings and annotations such that all memory is safely allocated and deallocated into regions (essentially using region types in a typed intermediate language). This was used in the ML-Kit compiler, allowing it to compile arbitrary SML programs without the need for a general-purpose garbage collector, but practical problems with region inference prevented it from catching on in mainstream language implementations [**60**].

One of the issues with region-based memory management is that some common patterns of functional programming end up causing memory leaks [**60**]. One such case is iteration via tail recursive functions. Each iteration will accumulate allocations in a region, even if the allocations from the previous iteration are no longer live and ideally should be discarded. In the worst case, a program may even end up accumulating data in a "global region" that spans the lifetime of the whole program, thus ensuring that the memory will not be de-allocated promptly. Various attempts at extensions or optimizations to region systems have been proposed to address this, such as *storage mode analysis*, where a compiler inserts special instructions to reset the allocation pointer in a region if it can prove the region contains no live values, essentially allowing its contents to be overwritten [**61**].

A related limitation of region-based memory management, and of region-based systems in general, is the requirement that regions have last-in-first-out (LIFO) lifetimes which follow the block structure of a language [**10**]. Essentially, as long as regions are introduced with `let region`, programs are limited to "static" regions, or regions that are introduced for the static scope of the `let` binding and eliminated after leaving that static scope. In other words, memory for the region `r` is allocated when control enters the `letregion r in e` syntactic form, `r` is live for the execution of `e`, then `r` is de-allocated when control leaves the `letregion` form.

6.4. Garbage Collection

In [**23**] Elsman et al. explored combining regions and garbage collection in the MLKit system. The combination proved less fruitful in this system because most of the memory was still reclaimed by the region mechanism instead of the generational collector. Moreover, MLKit primarily uses regions to get the benefits of stack-based memory management, but each object within a region is still traditional in every other way and all pointers among objects are retained. Thus, its collector does not face the challenges of copying serialized partially-written objects, and correspondingly, it also doesn't benefit from the resulting compaction.

The literature on traditional garbage collection not only includes much work on tracing and reference counting independently, but also in combination. Our proposed collector is similar to Ulterior reference counting [**11**] which also has a copying young-generation and a reference counting old-generation. This work was further extended in [**52**], which uses an efficient heap structure named Immix [**12**] and reference counting, and also includes a backup tracing collector. The recent LXR collector [**69**] further builds on this. It is also based on the Immix heap and uses a combination of tracing and reference counting. It brings together several optimizations and heuristics, and introduces an efficient remembered set and a low-overhead write barrier to make reference counting efficient, and is able to reclaim most memory without any copying. There is ample opportunity to improve our reference counting collector using these techniques.

Parallel GC. Efficient automatic memory management is a longstanding challenge for parallel functional languages. There is work proposing a split-heap collector that can handle a parallel lazy language [**41**] and a strict one [**56**], and there is work on a scalable, concurrent collector [**63**]. A promising new line of work explores scalable garbage collection by structuring the heap in a hierarchy of heaps, enabling task-private collections [**28**]. Recent work has extended the hierarchical heaps model for parallel functional programs [**48**] and a parallel garbage collector [**6**]. The idea is to coordinate memory allocation and garbage collection with thread scheduling decisions so that each processor can allocate memory without synchronization and independently collect a portion of memory by consulting a collection policy that is fully distributed and does not require communicating with other processors. All of these designs focus on a conventional object model for algebraic datatypes that, unlike LoCal$^{\text{par}}$, assume a uniform, boxed representation. In the future, we plan to investigate how results in these parallel collectors relate to our model, where objects may be laid out in a variety of ways.

Summary and Future Work

We presented two extensions to the Gibbon compiler and LoCal, concerning parallelism and memory management. First, we have shown how a practical form of task parallelism can be reconciled with dense data representations. We demonstrated this result inside a compiler designed to implicitly transform programs to operate on such dense representations. For a set of tree-manipulating programs we considered in Section 3.4, this experimental system yielded better performance than existing best-in-class compilers. To build on what we have presented in this paper, we plan to explore automatic granularity control [**4, 3**]; this would remove the last major source of manual tuning in Gibbon programs, which already automate data layout optimizations. Parallel Gibbon with automatic granularity control would represent the dream of implicitly parallel functional programming with good absolute wall-clock performance. While our current approach supports limited examples of data parallelism–friendly data structures beyond trees, such as dense arrays (Section 3.3.6), we plan to further generalize our system by adding additional data structures that capture mutable sparse and dense multi-dimensional data. We plan to support limited in-place mutation of densely-encoded algebraic data, by adding primitives based on linear types, which we expect to mesh well with the implicitly parallel functional paradigm. While Parallel Gibbon already out-performs competing parallel, functional approaches, we expect these additional features will both improve programmability (by relieving the programmer of the burden of granularity control) and performance (by supporting more efficient parallel structures and strategies).

Second, we presented a new approach to memory management for mostly-serialized heaps, as found in the Gibbon compiler and runtime system. This hybrid collector is able to allocate objects and regions quickly and coalesce objects, which were scattered at the points of their allocation, into efficient, serialized representations. This work is a first step in a new direction that invites further study and refinement. It is common in computer science to trade-off time and space using compression techniques, and these mostly-serialized heaps point to opportunities to explore these tradeoffs more deeply in the context of language's in-memory representations. More prosaic, there are additional optimizations to develop and apply to our system to further close the gap with traditional implementation techniques on their "home turf", (i.e. the

worst-case scenarios for Gibbon's native representations). Finally, a major topic of future work is to scale the approach to the parallel setting (both for the mutator and the collector).

7.1. Parallel Garbage Collection

Efficient automatic memory management is a longstanding challenge for parallel functional languages. We have focused our attention, however, only on sequential programs so far. That is, because it would be confounding, and also take a significant research and engineering effort, to address parallel garbage collection *simultaneously* with developing a collector that operates on serialiazed data representations. Extending GC-Gibbon to work efficiently in the parallel world is a promising area for future work. We have a rough sketch on what first step towards this goal would look like, which we explain next.

7.1.1. Parallel mutators. To support parallel programs in a generational setting, we would need to allow a mutator to allocate young-generation regions in parallel, using multiple threads. We could take the standard approach of using *thread-local young-generation heaps* for this purpose. Concretely, we would initialize an array of young-generations indexed by a *thread id*. Each mutator would use its host thread's id to access the corresponding young-generation and then allocate a region within that. Analogous to the young-generation, we would also need to maintain thread-local shadow-stacks—we would similarly initialize an array of shadow-stacks indexed by a thread id.

Since Parallel-Gibbon uses the Intel Cilk Plus language extension [**14**] to realize parallelism, we could use the Cilk Plus API to implement the abstraction of a thread id. Fortunately, there is an already existing mechanism in the API that is sufficient for our purposes. The function `__cilkrts_get_nworkers()` returns the total number of Cilk Plus worker threads that can run a parallel task. We could use this function to compute the size of the young-generation and shadow-stack arrays. Next, the function `__cilkrts_get_worker_number()` returns an integer indicating the Cilk Plus worker id (a.k.a. the thread) on which the callee is executing. We could use this as the thread id.

When a mutator running on a worker thread exhausts its local young-generation heap, it would perform three actions: (1) stop executing user code, (2) atomically bump a *paused mutator count* to indicate that it has paused, and (3) signal its intention to trigger a minor collection by atomically setting a global flag. But, we must wait until all the mutators have paused before starting the minor collection. To this end, we would arrange each mutator to periodically check the global flag, before a function call or a `letregion`

expression. If the flag is set, each mutator would pause itself. Eventually all mutators would come to a halt using this mechanism and the last one to do so would trigger the minor collection[1].

This simple design has some drawbacks. Checking the global flag would add overhead to function calls and `letregion` expressions. Also, this mechanism isn't a complete solution since a particular mutator might run for a long time before reaching either of these expressions, in which case the other mutators will pause indefinitely! In the future, when the collector is more mature, we intend to study and address these sore spots.

7.1.2. Parallel collection, future directions. Thus far we have only worked out how to enable parallel mutators, but it's equally important to parallelize the collector to scale the overall memory management system. Parallelizing a copying collector might seem straightforward: run multiple collector threads that evacuate different objects. But, it's hard to get the details right such that the resulting system performs well. We intend to tackle this problem in the future.

7.2. Layout Optimizations

Gibbon has many benefits: programmers no longer need to take control of low-level data representation and allocation to serialize linked structures; and rather than writing error-prone index math to access data, the Gibbon compiler automatically translates idiomatic data structure accesses into operations on the serialized representation. Gibbon also suffers from drawbacks complementary to the other approaches mentioned above. It does not attempt to match data layout to the access patterns of a program. If the particular serialization decisions for a data type chosen by Gibbon do not correspond to the behavior of functions accessing that data, then pathological behavior can result. Consider a tree laid out in left-to-right pre-order with a program that accesses that tree right-to-left. Rather than scanning straightforwardly through the structure, the program would have to jump back and forth through the buffer to access the necessary data. When encountering such cases, Gibbon does not break: instead, it inserts *shortcut pointers* to allow more-random access to structures. But this defeats the purpose of a dense, packed representation: not only are accesses no longer nicely strided through memory, but the pointers and pointer-chasing of boxed data have returned. Indeed, when Gibbon is presented with a program whose access patterns do not match the chosen data layout, the generated code can be *significantly slower* than plain, pointer-based

[1]The paused mutator count will be equal to the total number of worker threads when the last mutator pauses itself, which is when collection would begin.

representations. What we want is the best of both worlds: Gibbon's dense, packed data layouts combined with access pattern–aware decisions of *how* to lay out that data.

To that end, we have been working on an extension of the Gibbon compiler named *Marmoset* that creates dense, packed data representations that match the way a program accesses that data. Marmoset's key insight is twofold. First, that a compile-time analysis can yield insights into how a program is *likely* to access the fields of a data structure, including recursive traversal patterns. Second, that Gibbon's dense, serialized representation is ideal for taking advantage of this access information. Gibbon's compilation strategy makes explicit when field references will result in sub-optimal access patterns, making it possible to statically estimate the costs of different layout choices. This cost model can be used to drive a search process for a minimum-cost layout, subject to any additional constraints provided by the programmer. Marmoset thus analyzes the behavior of a program, and synthesizes a data layout that corresponds to that behavior. It then rewrites the data type and lowers the program to produce code that operates on a dense, packed data representation in a way that matches access patterns and data layout, resulting in improved locality.

We evaluated Marmoset using a realistic blog software application and observed a 2-4× speedup compared to Gibbon. For a microbenchmark that computes the length of a linked list, the performance improvement is more pronounced—Marmoset produces code that is 43× faster compared to Gibbon! Consider a linked list datatype with inlined variable-sized content:

```
data List = Nil | Cons Content List
```

If each element of the list is constructed using `Cons`, the traversal has to de-reference a pointer—to *jump over* the content—each time to access the tail of the list. This is an expensive operation, especially if the target memory addresses is not present in the cache. On the contrary, if the `Content` and `List` fields were swapped, then to compute the length, the program only has to traverse n bytes for a list of length n—one byte per `Cons` tag—which is extremely efficient. Essentially, Marmoset transforms the program to use the following datatype, while preserving its behavior:

```
data List' = Nil | Cons List' Content
```

The performance of the list constructed using the original `Cons` is ~43× worse than the performance with the Marmoset-optimized, flipped layout. Not only does the optimized layout have better data locality and cache behavior, but it also has to execute fewer instructions since it does not de-reference the pointer.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

[2] D. Abuaiadh, Y. Ossia, E. Petrank, and U. Silbershtein. An efficient parallel heap compaction algorithm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, page 224–236. Association for Computing Machinery, New York, NY, USA, 2004. ISBN 1581138318. URL `https://doi.org/10.1145/1028976.1028995`.

[3] U. A. Acar, V. Aksenov, A. Charguéraud, and M. Rainey. Provably and practically efficient granularity control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 214–228. ACM, New York, NY, USA, 2019. ISBN 978-1-4503-6225-2. URL `http://mike-rainey.site/papers/oracle-ppop19-long.pdf`.

[4] U. A. Acar, A. Charguéraud, , A. Guatto, M. Rainey, and F. Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. 2018. URL `http://mike-rainey.site/papers/heartbeat.pdf`.

[5] T. A. Anderson, H. Liu, L. Kuper, E. Totoni, J. Vitek, and T. Shpeisman. Parallelizing Julia with a Non-Invasive DSL. In P. Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017. ISBN 978-3-95977-035-4. ISSN 1868-8969. URL `http://drops.dagstuhl.de/opus/volltexte/2017/7269`.

[6] J. Arora, S. Westrick, and U. A. Acar. Provably space-efficient parallel functional programming. *Proc. ACM Program. Lang.*, 5 (POPL), jan 2021. URL `https://doi.org/10.1145/3434299`.

[7] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11:598–632, October 1989. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/69558.69562`.

[8] 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. Association for Computing Machinery, New York, NY, USA, 2013. ISBN 9781450319225. URL `https://doi.org/10.1145/2442516.2442525`.

[9] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2(POPL), Dec. 2017. URL `https://doi.org/10.1145/3158093`.

[10] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page

171–183. Association for Computing Machinery, New York, NY, USA, 1996. ISBN 0897917693. URL https://doi.org/10.1145/237721.237771.

[11] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, page 344–358. Association for Computing Machinery, New York, NY, USA, 2003. ISBN 1581137125. URL https://doi.org/10.1145/949305.949336.

[12] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 22–32. Association for Computing Machinery, New York, NY, USA, 2008. ISBN 9781595938602. URL https://doi.org/10.1145/1375581.1375586.

[13] G. E. Blelloch. Nesl: A nested data-parallel language. Technical report, USA, 1992.

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 207–216. Association for Computing Machinery, New York, NY, USA, 1995. ISBN 0897917006. URL https://doi.org/10.1145/209936.209958.

[15] D. G. Bobrow and D. W. Clark. Compact encodings of list structure. *ACM Trans. Program. Lang. Syst.*, 1(2):266–286, oct 1979. ISSN 0164-0925. URL https://doi.org/10.1145/357073.357081.

[16] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 3–14. Association for Computing Machinery, New York, NY, USA, 2011. ISBN 9781450304863. URL https://doi.org/10.1145/1926354.1926358.

[17] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, nov 1970. ISSN 0001-0782. URL https://doi.org/10.1145/362790.362798.

[18] A. Chlipala. An optimizing compiler for a purely functional web-application language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 10–21. ACM, New York, NY, USA, 2015. ISBN 978-1-4503-3669-7. URL http://doi.acm.org/10.1145/2784731.2784741.

[19] A. Chlipala. Ur/web: A simple model for programming the web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 153–165. Association for Computing Machinery, New York, NY, USA, 2015. ISBN 9781450333009. URL https://doi.org/10.1145/2676726.2677004.

[20] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel sah k-d tree construction. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, page 77–86. Eurographics Association, Goslar, DEU, 2010.

[21] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, page 37–48. Association for Computing Machinery, New York, NY, USA, 2004. ISBN 1581139454. URL https://doi.org/10.1145/1029873.1029879.

[22] G. Developers. System fc, as implemented in ghc. URL https://gitlab.haskell.org/minimario/ghc/-/blob/master/docs/core-spec/core-spec.pdf.

[23] M. Elsman and N. Hallenberg. On the effects of integrating region-based memory management and generational garbage collection in ml. In *International Symposium on Practical Aspects of Declarative Languages*, pages 95–112. Springer, 2020.

[24] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[25] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, Sept. 1977. ISSN 0098-3500. URL https://doi.org/10.1145/355744.355745.

[26] M. Goldfarb, Y. Jo, and M. Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, SC '13, 2013.

[27] Google. Flatbuffers, 2014. URL https://google.github.io/flatbuffers/.

[28] A. Guatto, S. Westrick, R. Raghunathan, U. Acar, and M. Fluet. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 81–93. Association for Computing Machinery, New York, NY, USA, 2018. ISBN 9781450349826. URL https://doi.org/10.1145/3178487.3178494.

[29] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985. ISSN 0164-0925. URL https://doi.org/10.1145/4472.4478.

[30] T. Harris and S. Singh. Feedback directed implicit parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 251–264. Association for Computing Machinery, New York, NY, USA, 2007. ISBN 9781595938152. URL https://doi.org/10.1145/1291151.1291192.

[31] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, page 150–156. Association for Computing Machinery, New York, NY, USA, 2002. ISBN 1581135394. URL https://doi.org/10.1145/512429.512449.

[32] G. Keller and M. M. T. Chakravarty. Flattening trees. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, Euro-Par '98, pages 709–719. Springer-Verlag, Berlin, Heidelberg, 1998. ISBN 3540649522.

[33] K. Kodama, K. Suenaga, and N. Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. In *Asian Symposium on Programming Languages and Systems*, pages 41–56. Springer, 2004.

[34] C. Koparkar, M. Rainey, M. Vollmer, M. Kulkarni, and R. R. Newton. Efficient tree-traversals: Reconciling parallelism and dense data representations. *Proc. ACM Program. Lang.*, 5(ICFP), 2021.

[35] L. Kuper, A. Todd, S. Tobin-Hochstadt, and R. R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 2–14. Association for Computing Machinery, New York, NY, USA, 2014. ISBN 9781450327848. URL https://doi.org/10.1145/2594291.2594312.

[36] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

[37] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The ocaml system release. Feb 2020. URL https://ocaml.org/releases/4.10/htmlman/index.html.

[38] J. Makino. Vectorization of a treecode. *J. Comput. Phys.*, 87:148–160, March 1990. ISSN 0021-9991. URL http://portal.acm.org/citation.cfm?id=78582.78602.

[39] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, page 11–20. Association for Computing Machinery, New York, NY, USA, 2008. ISBN 9781605581347. URL https://doi.org/10.1145/1375634.1375637.

[40] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. *SIGPLAN Not.*, 46(12):71–82, Sept. 2011. ISSN 0362-1340. URL https://doi.org/10.1145/2096148.2034685.

[41] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, page 65–78. Association for Computing Machinery, New York, NY, USA, 2009. ISBN 9781605583327. URL https://doi.org/10.1145/1596550.1596563.

[42] M. D. McIlroy. Power series, power serious. *J. Funct. Program.*, 9(3):325–337, may 1999. ISSN 0956-7968. URL https://doi.org/10.1017/S0956796899003299.

[43] R. Menon and L. Dagum. Openmp: An industry-standard api for shared-memory programming. *Computing in Science and Engineering*, v(01):46–55, jan 1998. ISSN 1558-366X.

[44] L. A. Meyerovich, T. Mytkowicz, and W. Schulte. Data parallel programming for irregular tree computations. In *HotPAR*. USENIX, May 2011. URL https://www.microsoft.com/en-us/research/publication/data- parallel-programming-for-irregular-tree-computations/.

[45] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8026–8037, 2019.

[47] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. *ACM SIGPLAN Notices*, 38(1):172–184, 2003.

[48] R. Raghunathan, S. K. Muller, U. A. Acar, and G. Blelloch. Hierarchical memory management for parallel programs. *ACM SIGPLAN Notices*, 51(9):392–406, 2016.

[49] M. Rainey, K. A. Hale, R. Newton, N. Hardavellas, S. Campanoni, P. Dinda, and U. Acar. Task parallel assembly language for uncompromising parallelism. 2021.

[50] J. Reppy, C. V. Russo, and Y. Xiao. Parallel concurrent ml. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, page 257–268. Association for Computing Machinery, New York, NY, USA, 2009. ISBN 9781605583327. URL https://doi.org/10.1145/1596550.1596588.

[51] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, page 201–212. Association for Computing Machinery, New York, NY, USA, 2004. ISBN 1581139055. URL https://doi.org/10.1145/1016850.1016878.

[52] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley. Taking off the gloves with reference counting immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*,

OOPSLA '13, page 93–110. Association for Computing Machinery, New York, NY, USA, 2013. ISBN 9781450323741. URL https://doi.org/10.1145/2509136.2509527.

[53] A. Shaikhha, A. Fitzgibbon, S. Peyton Jones, and D. Vytiniotis. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 12–23. ACM, New York, NY, USA, 2017. ISBN 978-1-4503-5181-2. URL http://doi.acm.org/10.1145/3122948.3122949.

[54] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70. Association for Computing Machinery, New York, NY, USA, 2012. ISBN 9781450312134. URL https://doi.org/10.1145/2312005.2312018.

[55] J. Siek, C. Factora, A. Kuhlenschmidt, R. Newton, S. Ryan, C. Swords, M. Vitousek, and M. Vollmer. Essentials of compilation: An incremental approach, 2020. URL https://iucompilercourse.github.io/IU-P423-P523-E313-E513-Fall-2020/.

[56] K. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, and A. Madhavapeddy. Retrofitting parallelism onto ocaml. volume 4. Association for Computing Machinery, New York, NY, USA, Aug. 2020. URL https://doi.org/10.1145/3408995.

[57] K. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, and A. Madhavapeddy. Retrofitting parallelism onto ocaml. *arXiv preprint arXiv:2004.11663*, 2020.

[58] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, page 53–66. Association for Computing Machinery, New York, NY, USA, 2007. ISBN 159593393X. URL https://doi.org/10.1145/1190315.1190324.

[59] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, jul 1998. ISSN 0164-0925. URL https://doi.org/10.1145/291891.291894.

[60] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher Order Symbol. Comput.*, 17(3):245–265, Sept. 2004. ISSN 1388-3690. URL https://doi.org/10.1023/B:LISP.0000029446.78563.a4.

[61] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 188–201. Association for Computing Machinery, New York, NY, USA, 1994. ISBN 0897916360. URL https://doi.org/10.1145/174675.177855.

[62] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997. ISSN 0890-5401. URL http://dx.doi.org/10.1006/inco.1996.2613.

[63] K. Ueno and A. Ohori. A fully concurrent garbage collector for functional programs on multicore processors. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 421–433, 2016.

[64] K. Varda. Cap'n Proto, 2015. URL https://capnproto.org/.

[65] M. Vollmer, C. Koparkar, M. Rainey, L. Sakka, M. Kulkarni, and R. R. Newton. Local: A language for programs operating on serialized data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*,

PLDI 2019, pages 48–62. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450367127. URL https://doi.org/10.1145/3314221.3314631.

[66] M. Vollmer, S. Spall, B. Chamith, L. Sakka, C. Koparkar, M. Kulkarni, S. Tobin-Hochstadt, and R. R. Newton. Compiling Tree Transforms to Operate on Packed Representations. In P. Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:29. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017. ISBN 978-3-95977-035-4. ISSN 1868-8969. URL http://drops.dagstuhl.de/opus/volltexte/2017/7273.

[67] S. Westrick, R. Yadav, M. Fluet, and U. A. Acar. Disentanglement in nested-parallel programs. *Proc. ACM Program. Lang.*, 4 (POPL), Dec. 2019. URL https://doi.org/10.1145/3371115.

[68] E. Z. Yang, G. Campagna, O. S. Ağacan, A. El-Hassany, A. Kulkarni, and R. R. Newton. Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 362–374. ACM, New York, NY, USA, 2015. ISBN 978-1-4503-3669-7. URL http://doi.acm.org/10.1145/2784731.2784735.

[69] W. Zhao, S. M. Blackburn, and K. S. McKinley. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 76–91. Association for Computing Machinery, New York, NY, USA, 2022. ISBN 9781450392655. URL https://doi.org/10.1145/3519939.3523440.

Sample Gibbon Programs

```
-- — Compute the nth fibonacci number, in parallel.
parfib :: Int → Int → Int
parfib cutoff n =
  if n ≤ 1
  then n
  else if n ≤ cutoff
       then seqfib n
       else let (x,y) = (parfib cutoff (n-1)) ‖
                        (parfib cutoff (n-2))
            in x + y

-- — Parallel 'buildTreeHvyLf'.
buildTreeHvyLf :: Int → Int → Tree
buildTreeHvyLf cutoff i =
  if i ≤ 0
  then Leaf (seqfib 20)
  else if i ≤ cutoff
       then buildTreeHvyLf_seq i
       else let (x,y) = (buildTreeHvyLf cutoff (i-1)) ‖
                        (buildTreeHvyLf cutoff (i-1))
            in Node i x y
```

FIGURE A.1. Programs for `fib`, buildtreeHvyLf.

```
type Point3d = (Float, Float, Float)

-- In Gibbon, the Kdtree type will be represented in a dense, mostly-serialized format:
data KdTree = KdLeaf { x :: Float, y :: Float, z :: Float }
            | KdNode { x :: Float, y :: Float, z :: Float
                     , total_elems   :: Int    -- Number of elements in this node
                     , split_axis    :: Int    -- (0: x, 1: y, 2: z)
                     , split_value   :: Float
                     , min :: Point3d, max :: Point3d
                     , left :: KdTree, right :: KdTree
                     }
            | KdEmpty

-- -- Maps an array of points to an array of their nearest neighbor.
allNearest :: KdTree → Array Point3d → Array Point3d
allNearest tr ls =
    -- parallel map with a chunk-size of 1024.
    par_map 1024 (\p → nearest tr p) ls
  where
    nearest :: KdTree → Point3d → Point3d
    nearest = ···
```

FIGURE A.2. `allNearest` in Gibbon's front-end language (Haskell).

Type-Safety Proof for LoCal$^{\text{par}}$

This section contains typing rules for LoCal$^{\text{par}}$ (Section B.1) and the complete proof of type-safety for LoCal$^{\text{par}}$ (Section B.2).

B.1. Typing Rules for LoCal

$$\frac{\begin{array}{l}\text{[T-Function-Definition]}\\ \quad \Gamma; \Sigma; C; A; N \vdash A; N'; e : \tau@l^r \qquad l^r \notin N' \\ \quad \forall_{i \in \{1,\ldots,n\}}.\exists_j.\overrightarrow{l_i^{r_i}} = \overrightarrow{l_j'^{r_j'}} \qquad \exists_j.l^r = \overrightarrow{l_j'^{r_j'}}\end{array}}{\begin{array}{l}\vdash_{fun} f : \forall_{\overrightarrow{l'^{r'}}}.\overrightarrow{\tau@l^r} \to \tau@l^r; f\,\overrightarrow{x} = e \\ \text{where}\ \ \Gamma = \{\, \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1@l_1^{r_1}},\ \ldots\ , \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n@l_n^{r_n}} \,\} \\ \qquad\quad \Sigma = \{\, \overrightarrow{l_1^{r_1}} \mapsto \overrightarrow{\tau_1},\ \ldots\ , \overrightarrow{l_n^{r_n}} \mapsto \overrightarrow{\tau_n} \,\} \\ \qquad\quad C = \emptyset;\ A = \{\, r \mapsto l^r \,\};\ N = \{\, l^r \,\}; \\ \qquad\quad n = |\overrightarrow{x}| = |\overrightarrow{\tau@l^r}|\end{array}}$$

$$\frac{\begin{array}{l}\text{[T-Program]}\\ \quad \vdash_{fun} \overrightarrow{fd} \qquad \Gamma; \Sigma; C; A; N \vdash A'; N'; e : \tau@l^r\end{array}}{\vdash_{prog} A'; N'; \overrightarrow{dd}\,; \overrightarrow{fd}\,; e : \tau@l^r}$$

$$\begin{array}{l}\text{where}\ \ \Gamma = \emptyset;\ \Sigma = \emptyset \\ \qquad\quad C = \{\, l^r \mapsto \texttt{start}\ r \,\};\ A = \{\, r \mapsto l^r \,\};\ N = \{\, l^r \,\}\end{array}$$

Figure B.1. A copy of the typing rules for LoCal given in [**65**]. See also Figure B.3.

$$\text{[T-Var]}\quad \frac{\Gamma(x) = \tau@l^r \qquad \Sigma(l^r) = \tau}{\Gamma;\Sigma;C;A;N \vdash A;N;x : \tau@l^r} \qquad \text{[T-Concrete-Loc]}\quad \frac{\Sigma(l^r) = \tau}{\Gamma;\Sigma;C;A;N \vdash A;N;\langle r,i\rangle^l : \tau@l^r}$$

$$\text{[T-Let]}\quad \frac{\begin{array}{l}\Gamma;\Sigma;C;A;N \vdash A';N';e_1 : \tau_1@{l_1}^{r_1} \quad {l_1}^{r_1} \in N \quad {l_1}^{r_1} \notin N' \\ \Gamma';\Sigma';C;A';N' \vdash A'';N'';e_2 : \tau_2@{l_2}^{r_2} \quad {l_2}^{r_2} \in N\end{array}}{\begin{array}{l}\Gamma;\Sigma;C;A;N \vdash A'';N'';\texttt{let } x : \tau_1@{l_1}^{r_1} = e_1 \texttt{ in } e_2 : \tau_2@{l_2}^{r_2} \\ \text{where } \Gamma' = \Gamma \cup \{ x \mapsto \tau_1@{l_1}^{r_1} \};\ \Sigma' = \Sigma \cup \{ {l_1}^{r_1} \mapsto \tau_1 \}\end{array}}$$

$$\text{[T-LetRegion]}\quad \frac{\Gamma;\Sigma;C;A';N \vdash A'';N';e : \tau@{l'}^{r'} \quad {l'}^{r'} \in N}{\begin{array}{l}\Gamma;\Sigma;C;A;N \vdash A'';N';\texttt{letregion } r \texttt{ in } e : \tau@{l'}^{r'} \\ \text{where } A' = A \cup \{ r \mapsto \emptyset \}\end{array}}$$

$$\text{[T-LetLoc-Start]}\quad \frac{A(r) = \emptyset \qquad l^r \notin N'' \qquad {l'}^{r'} \neq l^r \qquad {l'}^{r'} \in N \qquad \Gamma;\Sigma;C';A';N' \vdash A'';N'';e : \tau'@{l'}^{r'}}{\begin{array}{l}\Gamma;\Sigma;C;A;N \vdash A'';N'';\texttt{letloc } l^r = \texttt{start } r \texttt{ in } e : \tau'@{l'}^{r'} \\ \text{where } C' = C \cup \{ l^r \mapsto \texttt{start } r \};\ A' = A \cup \{ r \mapsto l^r \};\ N' = N \cup \{ l^r \}\end{array}}$$

$$\text{[T-LetLoc-Tag]}\quad \frac{A(r) = {l'}^r \qquad {l'}^r, {l''}^{r''} \in N \qquad l^r \notin N'' \qquad l^r \neq {l''}^{r''} \qquad \Gamma;\Sigma;C';A';N' \vdash A'';N'';e : \tau''@{l''}^{r''}}{\begin{array}{l}\Gamma;\Sigma;C;A;N \vdash A'';N'';\texttt{letloc } l^r = ({l'}^r + 1) \texttt{ in } e : \tau''@{l''}^{r''} \\ \text{where } C' = C \cup \{ l^r \mapsto ({l'}^r + 1) \};\ A' = A \cup \{ r \mapsto l^r \};\ N' = N \cup \{ l^r \}\end{array}}$$

$$\text{[T-LetLoc-After]}\quad \frac{\begin{array}{l}A(r) = {l_1}^r \\ \Sigma({l_1}^r) = \tau' \qquad {l_1}^r \notin N \qquad l^r \notin N'' \qquad l^r \neq {l'}^{r'} \qquad \Gamma;\Sigma;C';A';N' \vdash A'';N'';e : \tau'@{l'}^{r'} \qquad {l'}^{r'} \in N\end{array}}{\begin{array}{l}\Gamma;\Sigma;C;A;N \vdash A'';N'';\texttt{letloc } l^r = \texttt{after } \tau'@{l_1}^r \texttt{ in } e : \tau'@{l'}^{r'} \\ \text{where } C' = C \cup \{ l^r \mapsto \texttt{after } \tau'@{l_1}^r \};\ A' = A \cup \{ r \mapsto l^r \};\ N' = N \cup \{ l^r \}\end{array}}$$

FIGURE B.2. A copy of remaining typing rules for LoCal given in [**65**].

[T-App]

$$\frac{|\overrightarrow{l'^{r'}}| = |\overrightarrow{l''^{r''}}| \quad |\overrightarrow{v}| = |\overrightarrow{x}| \quad \Gamma;\Sigma;C;A;N \vdash A;N;\overrightarrow{v_i} : \overrightarrow{\tau_i@l_i^{r_i}} \qquad l^r \in N \qquad A(r) = l^r \quad \forall_i.\exists_j.\overrightarrow{l_i'''^{r_i'''}} = \overrightarrow{l_j''^{r_j''}} \wedge \overrightarrow{l_i^{r_i}} = \overrightarrow{l_j'^{r_j'}} \quad \exists_j.l'''^{r'''} = \overrightarrow{l_j''^{r_j''}} \wedge l^r = \overrightarrow{l_j'^{r_j'}}}{\Gamma;\Sigma;C;A;N \vdash A;N';f\ [\overrightarrow{l'^{r'}}]\ \overrightarrow{v} : \tau@l^r}$$

where $f : \forall_{\overrightarrow{l''^{r''}}}.\overrightarrow{\tau_i@l_i'''^{r_i'''}} \to \tau@l'''^{r'''}; (f\overrightarrow{x} = e) = Function(f)$

$N' = N - \{ l^r \};\ n = |\overrightarrow{v}|;\ i \in \{ 1, \ldots, n \}$

[T-DataConstructor]

$$\frac{TypeOfCon(K) = \tau \qquad TypeOfField(K,i) = \overrightarrow{\tau_i'} \quad l^r \in N \qquad A(r) = \overrightarrow{l_n^r} \quad \text{if } n \neq 0 \quad \text{else } l^r \quad C(\overrightarrow{l_1^r}) = l^r + 1 \qquad C(\overrightarrow{l_{j+1}^r}) = \texttt{after}\ (\overrightarrow{\tau_j'}@\overrightarrow{l_j'^r}) \quad \Gamma;\Sigma;C;A;N \vdash A;N;\overrightarrow{v_i} : \overrightarrow{\tau_i'@l_i^r}}{\Gamma;\Sigma;C;A;N \vdash A';N';K\ l^r\ \overrightarrow{v} : \tau@l^r}$$

where $A' = A \cup \{ r \mapsto l^r \};\ N' = N - \{ l^r \}$

$n = |\overrightarrow{v}|;\ i \in I = \{ 1, \ldots, n \};\ j \in I - \{ n \}$

[T-Case]

$$\frac{\Gamma;\Sigma;C;A;N \vdash A;N;v : \tau'@l'^{r'} \qquad l^r \in N \quad \tau';\Gamma;\Sigma;C;A;N \vdash_{pat} A';N';\overrightarrow{pat_i} : \tau@l^r}{\Gamma;\Sigma;C;A;N \vdash A';N';\texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat} : \tau@l^r}$$

where $n = |\overrightarrow{pat}|;\ i \in \{ 1, \ldots, n \}$

[T-Pattern]

$$\frac{TypeOfCon(K) = \tau'' \qquad ArgTysOfConstructor(K) = \overrightarrow{\tau'} \qquad \Sigma(l^r) = \tau \quad l^r \neq \overrightarrow{l_i'^{r'}} \qquad \Gamma';\Sigma';C;A;N \vdash A';N';e : \tau@l^r}{\tau'';\Gamma;\Sigma;C;A;N \vdash_{pat} A';N';K\ \overrightarrow{(x : \tau'@l'^{r'})}\ \to\ e : \tau@l^r}$$

where $\Gamma' = \Gamma \cup \{ \overrightarrow{x_1} \mapsto \overrightarrow{\tau_1'@l_1'^{r'}}, \ldots, \overrightarrow{x_n} \mapsto \overrightarrow{\tau_n'@l_n'^{r'}} \}$

$\Sigma' = \Sigma \cup \{ \overrightarrow{l_1'^{r'}} \mapsto \overrightarrow{\tau_1'}, \ldots, \overrightarrow{l_n'^{r'}} \mapsto \overrightarrow{\tau_n'} \}$

$i \in \{ 1, \ldots, n \};\ n = |\overrightarrow{\tau'}| = |\overrightarrow{x : \tau'@l'^r}|$

FIGURE B.3. A copy of remaining typing rules for LoCal given in [**65**].

B.2. Type-Safety

We state the type-safety theorem as follows:

Theorem B.2.1 (Type Safety).

$$\begin{aligned} &\textit{If } \emptyset;\mathbb{\Sigma};\mathbb{C};\mathbb{A};\mathbb{N} \vdash_{taskset} \mathbb{A}';\mathbb{N}';\mathbb{T} \ \wedge\ \mathbb{\Sigma};\mathbb{C};\mathbb{A};\mathbb{N} \vdash_{wf_{tasks}} \mathbb{T} \\ &\textit{and } \mathbb{T} \Longrightarrow_{rp}^{n} \mathbb{T}' \\ &\textit{then, either } \forall T \in \mathbb{T}'.\ TaskComplete(T) \\ &\textit{or } \exists\, \mathbb{T}''.\ \mathbb{T}' \Longrightarrow_{rp} \mathbb{T}''. \end{aligned}$$

This theorem states that, if a given task set $\mathbb{T}$ is well typed and its overall store is well formed, and if $\mathbb{T}$ makes a transition to some task set $\mathbb{T}'$ in n steps, then either all tasks in $\mathbb{T}'$ are fully evaluated or $\mathbb{T}'$ can take a step to some task set $\mathbb{T}''$. As usual, we prove this theorem by showing progress and preservation.

Lemma B.2.2 (Top Level Progress).

$$\begin{aligned} &\textit{If } \emptyset;\mathbb{\Sigma};\mathbb{C};\mathbb{A};\mathbb{N} \vdash_{taskset} \mathbb{A}';\mathbb{N}';\mathbb{T} \\ &\textit{and } \mathbb{\Sigma};\mathbb{C};\mathbb{A};\mathbb{N} \vdash_{wf_{tasks}} \mathbb{T} \\ &\textit{then } \forall T \in \mathbb{T}.\ TaskComplete(T) \\ &\textit{else } \mathbb{T} \Longrightarrow_{rp} \mathbb{T}'. \end{aligned}$$

Proof: If every $T \in \mathbb{T}$ has evaluated to a value, we have no further proof obligations. Otherwise, the obligation is to show that there is at least one task which can take a sequential or a parallel step. By inversion on T-Taskset, and the typing rule given in the premise of this lemma, we know that all tasks in the task set $\mathbb{T}$ are well-typed. Then, by inversion on T-Task, we know that the expression e it is evaluating is also well-typed. We show that there is at least one $T \in \mathbb{T}$ which can take a sequential or a parallel step, by performing induction on the typing derivation of the expression e that a task is evaluating.

Case: T-Let, $e = \texttt{let}\ x : \hat{\tau_1} = e_1\ \texttt{in}\ e_2$

$$\frac{\begin{array}{c}\text{[T-Let]} \\ \Gamma;\Sigma;C;A;N \vdash A';N';e_1 : \tau_1@{l_1}^{r_1} \quad {l_1}^{r_1} \in N \quad {l_1}^{r_1} \notin N' \\ \Gamma';\Sigma';C;A';N' \vdash A'';N'';e_2 : \tau_2@{l_2}^{r_2} \quad {l_2}^{r_2} \in N\end{array}}{\begin{array}{l}\Gamma;\Sigma;C;A;N \vdash A'';N'';\texttt{let}\ x : \tau_1@{l_1}^{r_1} = e_1\ \texttt{in}\ e_2 : \tau_2@{l_2}^{r_2} \\ \text{where } \Gamma' = \Gamma \cup \{ x \mapsto \tau_1@{l_1}^{r_1} \};\ \Sigma' = \Sigma \cup \{ {l_1}^{r_1} \mapsto \tau_1 \}\end{array}}$$

Because e is not a value, the proof obligation is to show that there is at least one task which can take a step. That is, there is a rule in the dynamic semantics whose left-hand side matches the machine configuration $\mathbb{T}$. There are two rules that can match.

(1)

[D-Par-Let-Fork]

$T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow_{rp} T_1, \ldots, (\hat{\tau}_1, cl'_1, S; M; e_1), \ldots T_n, (\hat{\tau}, cl, S; M_2; e'_2)$

where $e = (\texttt{let}\ x : \hat{\tau}_1 = e_1\ \texttt{in}\ e_2);\ \hat{\tau}_1 = \tau_1@{l_1}^{r_1}$

x_1 fresh; $cl'_1 = \langle r_1, \texttt{ivar}\ x_1 \rangle;\ e'_2 = e_2[cl'_1/x]$

$M = \{\ {l_1}^{r_1} \mapsto cl_1\ \} \cup M';\ M_2 = \{\ {l_1}^{r_1} \mapsto cl'_1\ \} \cup M'$

By inversion on D-Par-Let-Fork, the only obligation is to estalish that $cl_1 = \hat{M}({l_1}^{r_1})$. By applying the rule WF 3.2.5.1;2 for the well-formedness of the task set given in the premise of this lemma, we can obtain a well-formedness result for the store and location map used by this task: $\Sigma; C; A; N; \mathbb{T} \vdash_{wf} M; S$. We can then perform inversion on the well-formedness of the store, and apply the rule WF 3.2.5.2;3 to obtain $A; N \vdash_{wf_{ca}} M; S$. Finally, we apply WF 3.2.5.4;3 to obtain $\langle r, i \rangle = \hat{M}({l_1}^{r_1})$. Thus we have discharged the proof obligation, and this task can take a parallel step using D-Par-Let-Fork.

(2)

[D-Par-Step]

$$\frac{S; M; e \Rightarrow S'; M'; e'}{T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow_{rp} T_1, \ldots, (\hat{\tau}, cl, S'; M'; e'), \ldots T_n}$$

To obtain this result, we use Lemma B.2.3 for single thread progress. There are two preconditions in order to use this lemma: $\emptyset; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}$, and $\Sigma; C; A; N; M \vdash_{wf} S;$. The first precondition is already established at the start of the proof, by performing inversion on T-Taskset, and then on T-Task. The second precondition can be discharged by performing inversion on the on the well-formedness of the task set given in the premise of this lemma, and then applying the rule WF 3.2.5.1;2.

Case: T-DataConstructor-Ivars, $e = K\ l^r\ \overrightarrow{v}$

$$\frac{\begin{array}{c}\text{[T-DataConstructor-Ivars]}\\ TypeOfCon(K) = \tau \qquad TypeOfField(K, i) = \overrightarrow{\tau'_i}\\ l^r \in N \qquad A(r) = \overrightarrow{l_n^{\ r}} \quad \text{if } n \neq 0 \quad \text{else } l^r\\ C(\overrightarrow{l_1^{\ r}}) = l^r + 1 \qquad C(\overrightarrow{l_{j+1}^{\ r}}) = \texttt{after}\ (\overrightarrow{\tau'_j @ l_j'^r})\\ \exists_{k\in I}.\langle r, \texttt{ivar}\ x_k\rangle = \overrightarrow{v_k} \qquad \Gamma; \Sigma; C; A; N \vdash A; N; \overrightarrow{v_i} : \overrightarrow{\tau'_i @ l_i^{\ r}}\end{array}}{\begin{array}{c}\Gamma; \Sigma; C; A; N \vdash A'; N'; K\ l^r\ \overrightarrow{v} : \tau @ l^r\\ \text{where}\ \ n = |\overrightarrow{v}|;\ i \in I = \{1, \ldots, n\};\ j \in I - \{n\}\end{array}}$$

Because e is not a value, the proof obligation is to show that there is a task which can take a step. That is, there is a rule in the dynamic semantics whose left-hand side matches the machine configuration $\mathbb{T}$. There is a single rule that can match.

$$\frac{\begin{array}{c}\text{[D-Par-DataConstructor-Join]}\\ T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots, T_n \Longrightarrow_{rp} T_1, \ldots, T', \ldots, T_n\end{array}}{\begin{array}{l}\text{where}\ \ e = K\ l^r\ \overrightarrow{v};\ \langle r, \texttt{ivar}\ x_j\rangle = \overrightarrow{v_j};\ T_c \in \{T_1, \ldots, T_n\}\\ \quad T_c = (\tau_c @ l_c^{\ r}, \langle r, \texttt{ivar}\ x_j\rangle, S_c; M_c; \langle r, i_c\rangle^{l_c})\\ \quad M' = MergeM(M_c, M);\ S' = MergeS(S_c, S)\\ \quad n = |\overrightarrow{v}|;\ \overrightarrow{v'} = [\overrightarrow{v_1}, \ldots, \overrightarrow{v_{j-1}}, \langle r, i_c\rangle^{l_c}, \overrightarrow{v_{j+1}}, \ldots, \overrightarrow{v_n}]\\ \quad \tau_j = TypeOfField(K, j);\\ \quad S'' = LinkFields(S', M, \tau_j, \langle r, i_c\rangle^{l_c})\ \ \text{if } j \neq n\ \ \text{else } S'\\ \quad e' = K\ l^r\ \overrightarrow{v'};\ T' = (\hat{\tau}, cl, S''; M'; e')\end{array}}$$

If any of the fields of the data constructor have evaluated to an ivar, the only rule that can match is D-Par-DataConstructor-Join. By inversion on D-Par-DataConstructor-Join, the only obligation then is to find a task T_c which supplies a value of that ivar. To obtain this result we perform inversion on the well-formedness of the task set given in the premise of this lemma, and apply the rule WF 3.2.5.1;1. Thus, we know that exactly one such task T_c exists. There are two possible states in which the task T_c may be in — it may have evaluated its expression to a value, or not. If the former is true, then the rule D-Par-DataConstructor-Join matches, and this task can take a step. Otherwise, the T_c, or one of its child tasks can take a step, and thus this case.

Case: T-Case, $e = \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat} : \tau@l^r$

Because e is not a value, the proof obligation is to show that there is a task which can take a step. That is, there is a rule in the dynamic semantics whose left-hand side matches the machine configuration $\mathbb{T}$. There are two rules that can match: D-Par-Case-Join, or D-Par-Step. With respect to D-Par-Case-Join, it can be discharged by using similar reasoning as the T-DataConstructor-Ivars case. As for D-Par-Step, we use Lemma B.2.3 for single thread progress like earlier.

Case:

The cases for T-DataConstructor, T-LetRegion, T-LetLoc-Tag, T-LetLoc-Start, T-LetLoc-After, T-Var, and T-Concrete-Loc use the Lemma B.2.3 for single thread progress, and can be discharged by using similar reasoning to the previous cases.

■

Lemma B.2.3 (Single Thread Progress).

$$\begin{aligned} &\textit{If}\ \ \emptyset; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau} \\ &\textit{and}\ \ \Sigma; C; A; N; M \vdash_{wf} S; \\ &\textit{then}\ \ e\ \textit{value} \\ &\textit{else}\ \ S; M; e \Longrightarrow S'; M'; e'. \end{aligned}$$

Proof: The proof is by induction on the typing derivation of e.

Case: T-LetLoc-After, $e = \texttt{letloc}\ l^r = \texttt{after}\ \tau'@{l_1}^r\ \texttt{in}\ e'$

[T-LetLoc-After]

$$\frac{A(r) = {l_1}^r \qquad \Sigma({l_1}^r) = \tau' \qquad {l_1}^r \notin N \qquad l^r \notin N'' \qquad l^r \neq {l'}^{r'} \qquad \Gamma; \Sigma; C'; A'; N' \vdash A''; N''; e : \tau'@{l'}^{r'} \qquad {l'}^{r'} \in N}{\Gamma; \Sigma; C; A; N \vdash A''; N''; \texttt{letloc}\ l^r = \texttt{after}\ \tau'@{l_1}^r\ \texttt{in}\ e : \tau'@{l'}^{r'}}$$

$$\text{where}\ \ C' = C \cup \{\, l^r \mapsto \texttt{after}\ \tau'@{l_1}^r \,\};\ A' = A \cup \{\, r \mapsto l^r \,\};\ N' = N \cup \{\, l^r \,\}$$

Because e is not a value, the proof obligation is to show that there is a task which can take a step. That is, there is a rule in the dynamic semantics whose left-hand side matches the machine configuration $S; M; e$. There are two rules that can match: D-LetLoc-After, or D-LetLoc-After-NewReg. They both have a similar precondition: $cl = \hat{M}({l_1}^r)$. To obtain this result, we need to use rule WF 3.2.5.2;1 of the well-formedness of the store given in the premise of this lemma. This rule requires that $\Sigma({l_1}^r) = \tau'$, which can be obtained by

inversion on T-LetLoc-After. By inversion on WF 3.2.5.2;1, we can establish that either $\langle r, i\rangle = \hat{M}(l_1{}^r)$, or $\langle r, \texttt{ivar}\ x\rangle = \hat{M}(l_1{}^r)$ holds. The individual requirements for each case are handled by the following case analysis.

(1) $\langle r, i\rangle = \hat{M}(l_1{}^r)$

[D-LetLoc-After]

$$S; M; \texttt{letloc}\ l^r = \texttt{after}\ \tau@l_0{}^r\ \texttt{in}\ e \Rightarrow S; M'; e$$
$$\text{where}\ \ \langle r, i\rangle = \hat{M}(l_0{}^r);\ \tau; \langle r, i\rangle; S \vdash_{ew} \langle r, j\rangle$$
$$M' = M \cup \{\ l^r \mapsto \langle r, j\rangle\ \}$$

If $\langle r, i\rangle = \hat{M}(l_1{}^r)$, the only rule that can match is D-LetLoc-After. The only remaining obligation is to show that an end-witness j for the value allocated at the address $\langle r, i\rangle$ exists: $\tau'; \langle r, i\rangle; S \vdash_{ew} \langle r, j\rangle$. This can be discharged by applying the rule WF 3.2.5.2;1. If $\langle r, i\rangle = \hat{M}(l_1{}^r)$, the first disjunct of WF 3.2.5.2;1 holds, and thus discharges our required obligation.

(2) $\langle r, \texttt{ivar}\ x\rangle = \hat{M}(l_1{}^r)$

[D-LetLoc-After-NewReg]

$$S; M; \texttt{letloc}\ l^r = \texttt{after}\ \tau@l_0{}^r\ \texttt{in}\ e \Rightarrow S'; M'; e$$
$$\text{where}\ \ \langle r, \texttt{ivar}\ x\rangle^{l_0} = \hat{M}(l_0{}^r); r'\ \text{fresh}$$
$$S' = S \cup \{\ r' \mapsto \emptyset\ \};\ M' = M \cup \{\ l^r \mapsto \langle r, \&(r', 0)\rangle\ \}$$

This rule has no remaining obligations, and matches directly, and thus this case.

Case:

The cases for T-DataConstructor, T-Let, T-LetRegion, T-LetLoc-Tag, T-LetLoc-Start, T-Var, T-Concrete-Loc, T-Appare similar to the proof of progress for sequential LoCal given in [**65**].

■

Lemma B.2.4 (Top Level Preservation).

$$\textit{If}\ \ \emptyset; \mathbb{\Sigma}; \mathbb{C}; \mathbb{A}; \mathbb{N} \vdash_{taskset} \mathbb{A}'; \mathbb{N}'; \mathbb{T}$$
$$\textit{and}\ \ \mathbb{\Sigma}; \mathbb{C}; \mathbb{A}; \mathbb{N} \vdash_{wf_{tasks}} \mathbb{T}$$
$$\textit{and}\ \ \mathbb{T} \Longrightarrow_{rp} \mathbb{T}'$$
$$\textit{then for some}\ \ \textit{deepSuperSetEqS}(\mathbb{\Sigma}', \mathbb{\Sigma}),\ \textit{deepSuperSetEqC}(\mathbb{C}', \mathbb{C}),\ \mathbb{A}'' \supseteq \mathbb{A}',\ \mathbb{N}'' \supseteq \mathbb{N}'.$$
$$\emptyset; \mathbb{\Sigma}'; \mathbb{C}'; \mathbb{A}''; \mathbb{N}'' \vdash_{taskset} \mathbb{A}'''; \mathbb{N}'''; \mathbb{T}'$$
$$\textit{and}\ \ \mathbb{\Sigma}'; \mathbb{C}'; \mathbb{A}''; \mathbb{N}'' \vdash_{wf_{tasks}} \mathbb{T}'.$$

Proof: The proof is by induction on the given derivation of the dynamic semantics.

Case: D-Par-Let-Fork

$$\begin{array}{l}\text{[D-Par-Let-Fork]}\\ T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow_{rp} T_1, \ldots, (\hat{\tau}_1, cl'_1, S; M; e_1), \ldots T_n, (\hat{\tau}, cl, S; M_2; e'_2)\\ \quad \text{where } \; e = (\texttt{let } x : \hat{\tau}_1 = e_1 \texttt{ in } e_2);\; \hat{\tau}_1 = \tau_1 @ {l_1}^{r_1}\\ \quad\quad x_1 \text{ fresh};\; cl'_1 = \langle r_1, \texttt{ivar } x_1 \rangle;\; e'_2 = e_2[cl'_1/x]\\ \quad\quad M = \{\, {l_1}^{r_1} \mapsto cl_1 \,\} \cup M';\; M_2 = \{\, {l_1}^{r_1} \mapsto cl'_1 \,\} \cup M'\end{array}$$

Let $\Sigma = \mathbb{\Sigma}(cl)$, $C = \mathbb{C}(cl)$, $A = \mathbb{A}(cl)$, and $N = \mathbb{A}(cl)$ be the environments corresponding to the task $(\hat{\tau}, cl, S; M; e)$. We instantiate the new environment maps as:

$$\begin{array}{l}\Sigma' = \Sigma \cup \{\, {l_1}^{r_1} \mapsto \hat{\tau}_1 \,\}\\ \mathbb{\Sigma}' = \mathbb{\Sigma} \cup \{\, cl'_1 \mapsto \Sigma, cl \mapsto \Sigma' \,\}\\ \mathbb{C}' = \mathbb{C} \cup \{\, cl'_1 \mapsto C \,\}\\ \mathbb{A}'' = \mathbb{A}' \cup \{\, cl'_1 \mapsto A \,\}\\ \mathbb{N}'' = \mathbb{N}' \cup \{\, cl'_1 \mapsto N \,\}\end{array}$$

The task executing the body of the let expression has the target location cl. In $\mathbb{\Sigma}'$, we include an updated entry, $(cl \mapsto \Sigma')$, to establish the allocation of the bound expression. And we extend the environments $\mathbb{\Sigma}'$, $\mathbb{C}'$, $\mathbb{A}'$, and $\mathbb{N}'$ to contain an entry for cl'_1, which is the target location of the task executing the bound expression, as the incoming environments won't contain an entry for cl'_1, which is a fresh concrete location generated by D-Par-Let-Fork.

(1) The first obligation is to show that the result $\mathbb{T}'$ of the evaluation step is well-typed with respect to the environments $\mathbb{\Sigma}'$, $\mathbb{C}'$, $\mathbb{A}''$, and $\mathbb{N}''$. By inversion on T-Taskset, the obligation then is to show that all tasks in $\mathbb{T}'$ are well-typed. By the typing rule given in the premise of this lemma and by inversion on T-Taskset, we can directly establish the well-typedness of the tasks in $\mathbb{T}$. Thus there are only two remaining obligations, namely to show that the two new tasks spawned by D-Par-Let-Fork are well-typed.

(a) Let $\Sigma = \mathbb{\Sigma}(cl'_1),\; C = \mathbb{C}(cl'_1),\; A = \mathbb{A}(cl'_1),\; N = \mathbb{A}(cl'_1)$.

Obl: $\emptyset; \Sigma; C; A; N \vdash_{task} A'; N'; (\hat{\tau}_1, cl'_1, S; M; e_1)$

By inversion on T-Task, we see that in order to prove that this task is well-typed, we must show that the expression e_1 is well-typed with respect to the environments Σ, C, A, and N. Concretely, the obligation is to show that $\emptyset; \Sigma; C; A; N \vdash A'; N'; e_1 : \hat{\tau}_1$ holds. We can

discharge this obligation as follows. By the typing rule given in the premise of this lemma, and by inversion on T-Taskset, we can establish that the task $(\hat{\tau}, cl, S; M; e)$ is well-typed. By inversion on T-Task, we establish that the expression e is well-typed: $\Gamma; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}$. Then by inversion on T-Let, we can obtain the desired result.

(b) Let $\Sigma' = \mathbb{\Sigma}(cl),\ C = \mathbb{C}(cl),\ A = \mathbb{A}(cl),\ N = \mathbb{A}(cl)$.

Obl: $\emptyset; \Sigma'; C; A; N \vdash_{task} A'; N'; (\hat{\tau}, cl, S; M_2; e_2')$

This obligation can be discharged by using similar reasoning as above. The only difference is that in order for the expression e_2' to typecheck, T-Let requires the location of bound expression, ${l_1}^{r_1}$, to be in the store typing environment. The environment Σ' that is instantiated for this task fulfils this requirement, and thus this obligation is discharged.

(2) The second obligation is to show that the result of the evaluation step is well-formed. The individual requirements, labeled WF 3.2.5.1;1 - WF 3.2.5.1;2, are handled by the following case analysis.

- Case (WF 3.2.5.1;1): for each ivar $\langle r, \texttt{ivar}\ x \rangle$ in the result $\mathbb{T}'$, there exists exactly one task in $\mathbb{T}'$ which supplies a well-typed value for it.

  By the well-formedness of the task set given in the premise of this lemma, this already holds for all ivars in the task set $\mathbb{T}$. The only remaining obligation then is to show that a corresponding unique task exists for the only new ivar introduced by D-Par-Let-Fork, namely $\langle r_1, \texttt{ivar}\ x_1 \rangle$. This obligation discharges straightforwardly by inversion on D-Par-Let-Fork, and the typing rule given in the premise of this lemma.

- Case(WF 3.2.5.1;2): the stores of all tasks in the result $\mathbb{T}'$ are well-formed.

  By the well-formedness of the task set given in the premise of this lemma, we can directly establish the well formedness of stores of all tasks in the task set $\mathbb{T}$. Thus there are only two remaining obligations, to show that the stores of the two new tasks spawned by D-Par-Let-Fork are well-formed.

  (a) Case $(\hat{\tau}_1, cl_1', S; M; e_1); \Sigma; C; A'; N'; \mathbb{T}' \vdash_{wf} S; M$:

  Since all of the environments remain unchanged in this task, this case follows immediately by inversion on the well-formedness of the task set given in the premise of this lemma.

  (b) Case $(\hat{\tau}, cl, S; M_2; e_2')$; and $\Sigma'; C; A'; N'; \mathbb{T}' \vdash_{wf} S; M_2$:

This task uses the updated environment $\Sigma' = \Sigma \cup \{ {l_1}^{r_1} \mapsto \hat{\tau_1} \}$, so we must show the store S, and location map M_2 are well-formed. The individual requirements, labeled WF 3.2.5.2;1 - WF 3.2.5.2;4, are handled by the following case analysis.

- Case (WF 3.2.5.2;1):

  $(l^r \mapsto \tau) \in \Sigma \Rightarrow$
  $(\langle r, i_1 \rangle = \hat{M}(l^r) \wedge \tau; \langle r, i_1 \rangle; S \vdash_{ew} \langle r, i_2 \rangle) \oplus$
  $(\langle r, \texttt{ivar}\ x \rangle = \hat{M}(l^r) \wedge$
  $\exists_{S', M', e'}.(\tau, \langle r, \texttt{ivar}\ x \rangle, S'; M'; e') = GetSingleWriter(\mathbb{T}, \hat{\tau}, \texttt{ivar}\ x) \wedge$
  $\langle r, i_1 \rangle = \hat{M'}(l^r) \wedge$
  $(IsVal(e') \Rightarrow \tau; \langle r, i_1 \rangle; S' \vdash_{ew} \langle r, i_2 \rangle))$

  By the well formedness of the task set given in the premise of this lemma, we establish the well formedness of stores of all tasks in the task set $\mathbb{T}$, and we obtain the result $\Sigma; C; A; N; \mathbb{T} \vdash_{wf} S; M$. Thus, for all locations in Σ, the above already holds. Then the only obligation is to show that it holds for the new location added to Σ', ${l_1}^{r_1}$. For ${l_1}^{r_1}$, the second disjunct follows straightforwardly by inversion on D-Par-Let-Fork.

- Cases (WF 3.2.5.2;2), (WF 3.2.5.2;3), and (WF 3.2.5.2;4):

  The remaining three cases also discharge straightforwardly by inversion on D-Par-Let-Fork, and on the well-formedness of the task set given in the premise of this lemma.

Case: D-Par-DataConstructor-Join

[D-Par-DataConstructor-Join]

$$T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots, T_n \Longrightarrow_{rp} T_1, \ldots, T', \ldots, T_n$$

where $e = K\ l^r\ \overrightarrow{v};\ \langle r, \texttt{ivar}\ x_j \rangle = \overrightarrow{v_j};\ T_c \in \{ T_1, \ldots, T_n \}$

$T_c = (\tau_c @ {l_c}^r, \langle r, \texttt{ivar}\ x_j \rangle, S_c; M_c; \langle r, i_c \rangle^{l_c})$
$M' = MergeM(M_c, M);\ S' = MergeS(S_c, S)$
$n = |\overrightarrow{v}|;\ \overrightarrow{v'} = [\overrightarrow{v_1}, \ldots, \overrightarrow{v_{j-1}}, \langle r, i_c \rangle^{l_c}, \overrightarrow{v_{j+1}}, \ldots, \overrightarrow{v_n}]$
$\tau_j = TypeOfField(K, j);$
$S'' = LinkFields(S', M, \tau_j, \langle r, i_c \rangle^{l_c})$ if $j \neq n$ else S'
$e' = K\ l^r\ \overrightarrow{v'};\ T' = (\hat{\tau}, cl, S''; M'; e')$

Let $\Sigma = \mathbb{\Sigma}(cl)$, $C = \mathbb{C}(cl)$, $A = \mathbb{A}(cl)$, and $N = \mathbb{A}(cl)$ be the environments corresponding to the task $(\hat{\tau}, cl, S; M; e)$, and let $\Sigma_c = \mathbb{\Sigma}(\langle r, \texttt{ivar } x\rangle)$, $C_c = \mathbb{C}(\langle r, \texttt{ivar } x\rangle)$, $A_c = \mathbb{A}(\langle r, \texttt{ivar } x\rangle)$, and $N_c = \mathbb{A}(\langle r, \texttt{ivar } x\rangle)$ be the environments corresponding to the task $(\tau_c@{l_c}^{r_c}, cl, S; M; e)$.

We instantiate the new environment maps as:

$$\mathbb{\Sigma}' = \mathbb{\Sigma} \cup \{\ cl \mapsto (\Sigma \cup \Sigma_c)\ \}$$
$$\mathbb{C}' = \mathbb{C} \cup \{\ cl \mapsto (C \cup C_c)\ \}$$

(1) The first obligation is to show that the result $\mathbb{T}'$ of the evaluation step is well-typed with respect to the environments $\mathbb{\Sigma}'$, $\mathbb{C}'$, $\mathbb{A}'$, and $\mathbb{N}'$. By inversion on T-Taskset, the obligation then is to show that all tasks in $\mathbb{T}'$ are well-typed. By the typing rule given in the premise of this lemma and by inversion on T-Taskset, we can directly establish the well-typedness of the tasks in $\mathbb{T}$. Thus the only remaining obligation, is to show that the task that is updated by the rule D-Par-DataConstructor-Join is well-typed. By inversion on T-Task, we need to show that the expression it is evaluating is well-typed. Concretely, the proof obligation is:

$$\emptyset; \Sigma'; C'; A'; N' \vdash A''; N''; K\ l^r\ \overrightarrow{v'} : \hat{\tau}$$

where $\overrightarrow{v'} = [\overrightarrow{v_1}, \ldots, \overrightarrow{v_{j-1}}, \langle r, i_c\rangle, \overrightarrow{v_{j+1}}, \ldots, \overrightarrow{v_n}]$. We discharge this case by showing that $\overrightarrow{v'}$ and $\overrightarrow{v}$ have the same type. In order to establish this result, we first perform inversion on the typing judgement given in the premise of this lemma, and then on T-Taskset and T-Task, to obtain a result that $K\ l^r\ \overrightarrow{v}$ is well-typed. Since $\overrightarrow{v'}$ is obtained by replacing the j^{th} value in $\overrightarrow{v}$, namely $\langle r, \texttt{ivar } x_j\rangle$, with the value $\langle r, i_c\rangle$, if the values $\langle r, \texttt{ivar } x_j\rangle$ and $\langle r, i_c\rangle$, are of the same type, then $\overrightarrow{v'}$ and $\overrightarrow{v}$ should also have the same type. By performing inversion on the well-formedness of the task set given in the premise of this lemma, and then applying the rule WF 3.2.5.1;1, we establish that there is exactly one task in the task set which supplies a well-typed value for $\texttt{ivar } x_j$. Thus, $\langle r, i_c\rangle$ must have the same type as $\langle r, \texttt{ivar } x_j\rangle$, and thus this case.

(2) The second obligation is to show that the result of the evaluation step is well-formed. The individual requirements, labeled WF 3.2.5.1;1 - WF 3.2.5.1;2, are handled by the following case analysis.

- Case (WF 3.2.5.1;1): for each ivar $\langle r, \texttt{ivar } x\rangle$ in the result $\mathbb{T}'$, there exists exactly one task in $\mathbb{T}'$ which supplies a well-typed value for it

  By inversion on the well-formedness of the task set given in the premise of this lemma, this already holds for all ivar in $\mathbb{T}$. Since D-LetLoc-After-NewReg doesn't introduce any new ivars, this case discharges straightforwardly.

- Case (WF 3.2.5.1;2): $\Sigma; C; A'; N'; \mathbb{T}' \vdash_{wf} S''; M'$

  By inversion on the well-formedness of the task set given in the premise of this lemma, we establish that all tasks in the task set $\mathbb{T}$ are well formed. By applying WF 3.2.5.1;2, we obtain the following:

(1) $$\Sigma; C; A; N; \mathbb{T} \vdash_{wf} M; S$$

(2) $$\Sigma_c; C_c; A_c; N_c; \mathbb{T} \vdash_{wf} M_c; S_c$$

  The proof obligation is to show that the store S'' is well-formed with respect to the location map M', and the environments Σ', C', A', and N', where $M' = MergeM(M, M_c)$. To discharge this case, we perform a case analysis on the definition of $MergeM$.

  - Case 1:

    $\{\, l^r \mapsto \langle r, i_1\rangle \mid (l^r \mapsto \langle r, i_1\rangle) \in M, (l^r \mapsto \langle r, i_2\rangle) \in M_c, i_1 = i_2 \,\}$

    Identitical entries in M and M_c remain unchanged in the merged location map M'. For such entries, this case holds straightforwardly by using Results (1) and (2).

  - Case 2:

    $\{\, l^r \mapsto cl \mid (l^r \mapsto cl) \in M, l^r \notin M_c \,\}$ and $\{\, l^r \mapsto cl \mid l^r \notin M, (l^r \mapsto cl) \in M_c \,\}$

    Entries that are not shared in common by M and M_c remain unchanged in the merged location map M'. For such entries, this case holds straightforwardly by using Results (1) and (2).

  - Case 3: $\{\, l^r \mapsto \langle r, j\rangle \mid (l^r \mapsto \langle r, \texttt{ivar}\ x\rangle) \in M, (l^r \mapsto \langle r, j\rangle) \in M_c \,\}$

    When a location maps to an ivar in M, and to a concrete index in M_c, the merged location map M' keeps the concrete index and discards the ivar. To discharge this case, we must show that even when the location map is thus updated, the store S'' is well-formed. The individual requirements, labeled WF 3.2.5.2;1 - WF 3.2.5.2;4, are handled by the following case analysis.

    * Case (WF 3.2.5.2;1):

      $(l^r \mapsto \tau) \in \Sigma \Rightarrow$

      $(\langle r, i_1\rangle = \hat{M}(l^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle) \oplus$

      $(\langle r, \texttt{ivar}\ x\rangle = \hat{M}(l^r) \wedge$

      $\exists_{S_2, M_2, e_2}.(\tau, \langle r, \texttt{ivar}\ x\rangle, S_2; M_2; e_2) =$

      $GetSingleWriter(\mathbb{T}, \hat{\tau}, \texttt{ivar}\ x) \wedge$

$\langle r, i_1\rangle = \hat{M_2}(l^r) \wedge$

$(IsVal(e_2) \Rightarrow \tau; \langle r, i_1\rangle; S_2 \vdash_{ew} \langle r, i_2\rangle))$

By inversion on Result (1), we know that WF 3.2.5.2;1 holds for S. And since $(l^r \mapsto \langle r, \texttt{ivar } x\rangle) \in M$, the second disjunct of WF 3.2.5.2;1 must hold. Since this l^r is being updated to map to a concrete index, in order to discharge this case we must show that the first disjunct of WF 3.2.5.2;1 now holds for it. There are two obligations: $\langle r, j\rangle = \hat{M'}(l^r)$, and $\tau; \langle r, j\rangle; S'' \vdash_{ew} \langle r, j_2\rangle$. Since the second disjunct of WF 3.2.5.2;1 holds for the store S and location map M, we obtain the following:

(i) $\langle r, \texttt{ivar } x\rangle = \hat{M}(l^r)$

(ii) $\exists_{S_2, M_2, e_2}.(\tau, \langle r, \texttt{ivar } x\rangle, S_2; M_2; e_2) =$

$GetSingleWriter(\mathbb{T}, \hat{\tau}, \texttt{ivar } x)$

(iii) $\langle r, i_1\rangle = \hat{M_2}(l^r)$

(iv) $IsVal(e_2) \Rightarrow \tau; \langle r, i_1\rangle; S_2 \vdash_{ew} \langle r, i_2\rangle$

By applying WF 3.2.5.1;1, we can establish that there is exactly one task in $\mathbb{T}$ that can provide a well-typed value for an ivar. Thus, the task being merged by D-Par-DataConstructor-Join must be this unique task. As a result, the location map M_2, store S_2, and expression e_2, must be the same as M_c, S_c, and $\langle r, i_c\rangle$. Thus, we obtain:

(v) $\langle r, j\rangle = \hat{M_c}(l^r)$

(vi) $IsVal(\langle r, i_c\rangle) \Rightarrow \tau; \langle r, j\rangle; S_c \vdash_{ew} \langle r, j_2\rangle$

The first obligation, $\langle r, j\rangle = \hat{M'}(l^r)$, discharges by inspection on *MergeM*, which inserts the mapping $(l^r \mapsto \langle r, j\rangle)$ in M'. The second obligation, $\tau; \langle r, j\rangle; S'' \vdash_{ew} \langle r, j_2\rangle$, discharges since

$IsVal(\langle r, i_c\rangle)$ is true, and the store S'' contains all the allocations performed in S_c. Thus, this case.

* WF 3.2.5.2;2 $C \vdash_{wf_{cfc}} ; S''$
  - Case (WF 3.2.5.3;1):

    $(l^r \mapsto \texttt{start } r) \in C \Rightarrow (l^r \mapsto \langle r, 0\rangle) \in M$

    Since the address of a location at the start of a region cannot be an ivar, this case discharges trivially.
  - Case (WF 3.2.5.3;2):

    $(l^r \mapsto \texttt{start } r) \in C \Rightarrow (l^r \mapsto \langle r, 0\rangle) \in M$

Since the address of a location one past the start of a region cannot be an ivar, this case discharges trivially.

- Case (WF 3.2.5.3;3):

  $(l^r \mapsto \texttt{after}\ \tau@{l'}^r) \in C \Rightarrow$
  $(\langle r, i_1\rangle = \hat{M}({l'}^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle \wedge \langle r, i_2\rangle = \hat{M}(l^r)) \oplus$
  $(\langle r, \texttt{ivar}\ x_1\rangle = \hat{M}({l'}^r) \wedge (l^r \mapsto \langle r, \&(r_2, 0)\rangle) \in M \wedge \{\, r_2 \mapsto h \,\} \in S) \oplus$
  $(\langle r, i_1\rangle = \hat{M}({l'}^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle \wedge (l^r \mapsto \langle r, \&(r_2, 0)\rangle) \in M \wedge S(r)(i_2) = \&(r_2, 0))$

  By inversion on Result (1), we know that WF 3.2.5.3;3 holds for S. And since $(l^r \mapsto \langle r, \texttt{ivar}\ x\rangle) \in M$, the second disjunct of WF 3.2.5.3;3 must hold. But if l^r is being updated to map to a concrete index instead of an ivar by the merge, and since this case looks at state of the machine after a merge, we must show that the third disjunct of WF 3.2.5.3;3 now holds in order to discharge this case. There are three obligations:

  • $\langle r, j\rangle = \hat{M}'(l^r)$

  This obligation discharges straightforwardly by the premise of this case, and by inspection of *MergeM*, which updates l^r to map to a concrete index instead of an ivar.

  • $\tau; \langle r, j\rangle; S'' \vdash_{ew} \langle r, j_2\rangle$

  By Result 2, we get $\Sigma; C; A; N; \mathbb{T} \vdash_{wf} M_c; S_c$. And since $(l^r \mapsto \langle r, j\rangle)$, the first disjunct of WF 3.2.5.2;1 must hold. The only precondition of WF 3.2.5.2;1, $(l^r \mapsto \tau) \in \Sigma$, discharges straightforwardly by inversion on T-LetLoc-After, which is the only rule that adds a constraint $({l_1}^r \mapsto \texttt{after}\ \tau@l^r)$ to C. Thus, we get $\tau; \langle r, j\rangle; S_c \vdash_{ew} \langle r, j_2\rangle$. Since the S'' contains all the allocations performed in S_c, $\tau; \langle r, j\rangle; S'' \vdash_{ew} \langle r, j_2\rangle$ must hold as well.

  • $({l_1}^r \mapsto \langle r, j_2\rangle) \in M' \vee$
  $(S''(r)(j_2) = \&(r_2, j_3) \wedge ({l_1}^r \mapsto \langle r, \&(r_2, j_3)\rangle) \in M')$

  We can discharge this case by showing that the second disjunct holds. There are two obligations: $S''(r)(j_2) = \&(r_2, j_3)$, and $({l_1}^r \mapsto \langle r, \&(r_2, j_3)\rangle) \in M'$. The first obligation discharges by inspecting the definition of *LinkFields*, which writes the appropriate indirection to the store. To discharge the second obligation,

we perform inversion on Result 1. By inversion, we know that WF 3.2.5.3;3 must hold for M. And since $(l^r \mapsto \langle r, \texttt{ivar } x\rangle) \in M$, the second disjunct of WF 3.2.5.3;3 must hold for M. Thus, we get $({l_1}^r \mapsto \langle r, \&(r_2, j_3)\rangle) \in M$. Since M' contains this mapping from M, $({l_1}^r \mapsto \langle r, \&(r_2, j_3)\rangle) \in M'$ holds as well.

- Case $\Sigma'; C'; A'; N'; \mathbb{T}' \vdash_{wf} \{\ l^r \mapsto \langle r, j\rangle \mid (l^r \mapsto \langle r, j\rangle) \in M, (l^r \mapsto \langle r, \texttt{ivar } x\rangle) \in M_c\ \}; S''$

  This case is identical to the previous one.

* Case WF 3.2.5.2;3

  The individual requirements, labeled WF 3.2.5.4;1 - WF 3.2.5.4;4, are handled by the following case analysis.

  - Case (WF 3.2.5.4;1):

    $((r \mapsto l^r) \in A' \land l^r \in N') \Rightarrow$

    $((l^r \mapsto \langle r, i\rangle) \in M' \land i > \mathit{MaxIdx}(r, S)) \oplus (l^r \mapsto \&(r_2, i_2)) \in M'$

    By the well-formedness of the store given in the premise of this lemma, the above already holds for locations in the environments A and N. This case discharges straightforwardly by using Results (1) and (2) since D-Par-DataConstructor-Join doesn't introduce any new locations in A and N.

  - Case (WF 3.2.5.4;2)

    $((r \mapsto l^r) \in A \land \langle r, i_s\rangle = \hat{M}(l^r) \land l^r \notin N' \land \tau; \langle r, i_s\rangle; S \vdash_{ew} \langle r, i_e\rangle) \Rightarrow i_e > \mathit{MaxIdx}(r, S)$

    By inversion on T-DataConstructor-Ivars, $l^r \in N'$, and thus this case discharges immediately.

  - Case (WF 3.2.5.4;3):

    $l^r \in N' \Rightarrow \langle r, i\rangle = \hat{M'}(l^r) \land (r \mapsto (i \mapsto K)) \notin S'$

    By the well-formedness of the store given in the premise of this lemma, the above already holds for all locations in N. This case discharges straightforwardly by using Results (1) and (2) since D-Par-DataConstructor-Join doesn't introduce any new locations in N.

  - WF 3.2.5.4;4:

    $(r \mapsto \emptyset) \in A \Rightarrow (r \mapsto \emptyset) \in S$

This case discharges because, from the premise of the lemma, this property holds for the original environment A and store S, and, by inversion on T-DataConstructor-Ivars, continues to hold for A' and S'.

* Case (WF 3.2.5.2;4): $dom(\Sigma) \cap N = \emptyset$

  This case discharges because, from the premise of the lemma, this property holds for the original environments N and Σ, and, by inversion on T-DataConstructor, continues to hold for N and Σ.

Case: D-Par-Case-Join

[D-Par-Case-Join]

$$T_1, \ldots, T_c, \ldots, T_n \Longrightarrow_{rp} T_1, \ldots, T_c', \ldots T_n,$$

where

$T_c = (\hat{\tau_c}, cl_c, S_c; M_c; e_c);\ e_c = \texttt{case}\ \langle r, \texttt{ivar}\ x_c\rangle^{l_c}\ \texttt{of}\ \overrightarrow{pat}$

$T_p \in \{\ T_1, \ldots, T_n\ \} = (\tau_p @ {l_p}^r, \langle r, \texttt{ivar}\ x_c\rangle, S_p; M_p; \langle r, i_p\rangle)$

$M_3 = MergeM(M_p, M_c);\ S_3 = MergeS(S_p, S_c)$

$e_c' = \texttt{case}\ \langle r, i_p\rangle^{l_p}\ \texttt{of}\ \overrightarrow{pat}[i_p/\texttt{ivar}\ x_c];\ T_c' = (\hat{\tau_c}, cl_c, S_3; M_3; e_c')$

This case can be proved using similar reasoning as D-Par-DataConstructor-Join.

Case: D-Par-Step

[D-Par-Step]

$$\frac{S; M; e \Rightarrow S'; M'; e'}{T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow_{rp} T_1, \ldots, (\hat{\tau}, cl, S'; M'; e'), \ldots T_n}$$

Let $\Sigma = \mathbb{\Sigma}(cl)$, $C = \mathbb{C}(cl)$, $A = \mathbb{A}(cl)$, and $N = \mathbb{A}(cl)$ be the environments corresponding to the task $(\hat{\tau}, cl, S; M; e)$. We instantiate the new environment maps as:

$$\mathbb{\Sigma}' = \mathbb{\Sigma};\ \mathbb{C}' = \mathbb{C}$$

By inversion on 3.2.5.1, the two obligations are to show that the resulting $\mathbb{T}'$ is well typed, and that the stores of all tasks in it are well formed. By the typing rule given in the premise of this lemma and by inversion on T-Taskset, we can directly establish the well-typedness of the tasks in $\mathbb{T}$. Using similar reasoning, we establish that the stores of all tasks in $\mathbb{T}$ are well-formed. Thus, there are only two remaining obligations, namely to show that the task resulting from the sequential step is well-typed, and that its store is well-formed. And by inversion on T-Task, in order to show that a task is well-typed, we must show that

the expression it is evaluating is well-typed. Thus, there are two obligations that we must prove, namely $\emptyset; \Sigma'; C'; A'; N' \vdash A''; N''; e' : \hat{\tau}$, and $\Sigma'; C'; A'; N'; M' \vdash_{wf} S'$; We can discharge both of these by using Lemma B.2.5 for single thread preservation. There are three preconditions in order to use this lemma, which are handled by the following case analysis.

(1) $\emptyset; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau}$

By inversion on T-Taskset, and the typing rule given in the premise of this lemma, we know that all tasks in the task set $\mathbb{T}$ are well-typed. Thus, we obtain the result $\Sigma; C; A; N \vdash_{task} A'; N'; (\hat{\tau}, cl, S; M; e)$. Then, by inversion on T-Task, we obtain the result that the expression e is also well-typed. Thus, this obligation is discharged.

(2) $\Sigma; C; A; N; M \vdash_{wf} S$;

By the well-formedness of the task set given in the premise of this lemma, we establish the well-formedness of stores of all tasks in the task set $\mathbb{T}$, and we obtain the result $\Sigma; C; A; N; \mathbb{T} \vdash_{wf} S; M$, thus discharging this obligation.

(3) $S; M; e \Rightarrow S'; M'; e$.

This obligation discharges straightforwardly by inversion on D-Par-Step.

■

Lemma B.2.5 (Single Thread Preservation).

$$
\begin{aligned}
&\textit{If}\ \ \emptyset; \Sigma; C; A; N \vdash A'; N'; e : \hat{\tau} \\
&\textit{and}\ \ \Sigma; C; A; N; M \vdash_{wf} S; \\
&\textit{and}\ \ S; M; e \Rightarrow S'; M'; e' \\
&\textit{then for some}\ \ \Sigma' \supseteq \Sigma, C' \supseteq C, \\
&\qquad \emptyset; \Sigma'; C'; A'; N' \vdash A''; N''; e' : \hat{\tau} \\
&\textit{and}\ \ \Sigma'; C'; A'; N'; M' \vdash_{wf} S';.
\end{aligned}
$$

Proof: The proof is by induction on the given derivation of the dynamic semantics.

Case: D-LetLoc-After-NewReg

$$
\begin{aligned}
&\text{[D-LetLoc-After-NewReg]} \\
&\quad S; M; \texttt{letloc}\ l^r = \texttt{after}\ \tau@{l_0}^r\ \texttt{in}\ e \Rightarrow S'; M'; e \\
&\text{where}\ \ \langle r, \texttt{ivar}\ x\rangle^{l_0} = \hat{M}({l_0}^r); r' \text{ fresh} \\
&\quad S' = S \cup \{\, r' \mapsto \emptyset \,\};\ M' = M \cup \{\, l^r \mapsto \langle r, \&(r', 0)\rangle \,\}
\end{aligned}
$$

(1) The first obligation is to show that the result of the evaluation step is well-typed, that is

$$\emptyset; \Sigma; C'; A'; N' \vdash A''; N''; e' : \hat{\tau},$$

where $\hat{\tau} = \tau@{l'}^{r'}$. This proof obligation follows straightforwardly by inversion on T-LetLoc-After.

(2) The second obligation is to show that the result of the evaluation step is well-formed, that is

$$\Sigma; C'; A'; N'; \mathbb{T}' \vdash_{wf} S'; M'$$

The individual requirements, labeled

WF 3.2.5.2;1 WF 3.2.5.2;4, are handled by the following case analysis.

- Case (WF 3.2.5.2;1):

  $(l^r \mapsto \tau) \in \Sigma \Rightarrow$

  $(\langle r, i_1\rangle = \hat{M}(l^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle) \oplus$

  $(\langle r, \texttt{ivar}\ x\rangle = \hat{M}(l^r) \wedge$

  $\exists_{S', M', e'}.(\tau, \langle r, \texttt{ivar}\ x\rangle, S'; M'; e') = GetSingleWriter(\mathbb{T}, \hat{\tau}, \texttt{ivar}\ x) \wedge$

  $\langle r, i_1\rangle = \hat{M'}(l^r) \wedge$

  $(IsVal(e') \Rightarrow \tau; \langle r, i_1\rangle; S' \vdash_{ew} \langle r, i_2\rangle))$

  By the well-formedness of the store given in the premise of this lemma, the above already holds for all locations in the location environment M. The obligation discharges by inspecting the only new location in M', namely l^r, which is fresh and therefore cannot be in the domain of Σ.

- Case (WF 3.2.5.2;2): $C' \vdash_{wf_{cfc}}; S'$

  Of the requirements for this judgement, the only one that is not satisfied immediately by the well-formedness of the store given in the premise of the lemma is requirement WF 3.2.5.3;3. The specific requirement is to establish that:

  $(l^r \mapsto \texttt{after}\ \tau@{l'}^r) \in C \Rightarrow$

  $(\langle r, i_1\rangle = \hat{M}({l'}^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle \wedge \langle r, i_2\rangle = \hat{M}(l^r)) \oplus$

  $(\langle r, \texttt{ivar}\ x_1\rangle = \hat{M}({l'}^r) \wedge (l^r \mapsto \langle r, \&(r_2, 0)\rangle) \in M \wedge \{\, r_2 \mapsto h \,\} \in S) \oplus$

  $(\langle r, i_1\rangle = \hat{M}({l'}^r) \wedge \tau; \langle r, i_1\rangle; S \vdash_{ew} \langle r, i_2\rangle \wedge (l^r \mapsto \langle r, \&(r_2, 0)\rangle) \in M \wedge S(r)(i_2) = \&(r_2, 0))$

  The second disjunct follows immediately by inversion on D-LetLoc-After-NewReg.

- Case (WF 3.2.5.2;3): $A'; N' \vdash_{wf_{ca}} M'; S'$

  The individual requirements, labeled WF 3.2.5.4;1 - WF 3.2.5.4;4, are handled by the following case analysis.

- Case Case (WF 3.2.5.4;1):

  $((r \mapsto l^r) \in A' \wedge l^r \in N') \Rightarrow$

  $((l^r \mapsto \langle r, i\rangle) \in M' \wedge i > MaxIdx(r, S)) \oplus (l^r \mapsto \&(r_2, i_2)) \in M'$

  By the well-formedness of the store given in the premise of this lemma, the above already holds for locations in the environments A and N. The obligation discharges by inspecting the only new location added to environments, namely l^r. The second disjunct, namely $(l^r \mapsto \&(r_2, i_2)) \in M'$, follows immediately by inversion on D-LetLoc-After-NewReg.

- Case (WF 3.2.5.4;2)

  $((r \mapsto l^r) \in A \wedge \langle r, i_s\rangle = \hat{M}(l^r) \wedge l^r \notin N \wedge \tau; \langle r, i_s\rangle; S \vdash_{ew} \langle r, i_e\rangle) \Rightarrow i_e > MaxIdx(r, S)$

  By inversion on T-LetLoc-After, $l^r \in N'$, and thus this case discharges immediately.

- Case (WF 3.2.5.4;3):

  $l^r \in N' \Rightarrow \langle r, i\rangle = \hat{M'}(l^r) \wedge (r \mapsto (i \mapsto K)) \notin S'$

  Both conjuncts discharge immediately by inversion on D-LetLoc-After-NewReg. Since $(l^r \mapsto \&(r', 0)) \in M'$, we obtain $\langle r', 0\rangle = \hat{M'}$, thus satisfying the first obligation. And since r' is fresh, $(r' \mapsto (0 \mapsto K)) \notin S'$ holds as well, thus discharging this case.

- Case (WF 3.2.5.4;4): $(r \mapsto \emptyset) \in A \Rightarrow (r \mapsto \emptyset) \in S$

  This case discharges because, from the premise of the lemma, this property holds for the original environment A and store S, and, by inversion on T-LetLoc-After, continues to hold for A' and S'.

- Case (WF 3.2.5.2;4): $dom(\Sigma) \cap N = \emptyset$

  Because it is a bound location, $l \notin dom(\Sigma)$, and by inversion on T-LetLoc-After $l \in N'$, which discharges this obligation.

Case:

The cases for D-Let-Expr, D-Let-Val, D-LetRegion, D-LetLoc-Tag, D-LetLoc-Start, D-LetLoc-After, and D-App are similar to the proof of preservation for sequential LoCal given in [**65**].

■

Chaitanya Sunil Koparkar

Luddy Hall, Indiana University, Bloomington, IN
Email: ckoparkar@gmail.com GitHub: github.com/ckoparkar

Research Interests

Programming language design and implementation, program optimization, parallelism and compilers. My research focuses on developing programming language constructs and execution strategies that enable a compiler to improve the run time performance of a program in an *automatic* and *safe* manner.

Education

- **Indiana University**, Luddy School of Informatics, Computing, and Engineering, 2017–2023
  - Ph.D. Computer Science, June 2023.
    Dissertation: *Mostly-serialized Data Structures for Parallel and General-purpose Programming*.
- **Indiana University**, Luddy School of Informatics, Computing, and Engineering, 2016–2023
  - M.S. Computer Science, February 2023.
- **University of Mumbai, India**, 2010–2014
  - B.E. Information Technology, May 2014.

Employment History

- **The MathWorks**, Natick, MA
  - *Senior Software Engineer*, June 2023–present.
  - As part of the Semantic Driven Optimization team, I work on optimizing the Simulink compiler.
- **Indiana University**, Bloomington, IN
  - *Research Assistant*, Newton Lab, August 2017–May 2023.
  - I do research at the intersection of programming language design and program optimization. My primary project has explored a new approach to one of the fundamental tasks in computing—how to represent data in memory. By eschewing the traditional pointer-based approaches, our Gibbon compiler can pack data into a compact, pointer-free form in memory that jells with modern CPU architectures, producing significant performance improvements. Most recently, I worked on reconciling this dense memory representation with parallel execution, allowing programs to obtain the benefits of both. Since then, I've been developing an efficient garbage collector for Gibbon.
- **Cloudseal (acquired by Meta)**, Bloomington, IN
  - *SDE Intern*, May–August 2019.
  - As an intern in a very early stage startup, I helped in implementing Cloudseal's reproducible containers technology. A video demonstrating some features of the product is available on YouTube (link). All of Cloudseal's codebase was written in Rust.
- **Amazon**, Seattle, WA
  - *SDE Intern*, May–July 2017.
  - As an intern in the Retail Systems division, I designed and implemented a new shopping experience (UI/UX & server API's) for creating digital dash buttons.

- **Helpshift**, Pune, India
  - *Software Artisan*, February–June 2016.
  - As a member of the backend team, I worked on different parts of a SOA architecture, developed using Clojure. I wrote a library to distribute thunks of 'work' over a cluster of nodes, which was primarily used for distributed load testing, like Tsung. I was also one of the maintainers of a publish-subscribe messaging system which handled live updates and notifications across millions of devices.
- **TinyOwl (acquired by Zomato)**, Mumbai, India
  - *Software Developer*, April 2015–January 2016.
  - As a member of the backend team, I worked on various things – developing server API's, handling app deployments etc. I was part of a team that implemented a restaurant recommendation engine which increased the overall daily orders by 26%. I also wrote a web service in Go which served as a low latency ingress point for the data analytics pipeline.
- **Canopy Cloud – Atos**, Mumbai, India
  - *Software Developer*, June 2014–April 2015.
  - Developed and deployed CloudFabric, a PAAS offering powered by Cloud Foundry.

Conference Publications

- **Chaitanya Koparkar**, Laith Sakka, Michael Vollmer, Vidush Singhal, Sam Tobin-Hochstadt, Ryan R. Newton, Milind Kulkarni.
  Deforestation for *Some* Non-Linear Functions.
  Unpublished draft.
- **Chaitanya Koparkar**, Vidush Singhal, Aditya Gupta, Mike Rainey, Michael Vollmer, Sam Tobin-Hochstadt, Milind Kulkarni, Ryan R. Newton.
  Garbage Collection for Mostly-Serialized Heaps.
  Under submission.
- **Chaitanya Koparkar**, Mike Rainey, Michael Vollmer, Milind Kulkarni, Ryan R. Newton.
  Efficient Tree-Traversals: Reconciling Parallelism and Dense Representations (PDF).
  ICFP 2021 (33%).
- Michael Vollmer, **Chaitanya Koparkar**, Mike Rainey, Laith Sakka, Milind Kulkarni, Ryan R. Newton.
  LoCal: A Language for Programs Operating on Serialized Data (PDF).
  PLDI 2019 (27%).
- Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, **Chaitanya Koparkar**, Milind Kulkarni, Sam Tobin-Hochstadt, Ryan R. Newton.
  Compiling Tree Transforms to Operate on Packed Representations (PDF).
  ECOOP 2017 (33%).

Technical Reports

- Chaitanya Koparkar.
  Efficient Data Representation Using FlatBuffers (PDF).
  ACM XRDS, 2023.
- Chaitanya Koparkar.
  A primer on pointer tagging (PDF).
  ACM XRDS, 2023.
- Chaitanya Koparkar.
  Making GHC whole again or, how to perform whole-program analysis within GHC (PDF).
  ACM XRDS, 2022.

- **Chaitanya Koparkar**, Mike Rainey, Michael Vollmer, Milind Kulkarni and Ryan R. Newton.
  Extended Version of "Efficient Tree-Traversals: Reconciling Parallelism and Dense Representations" (PDF).
  arXiv, 2021.
- Michael Vollmer, **Chaitanya Koparkar**, Mike Rainey, Laith Sakka, Milind Kulkarni, Ryan R. Newton.
  Extended Version of "LoCal: A Language for Programs Operating on Serialized Data" (PDF).
  Indiana University, 2019

Talks

- *Parallel Gibbon: High performance Haskell compilation!*
  - Invited talk, Grosser Lab at University of Edinburgh, Virtual. October 11, 2021.
  - Invited talk, JetBrains Research Lab at St. Petersburg State University, Virtual. October 18, 2021.
  - Invited talk, PurPL Seminar at Purdue University, Virtual. October 29, 2021.
- *Efficient Tree-Traversals: Reconciling Parallelism and Dense Representations.*
  - ICFP 2021, Virtual. August 23, 2021.
  - SIGPLAN papers track at SPLASH, Chicago, IL. October 22, 2021.
- *Parallelism in (mostly) Serialized Heaps.*
  - IFL, Virtual. September 2, 2020.
- *An Efficient Compiler for Recursive Functions on Mostly Serialized Data.*
  - FHPC, St. Louis, MO. September 29, 2018.
  - PL Wonks, Indiana University, Bloomington, IN. September 21, 2018.

Teaching

- **Associate Instructor, Indiana University**
  - Spring 2023: CSCI P-536 Advanced Operating Systems.
  - Fall 2022: CSCI B-629 Topics in PL: Advanced Functional Programming.
  - Fall 2022: CSCI P-523 Compilers.
  - Spring 2022: CSCI P-536 Advanced Operating Systems.
  - Fall 2021: CSCI B-544 Security For Networked Systems.
  - Spring 2018: CSCI B-629 Topics in PL: Dependent Types.

Service

- Department Editor: Hello World, ACM XRDS. 2022-present.
- Program Committees:
  - ECOOP 2022, external review committee.
- Artifact Evaluation Committees:
  - ECOOP 2022.
  - ESOP 2022.
  - POPL 2021.
  - ECOOP 2021.
  - PLDI 2021.

- PLDI 2020.
- ICFP 2018.

Technical Skills

Proficient in Haskell and C. Productive with C++, Rust, Standard ML, OCaml, and Nix. Familiar with many others.