Language Support for Dynamic, Hierarchical Data Partitioning (Extended Version)

Sean Treichler Michael Bauer Alex Aiken
Stanford University
{sjt,mebauer,aiken}@cs.stanford.edu

Abstract

Applications written for distributed-memory parallel architectures must partition their data to enable parallel execution. As memory hierarchies become deeper, it is increasingly necessary that the data partitioning also be hierarchical to match. Current language proposals perform this hierarchical partitioning statically, which excludes many important applications where the appropriate partitioning is itself data dependent and so must be computed dynamically. We describe Legion, a region-based programming system, where each region may be *partitioned* into subregions. Partitions are computed dynamically and are fully programmable. The division of data need not be disjoint and subregions of a region may overlap, or *alias* one another. Computations use regions with certain *privileges* (e.g., expressing that a computation uses a region read-only) and data *coherence* (e.g., expressing that the computation need only be atomic with respect to other operations on the region), which can be controlled on a per-region (or subregion) basis.

We present the novel aspects of the Legion design, in particular the combination of static and dynamic checks used to enforce soundness. We give an extended example illustrating how Legion can express computations with dynamically determined relationships between computations and data partitions. We prove the soundness of Legion's type system, and show Legion type checking improves performance by up to 71% by eliding provably safe memory checks. In particular, we show that the dynamic checks to detect aliasing at runtime at the region granularity have negligible overhead. We report results for three real-world applications running on distributed memory machines, achieving up to 62.5X speedup on 96 GPUs on the Keeneland supercomputer.

1. Introduction

In the last decade machine architecture, particularly at the high performance end of the spectrum, has undergone a revolution. The latest supercomputers are now composed of heterogeneous processors and deep memory hierarchies. Current programming systems for these machines have elaborate features for describing parallelism, but few abstractions for describing the organization of data. However, having the data organized correctly within the machine is becoming ever more important. Current supercomputers have at least six levels of memory, most of which are explicitly managed by software; even current commodity desktop and mobile computers have at least five levels.[1] As machines of all scales increase the number of processing cores and quantity of available memory, the latency between system components inevitably increases. For many applications the placement and movement of data is already the dominant performance consideration, particularly in high-end machines, and this problem will only grow more acute as overall transistor counts and latencies in future machines increase while the total power budget remains relatively constant.

To program parallel machines with distributed memory (hierarchically organized or not), data must be partitioned into subsets that are placed in the individual memories. For example, in graph computations it is common to subdivide the graph into subgraphs sized to fit in fast memory close to a processor. Note that the term *partition* does not imply the subdivisions of the data are always disjoint—it is desirable to also allow subdivisions that overlap or *alias*. Continuing with the example, many graph computations require knowledge of the nodes bordering each subgraph. Some of these *ghost nodes* for a particular subgraph may also border other subgraphs. In general, the ghost nodes for different subgraphs often alias.

In machines with more than two levels of explicitly managed memory, data partitioning involves a hierarchy where the initial partitions of the data are themselves further partitioned. Often divide-and-conquer strategies repeatedly subdivide the data so that the finest granularity fits in the smallest, fastest memory closest to a processor where a specific computation can access it, which results in complex communication patterns as coarser and finer sets of data are shuffled up and down the memory hierarchy [10]. Thus, the placement and movement of data, and subsets of data, is a first-

[1] A typical organization is (1) distributed memory across a physical network of nodes; (2) shared RAM on chip; (3) one to three levels of cache for each CPU, some shared, some not; (4) GPU global memory; (5) GPU shared memory; (6) GPU registers. Only the CPU caches are managed by hardware and only the global network is not present in commodity consumer machines.

order programming concern. We adopt a region-based approach that makes these groupings of data explicit in the program: a *logical region* names a set of data, a *subregion* of a logical region names a subset of a logical region's data, and a *partitioning* of a logical region r names a number of (possibly overlapping) subregions of r. We use the term *logical region* (which we sometimes abbreviate to *region*) to emphasize that our language-level regions do not imply a physical layout or placement of the logical region's data in the memory hierarchy. Logical regions are just sets of elements and a subregion is literally a subset of its parent region.[2]

By making the groupings of data into regions explicit, it becomes possible for the programmer to express properties of the different regions in a program and for the language system to leverage this information for both performance and correctness in ways that would be difficult to infer without the programmer's guidance. In addition to partitioning regions into subregions, we focus on three properties that Legion programmers can express about regions:

- *Privileges*. Computations have privileges specifying how they can use regions: *read-only*, *read-write*, and *reduce*. More computations can execute in parallel using privileges than without. For example, regions that alias can still be accessed simultaneously by multiple parallel computations provided that the computations all access the regions with read-only privileges, or all access the regions to perform reductions using the same associative and commutative reduction operator.
- *Coherence*. Computations are written in a sequential program order. By default all computations access regions with *exclusive* coherence, which ensures the computations appear to execute in the sequential order, permitting parallelism only when computations access disjoint regions or have non-interfering privileges. However, computations can also request relaxed coherence modes *atomic* and *simultaneous* on regions. Relaxed coherence modes allow reordering and parallel execution of computations that otherwise would execute sequentially due to accessing aliased sets of regions. For example, two computations each requesting atomic coherence on the same region may be re-ordered with respect to the sequential execution order so long as their accesses are serializable. Simultaneous coherence imposes no restrictions on other computations' access to a region; one instance where simultaneous access is useful is when a programmer has implemented his own, higher-level synchronization mechanism.
- *Aliasing*. As outlined above, regions can be partitioned into subregions that may be disjoint or may overlap. Detecting region aliasing is necessary to identify computations that can run in parallel. A central insight of our approach is that detecting region aliasing is both easy and inexpensive when done dynamically at the granularity of logical regions instead of individual memory locations.

Previous work on hierarchically partitioned data has focused on fully static approaches with no runtime overhead. A key feature of these systems is that they disallow all aliasing to make their static analyses tractable. Two recent examples, Sequoia [10] and Deterministic Parallel Java (DPJ) [4], each provide a mechanism to statically partition the heap into a tree of collections of data. The two designs are different in many aspects, but agree that there is a single tree-shaped partitioning of data that must be checked statically (see Section 10). Both approaches also include a system of privileges, but have either no or limited coherence systems.

Our own experience writing high-performance applications in Sequoia [10] as well as in the current industry standard mix of MPI, shared-memory threads, and CUDA has taught us that a fully static system is insufficient. In many cases, the best way to partition data is a function of the data itself—the partitions must be dynamically computed and cannot be statically described. Furthermore, applications often need multiple, simultaneous partitions of the same data—a single partitioning is not enough. Because data partitioning is at the center of what these applications do, shifting from fully static partitions to partitions computed at runtime affects all aspects of the programming model, and in particular the interactions between aliasing, privileges, and coherence. The challenge is to design a system that is both semantically sound and flexible in handling partitions, privileges and coherence with minimal runtime overhead.

In this paper, we present static and dynamic semantics for Legion [2], a parallel programming model that supports multiple, dynamic data partitions and is able to efficiently reason about aliasing, privileges, and coherence. Specifically:

- Legion's logical regions are first-class values and may be dynamically allocated and stored in data structures.
- Logical regions can be dynamically partitioned into *subregions*; partitions are fully programmable.
- A logical region may be dynamically partitioned in multiple different ways; subregions from multiple partitions may include the same data.
- For each computation, *privileges* and *coherence* modes are specified on a per-region basis, giving the programmer fine-grained control over how data is accessed.

We make the following specific contributions:

- We present a type system for *Core Legion* programs that statically verifies the safety of individual pointers and region privileges at call boundaries (Section 4).

[2] A separate system of *physical regions* hold concrete copies of the data of logical regions at run-time. Physical regions have a specific data layout and live in a specific memory. The Legion run-time system may maintain multiple physical copies of a single logical region for performance reasons; for example, read-only may be replicated in multiple physical regions to put it closer to the computations that use it.

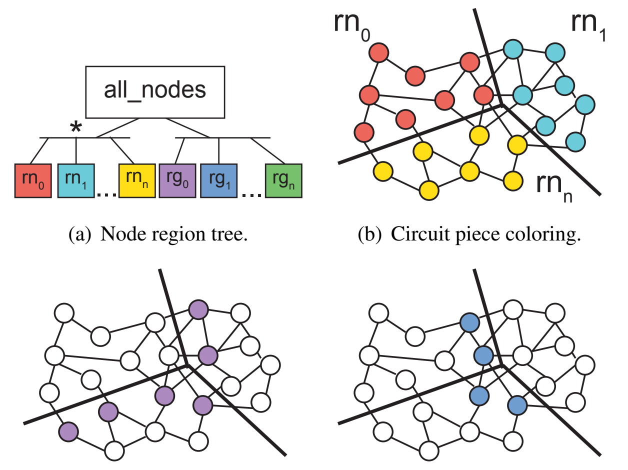
- We present a novel parallel operational semantics for Core Legion. This semantics is compositional, hierarchical, and asynchronous, reflecting the way such programs actually execute on the hardware (Section 5.3).
- We prove the soundness of Legion's static type and privilege system (Section 6). In particular, we show that Legion's very liberal dynamic manipulations of regions can be handled with a combination of static and inexpensive dynamic checks.
- Using the soundness of the type system, we show that if expressions e_1 and e_2 are *non-interfering* (can be executed in parallel), then subexpressions e_1' of e_1 and e_2' of e_2 are also non-interfering (Section 8). This result is the basis for Legion's hierarchical, distributed scheduler, which is crucial for high performance on the target class of machines. We note that no other parallel language or runtime system currently supports distributed scheduling.
- We give experimental evidence that supports the Legion design choices. On three real-world applications, we show that dynamic region pointer checks would be expensive, justifying checking this aspect of the type system statically. We also show that the cost of region aliasing checks is low, showing that an expressive and dynamic language with aliasing is compatible with both high performance and safety (Section 9).

2. Circuit Example

We begin by introducing a circuit simulation written in the Legion programming model that serves as a running example throughout the remainder of the paper. In this section we describe how the requirements of the simulation motivate the novel features of Legion. Section 3 introduces the Core Legion language by showing examples of code from the circuit simulation.

The circuit simulation takes as input an arbitrary graph of circuit elements (wires and nodes where the wires connect) represented by the two logical regions `all_nodes` and `all_wires`. The simulation iterates for many time steps, performing three computations during each time step: `calc_new_currents`, `distribute_charge`, and finally `update_voltage`. For these computations to be run in parallel, the regions representing the graph must be partitioned into pieces that match the simulation's data access patterns. The choice of partitioning will ultimately dictate performance and is therefore the most important decision in any Legion program.

An ideal partitioning depends on many factors, including the shape of data structures, the input, and the desired number of partitions (which usually varies with the target machine). Due to the multitude of factors that can influence partitioning, a critical design decision made in Legion is to provide a programmable interface whereby the application can compute a partitioning dynamically and communicate



(a) Node region tree. (b) Circuit piece coloring.

(c) Ghost `rg0` coloring. (d) Ghost `rg1` coloring.

Figure 1. Partitions of the all_nodes region.

that partitioning to the Legion runtime system. This design absolves the Legion implementation of the responsibility for computing an ideal partition for all regions across all applications on any potential architecture. Instead, our approach provides the application with direct control over all partitioning decisions that ultimately impact performance.

In Legion, partitioning takes place in two steps. First, the programmer assigns a *color* to each element of the region to be partitioned. The number of colors and how they are assigned to elements can be the result of an arbitrary computation, giving the programmer complete control over the coloring. Second, Legion creates new subregions, one for each color, with each region element assigned to the subregion of the appropriate color. Thus, the programmer expresses the desired partitioning of a region, and Legion provides the mechanism to carry out the programmer's directions.

To efficiently support the circuit simulation's access patterns, the region `all_nodes` holding all the nodes of the graph is partitioned in two different ways. The desired *region tree* is shown in Figure 1(a). First, there are subregions of `all_nodes` that describe the set of nodes "owned" by each piece, called `rn0`, `rn1`, ... Since each node is in one piece, this partition is *disjoint*, which is indicated by a $*$ on the left subtree. Figure 1(b) shows one possible partitioning along with the necessary coloring to generate the disjoint partition in Figure 1(a). Second, each piece of the circuit needs access to the ghost nodes on its border. The ghost nodes for two circuit pieces are shown in Figures 1(c) and 1(d); note that two nodes are in both sets. Because a node may neighbor more than one other circuit piece, this second partition of `all_nodes` is *aliased*. Thus, there are two sources of aliasing in the region tree: the two distinct partitions divide the `all_nodes` region in different ways, and the ghost node subregions are not disjoint.

There are two alternative approaches to using multiple partitions for the circuit simulation, both of which avoid

T	::=		types
	|	bool | int	base types
	|	$\langle T_1, \ldots, T_n \rangle$	tuple
	|	$T@(r_1, \ldots, r_n)$	pointer
	|	coloring(r)	region coloring
	|	$\exists r_1, \ldots, r_n.T$ where Ω	region relationship
	|	$\forall r_1, \ldots, r_n.(T_1, \ldots, T_n), \Phi, Q \to T_r$	functions
Ω	::=	$\{\omega_1, \ldots, \omega_n\}$	region constraints
ω	::=	$r_1 \le r_2$	subregion
	|	$r_1 * r_2$	disjointness
Φ	::=	$\{\phi_1, \ldots, \phi_n\}$	privileges
ϕ	::=	reads(r) | writes(r) | reduces$_{id}(r)$	
Q	::=	$\{q_1, \ldots, q_n\}$	coherence modes
q	::=	atomic(r) | simult(r)	
v	::=		values
	|	bv | iv	base values
	|	$\langle v_1, v_2 \rangle$	tuple
	|	null | l	memory location
	|	$\{(l, iv), \ldots\}$	coloring
	|	$\langle\langle \rho_1, \ldots, \rho_n, v \rangle\rangle$	reg. relation instance

bv	::=	false | true	
iv	::=	0 | 1 ...	
e	::=		expressions
	|	bv | iv	constants
	|	$\langle e_1, \ldots, e_n \rangle$ | $e.1$ | $e.2$ | ...	tuple
	|	id	
	|	new $T@r$ | null $T@r$ | isnull(e)	
	|	upregion$(e, r_1, \ldots, r_n)$	
	|	downregion$(e, r_1, \ldots, r_n)$	
	|	read(e_1) | write(e_1, e_2)	memory access
	|	reduce(id, e_1, e_2)	
	|	newcolor r | color(e_1, e_2, e_3)	coloring
	|	$e_1 + e_2$	integer ops
	|	$e_1 < e_2$	comparisons
	|	let $id : T = e_1$ in e_2	
	|	if e_1 then e_2 else e_3	
	|	$id[r_1, \ldots, r_n](e_1, \ldots, e_n)$	function calls
	|	partition r_p using e_1 as $r_1, \ldots, r_n$ in e_2	
	|	pack e_1 as $T[r_1, \ldots, r_n]$	
	|	unpack e_1 as $id : T[r_1, \ldots, r_n]$ in e_2	

Figure 2. Core Legion

introducing aliasing. We could create a single partition with 2^n subregions, one for each possible case of sharing, or computations on each piece could use the `all_nodes` region to access their ghost nodes. Neither option is attractive: the former significantly complicates programming and the latter greatly overestimates the required ghost nodes, increasing runtime data movement as well as limiting parallelism.

For simplicity this example has only one level of partitioning (although in two different ways). All the semantic issues that concern the results of this paper can be illustrated with one level of partitioning. In general, however, the region tree can have many levels, as subregions are themselves partitioned, perhaps also in multiple ways. Typically, the number of levels and size of partitions depends on both the data and the memory hierarchy of the target machine, allowing regions to be placed in levels of memory where they fit [2].

Because data is partitioned dynamically in arbitrary ways and because these partitions may not be disjoint, parallelism is necessarily detected dynamically in Legion. Functions that the Legion runtime considers for parallel execution are called *tasks*. Tasks are required to specify the regions that they access as well as the task's privileges and coherence modes on each region; the type system introduced in Section 4 verifies that Legion tasks abide by their declared region access privileges. The partitioning of the data, task region privileges, and task coherence modes all contribute to determining which tasks can be executed in parallel.

The Legion task scheduler considers task calls in sequential program execution order. If a task's region accesses do not conflict with a previously issued task, the task can be launched as a parallel task, otherwise it is serialized with all of its conflicting tasks. One of our main results is a sufficient condition for deciding that two tasks do not *interfere* on their region arguments and can be executed in parallel (Section 7). Subtasks may also be launched within tasks, giving nested parallelism. A second result allows even the scheduling decisions to be made in parallel, so that scheduling does not become a serial bottleneck (Section 8).

3. Core Legion

In this paper, we work with Core Legion, a subset of the full Legion language introduced in [2]. Although equally expressive, Core Legion trades programmer convenience for a reduction in the number of constructs, simplifying the proofs that follow. We illustrate Core Legion programming through snippets from the circuit simulation. The full Core Legion program for the circuit simulation is in Appendix A. (Line numbers in the code snippets can be used to locate them in the full program.)

Figure 2 defines Core Legion syntax. The basic types include booleans, integers, tuples, and pointers. In addition to specifying the type of the value they point to, pointer types in Legion are annotated with one or more logical regions; any non-null pointer value must point to a location that is contained in at least one of the regions. Pointers are created by using the `new` expression to allocate space within a specified region and may be tested for validity with the `isnull` expression.

To help address the proliferation of types that vary only in their regions, the Core Legion compiler supports type declarations parameterized on logical region names, which are expanded into the syntax of Figure 2 before any analysis is performed. For clarity and conciseness we present the examples using parameterized types, but we omit the translation step to monomorphic Core Legion types, which is completely standard.

The following code snippet shows the types used to describe the nodes and wires in a circuit. The `CircuitWire` is parameterized on two regions, with `rn` intended to be the region of nodes owned by a piece of the circuit, and `rg` the region of that piece's ghost nodes. An edge has two node endpoints, one of which is in the piece and the other which may be either in the piece or a ghost node—i.e., the edge is either entirely within the piece or crosses a boundary into another piece.

```
−−                    ⟨voltage,current,charge,capacitance,piece ID⟩
type CircuitNode      = ⟨int,int,int,int,int⟩
−−                    ⟨ owned node, owned or ghost node, resistance, current⟩
type CircuitWire⟨rn,rg⟩ = ⟨CircuitNode@rn, CircuitNode@(rn,rg),int,int⟩
```

Core Legion is an expression language, using `let` expressions to define local variables. Pointers are manipulated using explicit `read`, `write`, and `reduce` expressions as shown here:

```
−− update voltage on a node
let node : CircuitNode = read(node_ptr) in
let voltage : int = (node.3 / node.4) in
let new_node : CircuitNode = ⟨ voltage, node.2, node.3, node.4, node.5 ⟩ in
  write(node_ptr, new_node)
```

As described in Section 2, deciding how to partition regions is left to the application. In the circuit simulation we use METIS[14], a standard graph-partitioning library. Because we need a way to iterate over all the nodes and wires, we define (parameterized) types for lists of nodes and wires and then give a prototype for the actual METIS function:

```
type NodeList⟨rl,rn⟩      = ⟨ CircuitNode@rn, NodeList⟨rl,rn⟩@rl ⟩
type WireList⟨rl,rw,rn,rg⟩= ⟨ CircuitWire⟨rn,rg⟩@rw, WireList⟨rl,rw,rn,rg⟩@rl ⟩
function extern_metis[rl,rn,rw](node_list : NodeList⟨rl,rn⟩@rl,
         wires_list : WireList⟨rl,rw,rn,rn⟩@rl), reads(rl,rn,rw), writes(rn) : bool
```

METIS records how the graph is to be partitioned by annotating each `CircuitNode` with a piece ID. Note that both list types use a second region parameter to allow the spine of the list to be in a region different than the region where the nodes or wires themselves are placed. There are no global region names in Legion, so functions must be region-polymorphic, with all region names used in the function's prototype being implicitly universally quantified. In addition to giving names and types of formal parameters and the type of the return value, a Legion function also declares the necessary access privileges. In this case, all three regions are read by `extern_metis`, but only the `rn` region is written (since it contains the piece IDs). A function can be called only if the caller possesses all the privileges needed by the called function.

Once an application has decided how it wants to partition a region, that information must be provided to the Legion runtime. This is achieved through the use of an object of a special *coloring* type, which maps locations within a specified region to "colors". (Core Legion uses integers for colors.) A coloring is created by the `newcolor` expression, and the mapping is updated by the `color` expression. The following code snippet shows how the coloring for the "owned nodes" partition is generated. Similar code for the ghost nodes partition and wires can be found in Appendix A. The full Legion language includes a *multicoloring* type to conveniently describe aliased partitions. In Core Legion, a multicoloring and the corresponding partitioning operation are implemented by performing a separate coloring and partition for each aliased subregion, which soundly captures the aliased nature of a multicoloring.

```
function owned_node_coloring[rl,rn] ( node_list: NodeList⟨rl,rn⟩@rl ),
        reads(rl,rn) : coloring(rn) =
  if isnull(node_list) then
    newcolor rn
  else                        −− tuple fields accessed by .(field number)
    let list_elem : NodeList⟨rl,rn⟩ = read(node_list) in
    let part_coloring : coloring(rn) = owned_node_coloring[rl,rn](list_elem.2) in
    let node_ptr : CircuitNode@rn = list_elem.1 in
    let node : CircuitNode = read(node_ptr) in
    let piece_id_from_metis: int = node.5 in
      color(part_coloring, node_ptr, piece_id_from_metis)
```

Once a coloring has been created, it may be used in a `partition` expression, which gives local names to the subregions corresponding to each color used.

```
let owned_node_map : coloring(rn) = owned_node_coloring[rl,rn](all_nodes) in
partition rn using owned_node_map as rn0,rn1 in
```

At run time, the partition operation extends the region tree (recall Figure 1(a)) maintained by the Legion runtime; this data structure, which includes all the allocated dynamic regions and their parent-child relationships, is used to decide whether computations can run in parallel based on what regions they access and with what privileges[2]. At compile time, the partition operation introduces constraints into the static type environment describing both the disjointness of subregions (e.g., $\mathtt{rn0} * \mathtt{rn1}$) and the subregion relationships (e.g., $\mathtt{rn0} \leq \mathtt{all\_nodes}$).

Because subregions are entirely included in the original parent region, there is a subtyping-like relationship between a pointer-into-a-subregion and a pointer-into-the-parent-region. However, Core Legion provides no automatic conversions between pointer types. A pointer into a subregion may be "upcast" to a pointer to a parent region via the explicit `upregion` expression, which statically verifies the subregion relationship. The corresponding "downcast" is available via the `downregion` expression, which must perform a run-time check that the pointer does point into the specified subregion. (If it does not, the pointer value is replaced by `null`, which is defined to exist in all regions.)

Regions are first-class entities and may be stored in the heap. This feature is important in many applications; for example, in a simple work list algorithm the work list may be a queue of regions to be processed. When a region is stored into the heap, however, it escapes the scope of the enclosing partition expressions and the region's relationships to other regions (whether it is a subregion or disjoint from another region) are forgotten. To allow these facts to be retained across heap reads and writes of values containing regions, Core Legion has *region relationships*. A region relationship is a bounded existential type that allows a programmer to *pack* one or more regions, a value (whose type may include those regions), and subregion or disjointness constraints, together. The example region relationship below for a `CircuitPiece` involves its region of wires `rpw`, region of nodes `rpn`, region of ghost nodes `rg`, and important constraints. Note that the names of `rpw`, `rpn`, and `rg` are bound in the region relationship, and the knowledge that they are subregions of free region names `rw` and `rn` is captured in the constraints.

```
type CircuitPiece⟨rl,rw,rn⟩ = rr[rpw,rpn,rg]
                    ⟨ WireList⟨rl,rpw,rpn,rg⟩@rl, NodeList⟨rl,rpn⟩@rl ⟩
                    where rpn ≤ rn and rg ≤ rn and rpw ≤ rw and
                        rn ∗ rw and rl ∗ rn and rl ∗ rw
```

The Core Legion type system statically verifies the correctness of region relationships as part of a `pack` expression. The regions and constraints bound in a region relationship can be reintroduced (with fresh names) within the body of an `unpack` expression. In the circuit simulation given in Appendix A, region relationships are mostly a convenience, allowing the programmer to give a name to a collection of regions and constraints that results in simpler function interfaces. However, in a version of the circuit simulation that partitions the graph into many more than two pieces having a data structure that stores all the pieces with their associated ghost regions is essential.

In contrast to disjointness and subregion constraints, region access privileges cannot be captured in a region relationship. A function inherits a subset of the privileges of its caller, and thus privileges belong to functions. This is a key requirement for soundness of the Legion type system that we return to in Section 4. When a function unpacks a region r from a region relationship, no privileges for r itself are granted. To access r, the function must already hold the needed privileges on some region q that is a superset of r (i.e., q is r's parent or another ancestor region), and furthermore there must be constraints in the region relationship that prove $r \leq q$.

The main simulation loop, shown in Listing 1, runs for many time steps, each of which performs three computations: calculate new currents, distribute charges, and update voltages on the circuit. For simplicity, this example is written for a graph that is partitioned into only two pieces. For each time step, the loop (tail recursive function `execute_time_steps`, lines 15-26) unpacks the two previously packed circuit pieces, giving new names to the subregions introduced by each region relationship. The `execute_time_steps` function will have read/write privileges for the newly named regions, such as `rn0`, because it has read/write privileges for `rn` and the `CircuitPiece` region relationship ensures that `rn0 ≤ rn`.

The `execute_time_steps` function illustrates the importance of having different partitions provide multiple

```
1  −− Leaf Task Declarations (implementations in appendix)
2  function calc_new_currents[rl,rw,rn,rg] ( ptr_list : WireList⟨rl,rw,rn,rg⟩@rl ),
3      reads(rl,rw,rn,rg), writes(rw) : bool
4  function distribute_charge[rl,rw,rn,rg] ( ptr_list : WireList⟨rl,rw,rn,rg⟩@rl ),
5      reads(rl,rw,rn), reduces(reduce_charge,rn,rg), atomic(rn,rg) : bool
6  function update_voltage[rl,rn] ( ptr_list : NodeList⟨rl,rn⟩@rl ),
7      reads(rl,rn), writes(rn) : bool
8
9  −− Reduction function for distribute charge
10 function reduce_charge ( node : CircuitNode, current : int ) : CircuitNode
11   let new_charge : int = node.3 + current in
12     ⟨ node.1,node.2,new_charge,node.4⟩
13
14 −− Time Step Loop
15 function execute_time_steps[rl,rw,rn] ( p0 : CircuitPiece⟨rl,rw,rn⟩,
16     p1 : CircuitPiece⟨rl,rw,rn⟩, steps : int ) , reads(rn,rw,rl), writes(rn,rw) : bool =
17   if steps < 1 then true else
18   unpack p0 as piece0 : CircuitPiece⟨rl,rw,rn⟩[rw0,rn0,rg0] in
19   unpack p1 as piece1 : CircuitPiece⟨rl,rw,rn⟩[rw1,rn1,rg1] in
20   let _ : bool = calc_new_currents[rl,rw0,rn0,rg0](piece0.1) in
21   let _ : bool = calc_new_currents[rl,rw1,rn1,rg1](piece1.1) in
22   let _ : bool = distribute_charge[rl,rw0,rn0,rg0](piece0.1) in
23   let _ : bool = distribute_charge[rl,rw1,rn1,rg1](piece1.1) in
24   let _ : bool = update_voltage[rl,rn0](piece0.2) in
25   let _ : bool = update_voltage[rl,rn1](piece1.2) in
26     execute_time_steps[rl,rw,rn](p0,p1,steps−1)
```

Listing 1. Main Simulation Loop

views onto the same logical region. The `calc_new_currents` function uses the owned and ghost regions of a piece, which are from different partitions; no single partition of the nodes describes this access pattern. In `calc_new_currents` these regions only need read privileges, while the only writes are performed to the wires subregion belonging to that piece. Thus, both instances of `calc_new_currents` can be run as parallel tasks. Similarly, the `update_voltage` function (lines 6-7) modifies only the disjoint owned regions, while only reading from regions shared with the other instance; the two instances of `update_voltage` can also run in parallel.

The most interesting function is `distribute_charge` (lines 2-5), which uses a *reduction* privilege for regions `rn` and `rg`. A reduction names the reduction operator (which is assumed to be associative and commutative) as the first component of the privilege. Programmers can write their own reduction operators, such as the function `reduce_charge` in Listing 1. Reductions allow updates to the named regions that are performed with the named reduction operator to be reordered. For example, reductions can be performed locally by a task and only the final results folded in to the destination region. However, by default, functions with no coherence annotation have *exclusive* coherence for their region arguments: reads and writes have the results expected as if the original sequential execution order of the program was preserved, unaffected by any concurrently executing tasks. Thus, to fully exploit reductions it is important to use a relaxed coherence mode, in this case *atomic* coherence, which permits other tasks performing the same reduction operation on the named regions to execute in parallel. The most relaxed coherence mode is *simult*; simultaneous coherence allows concurrent access to the region by all functions that are using the region in a simultaneous mode. The interaction between tasks using the same region with different coherence modes is formalized in Section 7. While associative and commu-

$\Omega \subseteq \Omega^*$
$r_i \leq r_j \in \Omega^* \Rightarrow r_i \leq r_i \in \Omega^* \wedge r_j \leq r_j \in \Omega^*$
$r_i \leq r_j \in \Omega^* \wedge r_j \leq r_k \in \Omega^* \Rightarrow r_i \leq r_k \in \Omega^*$
$r_i \leq r_j \in \Omega^* \wedge r_j * r_k \in \Omega^* \Rightarrow r_i * r_k \in \Omega^*$
$r_i * r_j \in \Omega^* \Rightarrow r_j * r_i \in \Omega^*$

$\Phi \subseteq \Phi^*$
$r_i \leq r_j \in \Omega^* \wedge \text{reads}(r_j) \in \Phi^* \Rightarrow \text{reads}(r_i) \in \Phi^*$
$r_i \leq r_j \in \Omega^* \wedge \text{writes}(r_j) \in \Phi^* \Rightarrow \text{writes}(r_i) \in \Phi^*$
$r_i \leq r_j \in \Omega^* \wedge \text{reduces}_{id}(r_j) \in \Phi^* \Rightarrow \text{reduces}_{id}(r_i) \in \Phi^*$
$\text{reads}(r) \in \Phi^* \wedge \text{writes}(r) \in \Phi^* \Rightarrow \text{reduces}_{id}(r) \in \Phi^*$
for every function identifier id

Figure 3. Privilege and Constraint Closure

tative reductions always produce the same result regardless of execution order, in general relaxed coherence modes introduce non-determinism into Legion programs. This non-determinism is completely under programmer control, at the per-region (or subregion) granularity.

4. Type System

Core Legion is explicitly typed using judgments of the form

$$\Gamma, \Phi, \Omega \vdash e : T$$

Besides a type environment Γ, type judgments include the access privileges Φ for the logical regions in the expression e as well as constraints Ω that must hold between those logical regions.

A representative selection of the type rules is given in Figure 4. Both Φ and Ω are used in the heap access expressions `read`, `write`, and `reduce`. A valid heap access has the needed permission for logical region(s) in the pointer's type. Note the exact region need not be named in Φ if permissions exist for logical regions that provably contain the pointer's region(s). To simplify this check, Figure 3 defines closure operations Ω^* (all constraints implied by Ω) and Φ^* (all privileges implied by Φ and Ω^*).

Region constraints are introduced into Ω by the `partition` expression. The type system constrains the coloring used in a partitioning to only include pointers into the region being partitioned. In Core Legion, the subregions that result from a single partition expression are always disjoint. (As a reminder, the aliased subregions that result from a multicoloring are obtained in Core Legion through multiple nested partition expressions.)

The `pack` expression requires the programmer to explicitly name which regions are expected to satisfy the constraints of the region relationship's type. The programmer also provides the new names for regions that result from an `unpack`, with the constraint that fresh names are chosen.

Finally, the type checking rule for the overall Legion program shows how each function is type-checked separately, with no global variables or region constraints. Although the coherence modes (Q_i) are part of a function's prototype, they influence only the runtime behavior, not the type checking.

5. Operational Semantics

Operational semantics for parallel languages are traditionally constructed using small-step semantics. The state of the system includes the current state of each concurrent computation and a small step allows one of two things to happen: either a single computation makes progress or a subset of the computations rendezvous on an explicit synchronization primitive (e.g., a matching send and receive on a channel). Although an operational semantics for Legion can be constructed in such a manner, it is not natural, and certainly Legion programmers do not think about the execution of Legion programs in this way. Nested parallelism (subtasks recursively launching other subtasks) and the absence of explicit synchronization constructs encourage programmers to think about programs compositionally, as the execution of child tasks in the context of a parent task.

To formalize this view of Core Legion executions, we express the operational semantics in a big-step style, which captures the hierarchical nature of Legion tasks. In addition to being arguably more intuitive to someone trying to understand Legion runtime behavior, the preservation of the task hierarchy in our semantics makes it considerably easier to prove the soundness of the type system and the safety of our hierarchical scheduling algorithms. Finally, a big-step semantics simplifies the explanation of Legion's novel treatment of coherence.

Core Legion's operational semantics rules have the form

$$M, L, H, S, C \vdash e \mapsto v, E$$

and specify that the evaluation of expression e yields a value v. The environment includes the standard mapping L of local variables to their values, and an immutable heap typing H assigning types to heap locations. An additional mapping M is used to translate logical regions r_i to physical regions ρ_i, which are sets of concrete memory locations. M is extended in the standard way to map types, environments, and constraints that refer to logical regions into corresponding structures that refer to physical regions. For example, $M[\![int@r_1]\!] = int@\rho_1$.

The two unusual components of the operational semantics are the *dynamic memory trace* E and the *clobber set* C. As these are the key to making the Core Legion semantics composable, we discuss them in detail in the following two subsections.

5.1 Dynamic Memory Traces

In a sequential big-step semantics for a language with side effects, evaluation commonly begins in an initial store S and produces a value v and a final store S'. In our Core Legion semantics, instead of a final store, an explicit list of all memory operations (i.e. reads, writes, reductions) is returned

$$\frac{\begin{array}{l}\Gamma,\Phi,\Omega \vdash e_1 : T@(r_1,\ldots,r_n)\\ \forall i.\ \text{reads}(r_i) \in \Phi^*\end{array}}{\Gamma,\Phi,\Omega \vdash \text{read}(e_1) : T} \quad \text{[T-Read]}$$

$$\frac{\begin{array}{l}\Gamma,\Phi,\Omega \vdash e_1 : T@(r_1,\ldots,r_n)\\ \Gamma,\Phi,\Omega \vdash e_2 : T\\ \forall i.\ \text{writes}(r_i) \in \Phi^*\end{array}}{\Gamma,\Phi,\Omega \vdash \text{write}(e_1,e_2) : T@(r_1,\ldots,r_n)} \quad \text{[T-Write]}$$

$$\frac{\begin{array}{l}\Gamma(id) = (T_1,T_2),\emptyset,\emptyset \to T_1\\ \Gamma,\Phi,\Omega \vdash e_1 : T_1@(r_1,\ldots,r_n)\\ \Gamma,\Phi,\Omega \vdash e_2 : T_2\\ \forall i.\ \text{reduces}_{id}(r_i) \in \Phi^*\end{array}}{\Gamma,\Phi,\Omega \vdash \text{reduce}(id,e_1,e_2) : T_1@(r_1,\ldots,r_n)} \quad \text{[T-Reduce]}$$

$$\Gamma,\Phi,\Omega \vdash \text{new } T@r : T@r \quad \text{[T-New]}$$

$$\frac{\begin{array}{l}\Gamma,\Phi,\Omega \vdash e : T@(r'_1,\ldots r'_k)\\ \forall i.\exists j, r'_i \leq r_j \in \Omega^*\end{array}}{\Gamma,\Phi,\Omega \vdash \text{upregion}(e,r_1,\ldots,r_n) : T@(r_1,\ldots,r_n)} \quad \text{[T-UpRgn]}$$

$$\frac{\Gamma,\Phi,\Omega \vdash e : T@(r'_1,\ldots r'_k)}{\Gamma,\Phi,\Omega \vdash \text{downregion}(e,r_1,\ldots,r_n) : T@(r_1,\ldots,r_n)} \quad \text{[T-DnRgn]}$$

$$\Gamma,\Phi,\Omega \vdash \text{newcolor } r : \text{coloring}(r) \quad \text{[T-NewColor]}$$

$$\frac{\begin{array}{l}\Gamma,\Phi,\Omega \vdash e_1 : \text{coloring}(r)\\ \Gamma,\Phi,\Omega \vdash e_2 : T@r\\ \Gamma,\Phi,\Omega \vdash e_3 : \text{int}\end{array}}{\Gamma,\Phi,\Omega \vdash \text{color}(e_1,e_2,e_3) : \text{coloring}(r)} \quad \text{[T-Color]}$$

$$\frac{\begin{array}{l}\Gamma,\Phi,\Omega \vdash e_1 : \text{coloring}(r_p)\\ \Omega' = \Omega \wedge \bigwedge_{i\in[1,k]} r_i \leq r_p \wedge \bigwedge_{1\leq i<j\leq k} r_i * r_j\\ \Gamma,\Phi,\Omega' \vdash e_2 : T\\ \{r_1,\ldots,r_k\} \cap \mathit{regions\_of}(\Gamma,T) = \emptyset\end{array}}{\Gamma,\Phi,\Omega \vdash \text{partition } r_p \text{ using } e_1 \text{ as } r_1,\ldots,r_k \text{ in } e_2 : T} \quad \text{[T-Partition]}$$

$$\frac{\begin{array}{l}T_1 = \exists r'_1,\ldots r'_k.\ T_2 \text{ where } \Omega_1\\ \Omega_1[r_1/r'_1,\ldots,r_k/r'_k] \subseteq \Omega^*\\ \Gamma,\Phi,\Omega \vdash e_1 : T_2[r_1/r'_1,\ldots,r_k/r'_k]\end{array}}{\Gamma,\Phi,\Omega \vdash \text{pack } e_1 \text{ as } T_1[r_1,\ldots,r_k] : T_1} \quad \text{[T-Pack]}$$

$$\frac{\begin{array}{l}T_1 = \exists r'_1,\ldots,r'_k.\ T_2 \text{ where } \Omega_1\\ \Gamma,\Phi,\Omega \vdash e_1 : T_1\\ \Gamma' = \Gamma[T_2[r_1/r'_1,\ldots,r_k/r'_k]/id]\\ \Omega' = \Omega \cup \Omega_1[r_1/r'_1,\ldots,r_k/r'_k]\\ \Gamma',\Phi,\Omega' \vdash e_2 : T_3\\ \{r_1,\ldots,r_k\} \cap \mathit{regions\_of}(\Gamma,T_1,T_3) = \emptyset\end{array}}{\Gamma,\Phi,\Omega \vdash \text{unpack } e_1 \text{ as } id : T_1[r_1,\ldots,r_k] \text{ in } e_2 : T_3} \quad \text{[T-Unpack]}$$

$$\frac{\begin{array}{l}\Gamma(id) = \forall r'_1,\ldots r'_k.(T_1,\ldots,T_n),\Phi',Q' \to T_r\\ \Gamma,\Phi,\Omega \vdash e_i : T_i[r_1/r'_1,\ldots,r_k/r'_k]\\ \Phi'[r_1/r'_1,\ldots,r_k/r'_k] \subseteq \Phi^*\end{array}}{\Gamma,\Phi,\Omega \vdash id[r_1,\ldots,r_k](e_1,\ldots,e_n) : T_r[r_1/r'_1,\ldots,r_k/r'_k]} \quad \text{[T-Call]}$$

$$\frac{\begin{array}{l}\text{for } 1 \leq i \leq p,\\ \quad \Gamma(id_i) = \forall r^i_1,\ldots r^i_{k_i}.(T^i_1,\ldots,T^i_{n_i}),\Phi^i,Q^i \to T^i_r\\ \quad \Gamma^i = \Gamma[a^i_1/T^i_1,\ldots,a^i_{n_i}/T^i_{n_i}]\\ \quad \Gamma^i,\Phi^i,\emptyset \vdash e^i : T^i_r\end{array}}{\begin{array}{l}\vdash \{\text{function } id_1[r^1_1,...,r^1_{k_1}](a^1_1 : T^1_1,...,a^1_{n_1} : T^1_{n_1}),\Phi^1,Q^1 : T^1_r : e^1,\\ \quad \ldots\\ \quad \text{function } id_p[r^p_1,...,r^p_{k_p}](a^p_1 : T^p_1,...,a^p_{n_p} : T^p_{n_p}),\Phi^p,Q^p : T^p_r : e^p\} : \bullet\end{array}} \quad \text{[T-Program]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e \mapsto l,E \qquad S' = \mathit{apply}(S,E)\\ v = \begin{cases} S'(l), & \text{if} l \notin C\\ v' : H(l), & \text{otherwise}\end{cases}\end{array}}{M,L,H,S,C \vdash \text{read}(e) \mapsto v, E +\!\!+ [\mathit{read}(l,\mathit{excl},v,0)]} \quad \text{[E-Read]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e_1 \mapsto l,E_1 \qquad S' = \mathit{apply}(S,E_1)\\ M,L,H,S',C \vdash e_2 \mapsto v,E_2 \qquad \mathit{valid\_interleave}(S,C,E',E_1,E_2)\end{array}}{M,L,H,S,C \vdash \text{write}(e_1,e_2) \mapsto l, E' +\!\!+ [\mathit{write}(l,\mathit{excl},v,0)]} \quad \text{[E-Write]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e_1 \mapsto l,E_1 \qquad S' = \mathit{apply}(S,E_1)\\ M,L,H,S',C \vdash e_2 \mapsto v,E_2 \qquad \mathit{valid\_interleave}(S,C,E',E_1,E_2)\end{array}}{M,L,H,S,C \vdash \text{reduce}(id,e_1,e_2) \mapsto l, E' +\!\!+ [\mathit{reduce}_{id}(l,\mathit{excl},v,0)]} \quad \text{[E-Reduce]}$$

$$\frac{l \in M(r) \qquad l \notin \text{domain}(S) \qquad H(l) = M[\![T]\!]}{M,L,H,S,C \vdash \text{new } T@r \mapsto l, []} \quad \text{[E-New]}$$

$$\frac{M,L,H,S,C \vdash e \mapsto v,E}{M,L,H,S,C \vdash \text{upregion}(e,r_1,\ldots,r_n) \mapsto v,E} \quad \text{[E-UpRgn]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e \mapsto l,E\\ v = \begin{cases} l, & \text{if } \exists i, l \in M(r_i).\\ \mathit{null}, & \text{otherwise.}\end{cases}\end{array}}{M,L,H,S,C \vdash \text{downregion}(e,r_1,\ldots,r_n) \mapsto v,E} \quad \text{[E-DnRgn]}$$

$$\frac{\begin{array}{l}K = \{(l_1,iv_1),\ldots,(l_p,iv_p)\}, \text{ where}\\ (\forall i \in [1,p].l_i \in M(r)) \wedge (\forall i,j \in [1,p].l_i \neq l_j)\end{array}}{M,L,H,S,C \vdash \text{newcolor } r \mapsto K, []} \quad \text{[E-NewColor]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e_1 \mapsto K,E_1 \qquad S' = \mathit{apply}(S,E_1)\\ M,L,H,S',C \vdash e_2 \mapsto l,E_2 \qquad S'' = \mathit{apply}(S',E_2)\\ M,L,H,S'',C \vdash e_3 \mapsto v,E_3\\ K' = \{(l,v)\} \cup \{(l_i,v_i) : (l_i,v_i) \in K \wedge l \neq l_i\}\\ \mathit{valid\_interleave}(S,C,E',E_1,E_2,E_3)\end{array}}{M,L,H,S,C \vdash \text{color}(e_1,e_2,e_3) \mapsto K',E'} \quad \text{[E-Color]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e_1 \mapsto K,E_1 \qquad \rho_i = \{l : (l,i) \in K\}, \text{for} 1 \leq i \leq k\\ M' = M[\rho_1/r_1,\ldots,\rho_k/r_k] \qquad S' = \mathit{apply}(S,E_1)\\ M',L,H,S',C \vdash e_2 \mapsto v,E_2 \qquad \mathit{valid\_interleave}(S,C,E',E_1,E_2)\end{array}}{M,L,H,S,C \vdash \text{partition } r_p \text{ using } e_1 \text{ as } r_1,\ldots,r_k \text{ in } e_2 \mapsto v,E'} \quad \text{[E-Partition]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e_1 \mapsto v,E \qquad \rho_i = M[r_i], \text{for} 1 \leq i \leq k\\ v' = \langle\langle \rho_1,\ldots,\rho_k,v\rangle\rangle\end{array}}{M,L,H,S,C \vdash \text{pack } e_1 \text{ as } T_1[r_1,\ldots,r_k] \mapsto v',E} \quad \text{[E-Pack]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e_1 \mapsto \langle\langle \rho_1,\ldots,\rho_k,v_1\rangle\rangle,E_1 \qquad M' = M[\rho_1/r_1,\ldots,\rho_k/r_k]\\ L' = L[v_1/id] \qquad S' = \mathit{apply}(S,E_1)\\ M',L',H,S',C \vdash e_2 \mapsto v_2,E_2 \qquad \mathit{valid\_interleave}(S,C,E',E_1,E_2)\end{array}}{M,L,H,S,C \vdash \text{unpack } e_1 \text{ as } id : T_1[r_1,\ldots,r_k] \text{in } e_2 \mapsto v_2,E'} \quad \text{[E-Unpack]}$$

$$\frac{\begin{array}{l}M,L,H,S,C \vdash e_1 \mapsto v_1,E_1 \qquad S_1 = \mathit{apply}(S,E_1)\\ \quad \ldots\\ M,L,H,S_{n-1},C \vdash e_n \mapsto v_n,E_n \qquad S_n = \mathit{apply}(S_{n-1},E_n)\\ \mathit{valid\_interleave}(S,C,E',E_1,\ldots,E_n)\\ \text{function } id[r'_1,\ldots,r'_k](a_1 : T_1,\ldots,a_n : T_n),\Phi',Q' : T_r = e_{n+1}\\ M' = \{(r'_1,M(r_1)),\ldots(r'_k,M(r_k))\}\\ L' = \{(a_1,v_1),\ldots,(a_n,v_n)\}\\ S' = \mathit{apply}(S,E')\\ C' = C \cup \{l : \exists\rho.\ l \in \rho \wedge (\mathit{atomic}(\rho) \in M'[\![Q']\!] \vee \mathit{simult}(\rho) \in M'[\![Q']\!])\}\\ M',L',H,S',C' \vdash e_{n+1} \mapsto v_{n+1},E_{n+1}\\ E'_{n+1} = \mathit{mark\_coherence}(E_{n+1},M'[\![Q']\!],\mathit{taskid}) \qquad \mathit{taskid} \text{ fresh}\\ \mathit{valid\_interleave}(S,C,E'',E',E'_{n+1})\end{array}}{M,L,H,S,C \vdash id[r_1,\ldots,r_k](e_1,\ldots,e_n) \mapsto v_{n+1},E''} \quad \text{[E-Call]}$$

Figure 4. Type System and Operational Semantics

$$
\begin{aligned}
apply(S, []) &= S\\
apply(S, E{+\!+}[read(l,c,v,t)]) &= apply(S,E)\\
apply(S, E{+\!+}[write(l,c,v,t)]) &= apply(S,E)[v/l]\\
apply(S, E{+\!+}[reduce_{id}(l,c,v,t)]) &= S'[id(S'(l),v)/l],\\
&\quad \text{where } S' = apply(S,E)
\end{aligned}
$$

$$
\begin{aligned}
mark\_coherence([], \hat{Q}, taskid) &= []\\
mark\_coherence([op(l,c,v,t)]{+\!+}E, \hat{Q}, taskid) &= [op(l,c',v,taskid)]{+\!+}mark\_coherence(E,\hat{Q}),\\
\text{where } c' &= \begin{cases} simult, & \text{if } \exists \rho . l \in \rho \wedge \text{simult}(\rho) \in \hat{Q}\\ atomic, & \text{if } \exists \rho . l \in \rho \wedge \text{atomic}(\rho) \in \hat{Q}\\ excl, & \text{otherwise}\end{cases}
\end{aligned}
$$

Figure 5. Helper Functions for Type Rules and Operational Semantics

in the form of a *dynamic memory trace*. When necessary, the dynamic memory trace can be used to regenerate the final store using the $apply(S, E)$ helper function (see Figure 5).

Keeping the list of memory operations performed by the evaluation of an expression serves multiple purposes. First, the proof of soundness in Section 6 requires this list. Second, it makes it much easier to describe when and how computation of subexpressions may be interleaved (i.e. executed in parallel). As a simple example, consider the operational semantics rule for the addition of two integers:

$$
\frac{\begin{array}{l} M, L, H, S, C \vdash e_1 \mapsto v_1, E_1\\ S' = apply(S, E_1)\\ M, L, H, S', C \vdash e_2 \mapsto v_2, E_2\\ v' = v_1 + v_2\\ E' = valid\_interleave(S, C, E', E_1, E_2)\end{array}}{M, L, H, S, C \vdash e_1 + e_2 \mapsto v', E'} \quad \text{[E-Add]}
$$

In this rule, the subexpressions e_1 and e_2 are evaluated, producing memory traces E_1 and E_2. Our compositional semantics return a single memory trace E' for the parent expression by interleaving the individual operations from E_1 and E_2 according to certain constraints captured in the *valid_interleave* predicate, defined in Figure 7. A full explanation of these constraints is deferred to Section 7, but we describe the four most common cases here:

- If e_1 and e_2 access the same region(s) with exclusive coherence and there are no concurrently executing expressions that may modify the locations accessed by e_1 and e_2 (i.e. the locations are not in the clobber set C, see Section 5.2), then e_1 and e_2 execute in sequential program order, so $E' = E_1{+\!+}E_2$, where $+\!+$ is sequence concatenation.
- If e_1 and e_2 include task calls that access the same region(s) with atomic coherence (and there are no concurrent executions accessing the same locations), each of e_1 and e_2 must execute atomically, but the ordering of the two executions is not constrained. In this case, E' may be $E_1{+\!+}E_2$ or $\tilde{E}_2{+\!+}\tilde{E}_1$. ($\tilde{E}_2$ and $\tilde{E}_1$ are used in the second case to emphasize that the actual memory traces are likely to be different depending on which of e_1 or e_2 is executed first.)
- If e_1 and e_2 require exclusive (or atomic) coherence, but access disjoint sets of heap locations, they are *non-interfering* computations and may be performed in parallel while still giving the appearance of sequential execution. In this case, E' can be an arbitrary interleaving of the memory operations in E_1 and E_2.
- Finally, if e_1 and e_2 include task calls that access the same region(s) with simultaneous coherence, parallel computation of the subexpressions has been explicitly allowed by the programmer. The two computations may access the same locations and see the results of the other's writes and reductions. The resulting memory trace E' will be an interleaving of $\tilde{E}_1$ and $\tilde{E}_2$. (Again, $\tilde{E}_1$ and $\tilde{E}_2$ are used instead of E_1 to emphasize that the traces are likely to be different due to the interactions through the heap.)

5.2 Clobber Sets

As alluded to in the previous discussion, a composable parallel semantics must account for the unknown, concurrent context in which an expression executes. In particular, there may be locations read by an expression that are being altered (i.e. "clobbered") by other concurrently executing expressions. The set of such locations for a given expression is called the *clobber set* C. When a read is performed to a location that falls in C, the operational semantics leave the result of the read unconstrained. Instead, the check that the value of the read is consistent with the preceding writes (or reductions) is deferred to the first parent expression that encloses all of the computations that may be accessing the same locations.

To give a concrete example of how dynamic memory traces and clobber sets work, consider the following Core Legion tasks:

```
function A[r](i : int@r), reads(r), writes(r) : int =
  (B[r](i, 1) + B[r](i, 2)) + B[r](i, 3)

function B[r](i : int@r, v : int), reads(r), writes(r), atomic(r): int=
  let x: int = read(i) in
  let _: int@r = write(i, v) in
    x
```

There is a single region r in which a single integer has been allocated at location l (and given an initial value of 0), which is stored in pointer variable i. Function A requests exclusive access to r, and will return the sum of three calls to function B. Each call to function B performs an exchange on the memory location l, storing the value passed in as an argument and returning the original contents. Function B requests *atomic* coherence on region r allowing the three *sibling task* calls to B in A to execute in any order while guaranteeing that the individual exchanges are performed atomically. The scope of a coherence mode on a region for a task call t is always the sibling task calls of t within the parent task. The atomic coherence on region r affects the order of memory operations of the three calls to B within A, but not, for example, the interleaving of A with a sibling task, which is determined by A's exclusive coherence for r.

One valid execution for a call to $A[r](i)$ in a parent task would result in:

$$M, L, H, S, C \vdash A[r](i) \mapsto 5, E'$$

where:

$$\begin{aligned} M &= [\ r : \{l\}\ ] \\ L &= [\ i : l] \\ H &= [\ l : \mathit{int}\ ] \\ S &= [\ l \leftarrow 0\ ] \\ C &= \emptyset \\ E' &= [\ \mathit{read}(l, \mathit{excl}, 0, A), \mathit{write}(l, \mathit{excl}, 2, A), \\ &\qquad \mathit{read}(l, \mathit{excl}, 2, A), \mathit{write}(l, \mathit{excl}, 3, A), \\ &\qquad \mathit{read}(l, \mathit{excl}, 3, A), \mathit{write}(l, \mathit{excl}, 1, A)\ ] \end{aligned}$$

Recall that a memory trace records the sequence of memory operations performed by a task (and all of its subtasks). Each memory operation includes five pieces of information: the type of operation (*read*, *write*, or reduce_{id} with reduction operator *id*), the location affected, the coherence mode, the value that is read, written, or reduced (combined with the value already in the memory location by the reduction operator), and the unique identifier of the task performing the operation. Here the call $B[r](i, 2)$ (referred to as B_2 below) has executed first (reading the initial value 0 of i in the store), followed by $B[r](i, 3)$ (B_3 below) and finally $B[r](i, 1)$ (B_1). Note that the memory trace is *coherent* with respect to i: each read of i returns the value of the previous write of i or the initial value of i when there is no previous write. All the memory operations are marked with A's task id and with exclusive coherence, because this is the mode in which A accesses r. The fact that the accesses occurred in different subtasks of A (and with different coherence modes) is not visible outside of A.

To show how E' was obtained, we will follow the expression hierarchy, beginning at the leaf tasks:

$$\begin{aligned} B_1 &: M, [i : l, v : 1], H, S_{B_1}, \{l\} \vdash \mathit{let}\ x \ldots \mapsto 3, E_{B_1} \\ B_2 &: M, [i : l, v : 2], H, S_{B_2}, \{l\} \vdash \mathit{let}\ x \ldots \mapsto 0, E_{B_2} \\ B_3 &: M, [i : l, v : 3], H, S_{B_3}, \{l\} \vdash \mathit{let}\ x \ldots \mapsto 2, E_{B_3} \end{aligned}$$

where:

$$\begin{aligned} E_{B_1} &= [\ \mathit{read}(l, \mathit{excl}, 3, 0), \mathit{write}(l, \mathit{excl}, 1, 0)\ ] \\ E_{B_2} &= [\ \mathit{read}(l, \mathit{excl}, 0, 0), \mathit{write}(l, \mathit{excl}, 2, 0)\ ] \\ E_{B_3} &= [\ \mathit{read}(l, \mathit{excl}, 2, 0), \mathit{write}(l, \mathit{excl}, 3, 0)\ ] \end{aligned}$$

There are several important points to note here. First, each subtask's evaluation includes location l in the *clobber set*. Because these tasks access region r with *atomic* coherence, all locations in r (i.e. l) are added to the clobber set C' in the [E-Call] rule. This allows the *read* operations performed by the subtasks to return a value other than what is contained in the initial stores S_{B_1}, S_{B_2}, and S_{B_3} and allows the resulting dynamic memory traces to be non-coherent with respect to those initial stores. (Note that the stores are only used for the sequential portion of the semantics, which is the sequence of memory operations on locations with exclusive access that are not also in the clobber set. Thus, the stores are threaded through the rules in the usual sequential manner and used for operations on locations that can't be concurrently accessed.) Finally, although these tasks requested *atomic* coherence on region r, the memory operations within the task are marked with with the *excl* coherence mode, allowing proper ordering of the operations within each individual atomic subtask.

We next consider the function call expressions within the body of function A.

$$\begin{aligned} &M, L, H, S_{B_1}, \emptyset \vdash B[r](i, 1) \mapsto 3, E'_{B_1} \\ &M, L, H, S_{B_2}, \emptyset \vdash B[r](i, 2) \mapsto 0, E'_{B_2} \\ &M, L, H, S_{B_3}, \emptyset \vdash B[r](i, 3) \mapsto 2, E'_{B_3} \end{aligned}$$

where:

$$\begin{aligned} E'_{B_1} &= [\ \mathit{read}(l, \mathit{atomic}, 3, B_1), \mathit{write}(l, \mathit{atomic}, 1, B_1)\ ] \\ E'_{B_2} &= [\ \mathit{read}(l, \mathit{atomic}, 0, B_2), \mathit{write}(l, \mathit{atomic}, 2, B_2)\ ] \\ E'_{B_3} &= [\ \mathit{read}(l, \mathit{atomic}, 2, B_3), \mathit{write}(l, \mathit{atomic}, 3, B_3)\ ] \end{aligned}$$

Here we see the result of using the *mark_coherence* helper function (defined in Figure 5) to annotate the dynamic memory traces of function calls with their coherence modes and unique task id. The next step is to perform the inner addition:

$$M, L, H, S, \emptyset \vdash B[r](i, 1) + B[r](i, 2) \mapsto 3, E_{int}$$

where:

$$\begin{aligned} E_{int} = [\ &\mathit{read}(l, \mathit{atomic}, 0, B_2), \mathit{write}(l, \mathit{atomic}, 2, B_2) \\ &\mathit{read}(l, \mathit{atomic}, 3, B_1), \mathit{write}(l, \mathit{atomic}, 1, B_1)] \end{aligned}$$

Because all accesses to location l are performed with *atomic* coherence, either of $E'_{B_1} +\!\!+ E'_{B_2}$ or $E'_{B_2} +\!\!+ E'_{B_1}$ is permitted, and we have chosen the latter for our intermediate trace E_{int}. Note that this trace is not coherent (in particular, the second read of l does not return what was written by the previous write). Only sequential consistency of each subtask's accesses is required at this point.

The evaluation of the body of A is completed by performing the outer addition:

$$M, L, H, S, \emptyset \vdash (\ldots) + B[r](i, 3) \mapsto 5, E$$

where:

$$\begin{aligned} E = [\ &\mathit{read}(l, \mathit{atomic}, 0, B_2), \mathit{write}(l, \mathit{atomic}, 2, B_2) \\ &\mathit{read}(l, \mathit{atomic}, 2, B_3), \mathit{write}(l, \mathit{atomic}, 3, B_3), \\ &\mathit{read}(l, \mathit{atomic}, 3, B_1), \mathit{write}(l, \mathit{atomic}, 1, B_1)\ ] \end{aligned}$$

The requirements of the *valid_interleave* predicate allow for three possible interleavings of E_{int} and E'_{B_3}, and we have chosen the one that inserts E'_{B_3} in the middle of E_{int}. The final value of E' above is attained by applying the *mark_coherence* helper function to E, replacing the task ids and coherence modes of A's subtasks with those of A itself. Now that the accesses to location l are marked as *excl* rather than *atomic* and l is not in the clobber set, the trace is required to be coherent with respect to l, and this is the point at which any traces with inconsistencies between the choices of values read from location l in the calls to function B and the dynamic memory trace interleavings chosen in A are disallowed.

5.3 Operational Semantics Rules

In addition to the novel construction and interleaving of memory traces and clobber sets discussed above, the Core Legion operational semantics include rules for the new constructs introduced in the language. These rules are also shown in Figure 4.

The `new` expression selects a location that is not currently in use and that also has the correct heap typing from the set of locations assigned to the logical region argument. Similarly, `downregion` checks whether a location is within the set assigned to the logical region. If this dynamic check fails, *null* is returned. The application can use the `isnull` expression to test for this case and handle it appropriately. As discussed above, the correctness of `upregion` expressions is checked statically—there is no runtime component.

The `color` expression creates a copy of the input coloring in which the specified location is modified to have the specified color. The behavior of `newcolor` is subtler. The operational semantics for `new` requires that the newly allocated location already be present in the designated region. To allow allocations to be performed in subregions, additional, unused memory locations are assigned to each subregion when it is created. Because subregions are created by partitioning an existing region using a coloring, it is simplest to have `newcolor` put these extra locations in the initial coloring. Adding extra locations to a region cannot cause a computation to fail or alter its output, but it does admit executions in which some memory locations are assigned to a region but are never used (never allocated by `new`). This semantics reflects the behavior of our implementation, which also preallocates extra space in regions that may never be used, because adding space to a region on a call to `new` requires additional synchronization with users of that region and any containing regions to ensure all agree on the presence of the new location. It is much cheaper to simply add some extra locations when there is only a single user of the region, namely at the point where the region is created.

Because the necessary checks are performed at compile time, the operational semantics for the `pack` and `unpack` expressions are simple. A `pack` expression just uses M to map logical regions to physical regions, while `unpack` augments M with the new logical region names assigned to the physical regions stored in the region relationship.

6. Soundness of Privileges

Our first result shows that a well-typed expression accesses the heap in ways consistent with its static privileges. A judgment $E :_M \Phi$ holds if memory operations in memory trace E have types and locations covered by privileges Φ:

$$
\begin{aligned}
&E :_M \Phi \Leftrightarrow \forall \epsilon \in E. \\
&\quad (\epsilon = read(l,c,v,t) \Rightarrow \exists r, l \in M(r) \wedge \text{reads}(r) \in \Phi) \wedge \\
&\quad (\epsilon = write(l,c,v,t) \Rightarrow \exists r, l \in M(r) \wedge \text{writes}(r) \in \Phi) \wedge \\
&\quad (\epsilon = reduce_{id}(l,c,v,t) \Rightarrow \exists r, l \in M(r) \wedge \text{reduces}_{id}(r) \in \Phi)
\end{aligned}
$$

As usual, the soundness claim is proven assuming the initial type and execution environments are consistent. For our results, three consistency properties are needed:

- mapping consistency, written $M \sim \Omega$, guarantees a region mapping M satisfies the region constraints Ω
- local value consistency, written $L \sim_H M[\![\Gamma]\!]$, guarantees local values in L have types consistent with the environment Γ (using M to map logical regions in Γ to physical regions)
- store consistency, written $S \sim H$, guarantees locations in S have values consistent with heap typing H

Two additional properties are proven for each subexpression:

- result value consistency, written $v \sim_H M[\![T]\!]$, guarantees any evaluation of an expression yields a value of the right type
- memory trace consistency, written $E \sim H$, guarantees that all writes and reductions use values of the right types

Figure 6 defines these properties.

Theorem 1. If $\Gamma, \Phi, \Omega \vdash e : T$ and $M, L, H, S, C \vdash e \mapsto v, E$ and $M \sim \Omega$, $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$, then $v \sim_H M[\![T]\!]$, $E \sim H$ and $E :_M \Phi$.

In this section, we outline the general strategy of the proof, which makes use of a standard induction on the structure of the derivation. The full proof itself (which is lengthy primarily due to the number of expression types in Core Legion) can be found in Appendix B. For each of the Core Legion expressions, we show that the consistency of the expression's initial execution environment (i.e. mapping, local value, and store consistency) guarantees a consistent environment for subexpressions, and the consistency of subexpressions' results (i.e. result value consistency, memory trace consistency, and containment of heap accesses) results in similar consistency for the enclosing expression's results. Many of the cases are similar, and benefit from the use of the following lemmas (proofs of which can also be found in Appendix B). As discussed earlier, *apply*(S,E), defined in Figure 5, applies the operations in an execution trace E to a store S, the operator $++$ is sequence concatenation, and the *valid_interleave* predicate is defined in Figure 7.

Lemma 1. If $S \sim H$ and $E \sim H$, then *apply*$(S, E) \sim H$.

$$
\begin{array}{lcl}
M \sim \Omega & \Leftrightarrow & (\forall r_i, r_j . r_i \leq r_j \in \Omega \Rightarrow M(r_i) \subseteq M(r_j)) \wedge \\
 & & (\forall r_i, r_j . r_i * r_j \in \Omega \Rightarrow M(r_i) \cap M(r_j) = \emptyset) \\
L \sim_H M[\![\Gamma]\!] & \Leftrightarrow & \forall (id, v) \in L . v \sim_H M[\![\Gamma]\!](id) \\
S \sim H & \Leftrightarrow & \forall (l, v) \in S . v \sim_H H(l)
\end{array}
$$

$$
\begin{array}{lcl}
E \sim H & \Leftrightarrow & (\forall l, c, v . write(l, c, v, t) \in E \Rightarrow v \sim_H H(l)) \wedge \\
 & & (\forall id, l, v . reduce_{id}(l, c, v, t) \in E \Rightarrow \\
 & & (M[\![\Gamma]\!](id) = (\hat{T}_1, \hat{T}_2), \emptyset, \emptyset \rightarrow \hat{T}_1) \wedge H(l) = \hat{T}_1 \wedge v \sim_H \hat{T}_2)
\end{array}
$$

$$
\begin{array}{llcl}
 & l \sim_H \hat{T}@\rho & \Leftrightarrow & l \in \rho \wedge H(l) = \hat{T} \\
bv \sim_H bool & \langle v_1, v_2 \rangle \sim_H \langle \hat{T}_1, \hat{T}_2 \rangle & \Leftrightarrow & (v_1 \sim_H \hat{T}_1) \wedge (v_2 \sim_H \hat{T}_2) \\
iv \sim_H int & \langle\langle \rho_1, \ldots, \rho_n, v \rangle\rangle \sim_H \mathrm{rr}[r_1, \ldots, r_n]\ \hat{T} \text{ where } \hat{\Omega} & \Leftrightarrow & (v \sim_H T[\rho_1/r_1, \ldots \rho_n/r_n]) \wedge (\{(r_i, \rho_i)\} \sim \hat{\Omega}) \\
null \sim_H \hat{T}@\rho & K \sim_H coloring(\rho) & \Leftrightarrow & \forall l_1, v_1 . (l_1, v_1) \in K \Rightarrow (l_1 \in \rho \wedge \\
 & & & \forall l_2, v_2 . (l_2, v_2) \in K \Rightarrow (l_1 \neq l_2) \vee (v_1 = v_2))
\end{array}
$$

Figure 6. Consistency Properties

Lemma 2. If $E_1 \sim H$ and $E_2 \sim H$, then $E_1 +\!\!+ E_2 \sim H$.

Lemma 3. If $E_1 \sim H$ and $E_2 \sim H$ and $valid\_interleave(S, C, E', E_1, E_2)$, then $E' \sim H$.

Lemma 4. If $E_1 :_M \Phi$ and $E_2 :_M \Phi$, then $E_1 +\!\!+ E_2 :_M \Phi$.

Lemma 5. If $E_1 :_M \Phi$ and $E_2 :_M \Phi$ and $valid\_interleave(S, C, E', E_1, E_2)$, then $E' :_M \Phi$.

Lemma 6. $M \sim \Omega^*$ if and only if $M \sim \Omega$.

Lemma 7. $E :_M \Phi^*$ if and only if $E :_M \Phi$.

Lemma 8. $M \sim \Omega$ if and only if $\emptyset \sim M[\![\Omega]\!]$.

The interesting cases for each property are summarized here:

$M \sim \Omega$ - Three expressions have subexpressions that modify M or Ω and therefore do not trivially satisfy region mapping consistency. For `partition`, the consistency of the coloring preserves region mapping consistency with respect to the constraints. For `unpack`, the consistency of a region relation instance guarantees consistency of region mapping. Finally, the body of a called function uses an initially-empty set of constraints, which are trivially satisfied.

$L \sim_H M[\![\Gamma]\!]$ - Four expressions have subexpressions that modify L, Γ, or M. For `partition`, which only modifies M, the requirement that it not reuse existing names ensures that $M[\![\Gamma]\!]$ does not change. For `let`, the value and type of the binding is obviously consistent, while the binding created in an `unpack` is less obviously so, requiring an induction over the type of the unpacked value to show equivalence under the new mapping. The last case is the body of a called function, which requires the same style of proof as for `unpack` for each formal parameter.

$S \sim H$ - The heap typing consistency of all stores used in subexpressions follows directly from Lemma 1.

$v \sim_H M[\![T]\!]$ - The consistency of `upregion` is guaranteed by the type checking requirement of appropriate subregion constraints and the mapping's consistency with those constraints, and `downregion`'s result is consistent because of the runtime check. The consistency of a `read`'s result is trivial for an address in the clobber set and uses the consistency of the store otherwise. The consistency of a `color`'s result depends on the pointer subexpression's consistency and the removal of any previous coloring of that location from the coloring set.

The remaining interesting cases arise from changes to the mapping M rather than transformations on the value v. In the case of `partition` and `unpack`, the type system guarantees that the subexpression's result cannot use the regions that were added to the mapping, allowing the changes to the mapping to be ignored. The last case is again the body of a called function, and the same strategy that was used for the type consistency of the formal parameters works in reverse for the function's result.

$E \sim H$ - The type consistency of the values in an expression's memory trace follows from Lemma 2 and Lemma 3. New memory operations are added by `write` and `reduce` expressions, but consistency follows directly from the induction hypothesis. Finally, the consistency of the values in a called function's memory trace is addressed in the same way as the return value.

$E :_M \Phi$ - The proof of the crucial property of containment of heap accesses within the available privileges is similar in outline to the previous step. The easy cases are covered by Lemma 3. Straightforward proofs cover `read`, `write`, `reduce`, with one final special case for function calls.

7. Coherence

In our compositional operational semantics, the execution of an expression assumes any concurrent environment and there may be many possible execution traces for a given expression. When the semantics of multiple subexpressions are combined in the operational semantics rules, we can restrict the set of execution traces to those that are consistent with the joint behavior of the subexpressions under the given region coherence requirements.

Interestingly, however, an insight from the proof of Theorem 1 is that it does not rely on the full definition of *valid_interleave*. In fact, soundness of privileges is preserved even if the *valid_interleave* test is replaced with *any_interleave* (Figure 7), which allows arbitrary interleavings of memory traces from subexpressions. The stronger constraints in *valid_interleave* address the coherence of heap accesses, specifying permitted interleavings of memory operations for the particular coherence modes on logical regions.

To determine whether an interleaving of two or more memory traces is valid, we consider three sets of addresses:

- exclusive locations ($l \in L_{excl}$) are those which have at least one access in exclusive mode in the traces and are not in the clobber set. For these locations, we require sequential execution semantics—all reads to these locations see the effect of previous writes and reductions, and the resulting state of the store is as if all writes and reductions were applied from each trace in order.
- atomic locations ($l \in L_{atomic}$) are those which have at least one access in atomic mode in the traces and are in neither L_{excl} nor the clobber set. For these locations, we allow permutations of the original subexpression trace order.
- for locations with only access in *simult* mode or in the clobber set, no constraints are enforced. The valid interleaving of these accesses is determined within the context of the closest enclosing task call where the locations are neither in the clobber set nor accessed only with simultaneous coherence.

7.1 Sequential Execution

We now show that a sequential execution trivially satisfies the interleaving criteria required by the operational semantics. Our proof of the soundness of parallel scheduling depends on this result.

Sequential execution ignores the coherence mode Q in all function calls, using $Q' = \emptyset$ instead, and interleaves traces by concatenating the subexpressions' traces in program order. By ignoring the coherence modes, the clobber set remains empty and the result of all `read` expressions is fully determined. The following lemma and theorem show that the value and memory trace that result from a sequential execution are always valid executions.

Lemma 9. Let $\Gamma, \Phi, \Omega \vdash e : T$ and $M, L, H, S, C \vdash e \mapsto v, E$ and $S \sim H$. If $C \subseteq C'$, then $M, L, H, S, C' \vdash e \mapsto v, E$.

Theorem 2. Let $e_1, \ldots, e_n$ be expressions such that

$$M, L, H, S_{i-1}, C \vdash e_i \mapsto v_i, E_i$$

where $S_i = apply(S_{i-1}, E_i)$. If $E' = E_1 +\!\!+ \ldots +\!\!+ E_n$, then $valid\_interleave(S_0, C, E', E_1, \ldots, E_n)$.

7.2 Parallel Execution

To determine when parallel execution is safe, we start from the sequential execution trace and allow the reordering of adjacent heap operations that do not change the behavior of the application. If we can show that it is safe to reorder any pair of operations that come from two different constituent traces, then any interleaving of the constituent traces will be equivalent to a sequential execution and parallel execution is safe. To efficiently discover these cases at runtime, we require a test that can determine this property prior to the actual execution of the tasks that create the traces. We show that a test based on the subtask privileges and the current region mapping can soundly predict when this property will hold. We begin by defining a *non-interference* operator on two memory operations $\epsilon_1 = op_1(l_1, c_1, v_1, t_1)$ and $\epsilon_2 = op_2(l_2, c_2, v_2, t_2)$:

$$\begin{aligned}\epsilon_1 \# \epsilon_2 \Leftrightarrow & (op_1 = read \wedge op_2 = read) \vee \\ & (op_1 = reduce_{id_1} \wedge op_2 = reduce_{id_2} \wedge id_1 = id_2) \vee \\ & l_1 \neq l_2\end{aligned}$$

Reads have no side effects, and cannot change what another read returns. The safety of the second case follows from the requirement that reduction operations be commutative. Finally, accesses to different locations cannot affect each other. Therefore, an adjacent pair of non-interfering memory operations in a memory trace can be reordered while preserving the validity of an interleaving.

Lemma 10. Let S be a store, C a clobber set, $E_1, \ldots, E_n, E'_a, E'_b$ memory traces, and ϵ_1, ϵ_2 be two memory operations from E_i and E_j ($i \neq j$). Then,

$$\begin{aligned}& valid\_interleave(S, C, E'_a +\!\!+ [\epsilon_1, \epsilon_2] +\!\!+ E'_b, E_1, \ldots, E_n) \wedge \epsilon_1 \# \epsilon_2 \\ & \quad \Rightarrow valid\_interleave(S, C, E'_a +\!\!+ [\epsilon_2, \epsilon_1] +\!\!+ E'_b, E_1, \ldots, E_n).\end{aligned}$$

Two whole memory traces are non-interfering if no operation from one trace interferes with any from the other:

$$E_1 \# E_2 \Leftrightarrow \bigwedge_{\epsilon_1 \text{ in } E_1, \epsilon_2 \text{ in } E_2} \epsilon_1 \# \epsilon_2$$

If whole memory traces are non-interfering, any interleaving can be sorted via pairwise swaps to match the sequential memory trace. This gives us a result permitting safe parallel execution:

Lemma 11. Let S be an initial store, C be a clobber set, $E_1, \ldots, E_n$ be memory traces such that $E_i \# E_j$ for every $1 \leq i < j \leq n$. Then, $any\_interleave(E', E_1, \ldots, E_n) \Rightarrow valid\_interleave(S, C, E', E_1, \ldots, E_n)$.

We now use the bounds that static privileges place on runtime accesses to give an efficient runtime test for non-interference. We first extend the non-interference operator to work on privileges:

$$
\begin{array}{lcl}
\mathit{any\_interleave}([], [], \ldots, []) & = & \mathit{true} \\
\mathit{any\_interleave}([\epsilon]{+\!\!+}E', E_1, \ldots, [\epsilon]{+\!\!+}E_i, \ldots, E_n) & = & \mathit{any\_interleave}(E', E_1, \ldots, E_i, \ldots, E_n)
\end{array}
$$

$$
\begin{array}{l}
\mathit{valid\_interleave}(S, C, E', E_1, \ldots, E_n) = \\
\quad \mathit{any\_interleave}(E', E_1, \ldots, E_n) \wedge \\
\quad \mathit{coherent}(S, L_{excl}(E', C), L_{excl}(E', C), E') \wedge \\
\quad \mathit{seq\_equiv}(S, L_{excl}(E', C), L_{excl}(E', C), E', E_1, \ldots, E_n) \wedge \\
\quad \forall t. \mathit{seq\_equiv}(S, L_{atomic}(E', C), \emptyset, E' \downarrow_t, (E_1 {+\!\!+} \ldots {+\!\!+} E_n) \downarrow_t)
\end{array}
$$

$$
\begin{array}{l}
\mathit{coherent}(S, L_1, L_2, []) = \text{true} \\
\mathit{coherent}(S, L_1, L_2, [\epsilon]{+\!\!+}E) = \\
\quad \begin{cases}
(l \in L_2 \Rightarrow S(l) = v) \wedge & \\
\quad \mathit{coherent}(S, L_1, L_2, E), & \text{if } \epsilon = \mathit{read}(l, c, v, t) \\
\mathit{coherent}(\mathit{apply}(S, \epsilon), L_1, L_2 \cup \{l\}, E), & \text{if } \epsilon = \mathit{write}(l, c, v, t) \\
& \quad \text{and } l \in L_1 \\
\mathit{coherent}(\mathit{apply}(S, \epsilon), L_1, L_2, E), & \text{otherwise}
\end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
L_{excl}(E, C) & = & \{l : \mathit{op}(l, \mathit{excl}, v, t) \text{ in } E\} \setminus C \\
L_{atomic}(E, C) & = & \{l : \mathit{op}(l, \mathit{atomic}, v, t) \text{ in } E\} \setminus (C \cup L_{excl}(E, C))
\end{array}
$$

$$
\begin{array}{l}
\mathit{seq\_equiv}(S, L_1, L_2, E', E_1, \ldots, E_n) = \\
\quad \mathit{coherent}(S, L_1, L_2, E_1 {+\!\!+} \ldots {+\!\!+} E_n) \wedge \\
\quad \forall l \in L_1. \mathit{apply}(S, E')(l) = \mathit{apply}(S, E_1 {+\!\!+} \ldots {+\!\!+} E_n)(l)
\end{array}
$$

$$
\begin{array}{lcl}
[] \downarrow_t & = & [] \\
(\mathit{op}(l, c, v, t'){+\!\!+}E) \downarrow_t & = & \begin{cases} \mathit{op}(l, c, v, t'){+\!\!+}(E \downarrow_t), & \text{if } t = t' \\ E \downarrow_t, & \text{otherwise} \end{cases}
\end{array}
$$

Figure 7. Valid Interleaving Test

$$
\begin{array}{l}
\mathit{priv}_1(r_1) \#_M \mathit{priv}_2(r_2) \Leftrightarrow \\
\quad (\mathit{priv}_1 = \text{reads} \wedge \mathit{priv}_2 = \text{reads}) \vee \\
\quad (\mathit{priv}_1 = \text{reduces}_{id_1} \wedge \mathit{priv}_2 = \text{reduces}_{id_2} \wedge id_1 = id_2) \vee \\
\quad (r_1 * r_2) \vee \\
\quad (M(r_1) \cap M(r_2) = \emptyset)
\end{array}
$$

$$
\Phi_1 \#_M \Phi_2 \Leftrightarrow \bigwedge_{\phi_1 \in \Phi_1, \phi_2 \in \Phi_2} \phi_1 \#_M \phi_2
$$

The cases where both subtasks have read-only privileges or both subtasks have reduce-only privileges have equivalents for regions, which can be tested statically. Detecting the case where the two sets of memory addresses are disjoint is approximated by two tests. The first uses (logical) region disjointness constraints from the type system to statically infer non-interference. The second uses the region mapping M to dynamically determine the disjointness of the two regions. Although a dynamic test, it is performed once per pair of regions rather than for every pair of memory operations. An algorithm to perform the dynamic test efficiently is given in [2]. As region non-interference is an approximation of memory trace non-interference, we must show that it is sound.

Lemma 12. Let M be a region mapping and E_1 and E_2 two memory traces such that $E_1 :_M \Phi_1$ and $E_2 :_M \Phi_2$. If Φ_1 and Φ_2 are non-interfering under M, then E_1 and E_2 must be non-interfering.

We now state the theorem that allows the Legion runtime to perform hierarchical and parallel scheduling of non-interfering tasks.

Theorem 3. Let $e_1, \ldots, e_n$ be well-typed Legion expressions, each with its own privileges Φ_i. Let M be a region mapping, L a local value mapping, H a heap typing, and S be an initial store satisfying $M \sim \Omega$, $L \sim_H M[\![\Gamma]\!]$, and $S \sim H$. If $\Phi_i \#_M \Phi_j$ for $1 \le i < j \le n$, then any parallel execution of expressions $e_1, \ldots, e_n$ results in a valid interleaving of memory operations.

The proof follows directly from Lemmas 11 and 12. This result holds even if the clobber set C is non-empty, allowing locally independent subtasks to run in parallel even if they interact (in a programmer-permitted way) with another subtask.

We highlight an important aspect of a Legion implementation that is different from other systems and relies on the soundness of privileges. Dynamic non-interference of memory operations can only be determined after evaluation of an expression is completed, and only at great expense, as illustrated by work on transactional memory [13]. At the other extreme are systems like Jade [17] and DPJ [5] that check non-interference statically, but must disallow aliasing to do so. In contrast, Legion can verify non-interference of privileges at runtime, which is much simpler and more efficient than checking non-interference of dynamic memory traces. Even though the privileges themselves are static, the region mapping M is dynamic. Dynamically testing non-interference on the privileges of physical regions allows parallel execution in many more cases than a purely static analysis can achieve in the presence of aliasing. When a dynamic test fails, the Legion runtime is conservative and forces sequential ordering between the tasks to guarantee correct behavior.

7.3 Atomic Coherence

In cases where Legion cannot safely infer non-interference of privileges (perhaps because two tasks actually access the same data in aliased regions), relaxation of the constraints on execution order can still be requested by the programmer through the use of coherence annotations on individual regions passed to a task. The *atomic* coherence mode specifies that although two tasks interfere due to accessing aliased regions, they may execute in either order, allowing the task issued later in program order to possibly run before the task issued earlier in program order. This relaxation only applies if all aliased regions are annotated with atomic coherence. To show this is safe, we define a relaxed version of non-interference for atomic coherence:

$$\begin{aligned} &op_1(l_1, c_1, v_1, t_1) \#^A op_2(l_2, c_2, v_2, t_2) \Leftrightarrow \\ &\quad op_1(l_1, c_1, v_1, t_1) \# op_2(l_2, c_2, v_2, t_2) \vee \\ &\quad (c_1 = atomic \wedge c_2 = atomic \wedge t_1 \neq t_2) \end{aligned}$$

We repeat the steps in Section 7.2 using the $\#^A$ operator and reach another result used by the Legion runtime scheduler:

Theorem 4. Let $e_1, \dots, e_n$ be well-typed Legion expressions, each with its own privileges Φ_i. Let M be a region mapping, L a local value mapping, H a heap typing, and S be an initial store satisfying $M \sim \Omega$, $L \sim_H M[\![\Gamma]\!]$, and $S \sim H$. If $\Phi_i \#_M^A \Phi_j$ for $1 \leq i < j \leq n$, then for any permutation $(\pi_1, \dots, \pi_n)$ of $(1, \dots, n)$, $E_{\pi_1} +\!\!+ \dots +\!\!+ E_{\pi_n}$ is a valid interleaving.

7.4 Simultaneous Coherence

Coherence also can be relaxed using the *simult* mode, which allows multiple tasks to access the same region concurrently. The *simult* coherence mode is appropriate in two important cases:

1. When subtasks are accessing disjoint data, but the disjointness is difficult to describe (e.g. walking separate linked lists that have been allocated in the same region).
2. When the algorithm is tolerant of non-determinism (e.g. in a breadth-first search, setting the parent pointer of a node with multiple equally-short paths to the root).

To support the *simult* coherence, the non-interference test is extended with a $\#^S$ operator, analogous to $\#^A$ for atomic coherence. Because the rules for valid interleavings exclude locations that are only accessed in simult mode, it is straightforward to extend Theorem 3 to show that parallel execution is safe as long as $\Phi_i \#_M^S \Phi_j$.

It is also possible to have both atomic and simult coherence modes at the same time for different regions in a task call. In this case the non-interference test $\#^{AS}$ uses both the atomic and simult relaxations, and Theorem 4 is extended to allow arbitrary reordering (but not simultaneous execution) of subtasks when $\Phi_i \#_M^{AS} \Phi_j$.

8. Hierarchical Scheduling

Because testing non-interference of tasks is a pairwise operation, scheduling n tasks can require $\mathcal{O}(n^2)$ tests. Thus, a scheduler that must globally consider all pairs of tasks will be impractical for large machines and large numbers of tasks. The following theorem, however, shows that Legion programs enjoy a locality property that limits the scope of the needed non-interference tests.

Theorem 5. Let e_1 and e_2 be well-typed expressions using privileges Φ_1 and Φ_2 respectively, where $\Phi_1 \#_M \Phi_2$. Let e_1' be a subexpression of e_1 and e_2' be a subexpression of e_2. Any memory traces E_1' of e_1' and E_2' of e_2' resulting from evaluation of e_1 and e_2 (with the usual consistent M, L, H, and S) are non-interfering.

The Legion task scheduler uses Theorem 5 as follows: sibling function calls (those invoked within the same function body) e_1 and e_2 are checked for non-interference of their (dynamic) privileges. Since e_1 and e_2 are called on the parent task's node, no communication is required to perform the non-interference test. If they interfere they are executed in program order or serialized depending on their coherence specifications; otherwise they are considered for execution as parallel subtasks. If e_1 and e_2 are determined to be non-interfering and are scheduled in parallel on different remote processors then Theorem 5 guarantees that there is no communication required between e_1 and e_2 to perform non-interference tests between their sub-tasks. Therefore, the runtime requires no communication for scheduling.

9. Evaluation

We evaluate the design of Legion's static and dynamic semantics on four criteria: expressivity (can real applications be written—Section 9.1), overhead (what are the dynamic checking costs—Section 9.2), scalability (can it enable hierarchical scheduling—Section 9.3), and performance (does the performance increase from relaxed coherence modes warrant the increased semantic complexity—Section 9.4). Our prototype implementation has two components: a type checker for the language of Section 3 and a C++ runtime library for executing programs written in the Legion programming model. All experiments are conducted on the Keeneland supercomputer[19]. Each node of Keeneland consists of two Xeon 5660 CPUs, three Tesla M2090 GPUs, and 24 GB of DRAM. Nodes are connected by a QDR Infiniband interconnect.

9.1 Expressivity

We evaluate Legion on three real-world applications. To qualitatively gauge the expressivity of Legion, we introduce these applications by describing features used in their implementations. The Circuit example was already covered in detail in Section 2.

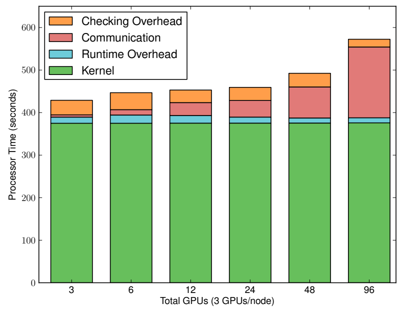


Figure 8. Overhead in Circuit simulation with 96 pieces.

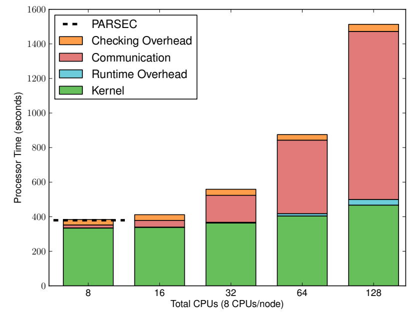


Figure 9. Overhead in Fluid simulation with 19200 cells.

Fluid is a distributed memory version of the `fluidanimate` benchmark from the PARSEC suite[3]. Fluid simulates the flow of an incompressible fluid using particles that move within a regular grid of cells. To perform operations in parallel, the array of cells is partitioned. Unlike Circuit, Fluid creates and partitions regions before allocating cells in them. Another difference is that Fluid maintains separate regions for ghost cells rather than using multiple partitions of the regions containing shared data. Region relationships are used to capture which regions are required for each grid.

The third application is a Legion port of an adaptive mesh refinement (AMR) benchmark from BoxLib [15]. The algorithm solves the two dimensional heat diffusion equation on a grid of cells using three levels of refinement with sub-refinements randomly placed on the surface. Every level of refinement uses a separate region, which is partitioned several ways to support multiple views of the cells. One partitioning separates cells into pieces that can be updated in parallel. Additional partitions are created for viewing data from coarser and finer levels of the simulation. Two types of region relationships are created: one describes pieces at each level of refinement, and another describes relationships between pieces at different levels of refinement. The dynamic nature of AMR requires that regions be created and partitioned at runtime.

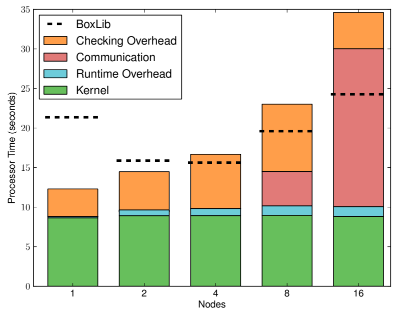
Dynamically creating and partitioning regions at runtime is crucial to Legion's ability to handle applications that make runtime decisions about data organization (AMR). Having multiple partitions of regions is necessary for describing the many ways that data can be accessed (Circuit, AMR). All the types of privileges and coherence are needed in some application; region relationships are used in all applications. Finally, all applications introduce aliasing of data either through the use of multicolorings or by having multiple partitions. Our implementations of these applications both type check and execute, proving that our type system is sufficiently expressive to handle real-world applications.

9.2 Checking Overhead

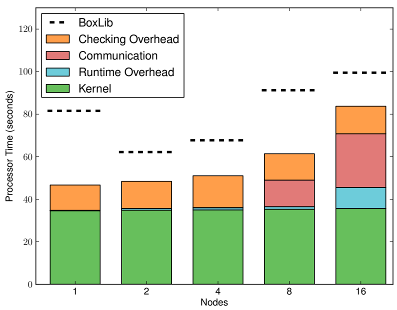
The first Legion implementation consisted of a C++ library of Legion primitives [2] with no checking of region memory accesses. When using this system we frequently encountered memory corruption due to illegal region accesses caused by application bugs. In many cases, this corruption occurred between nodes in the cluster or on GPUs, environments for which debugging tools are primitive at best. To locate the application bugs causing these illegal accesses, we initially added dynamic checks on all region accesses for both CPUs and GPUs which added considerable runtime overhead. In short, the standard benefits of type checking (increased program safety and efficiency) are magnified in high performance parallel applications, because debugging is so difficult and efficiency considerations are paramount. To preserve the benefit of checking every access without the cost of dynamic checks, we implemented the type, privilege, and coherence checker we have described. We then rewrote the applications in this language and type checked them, at which point the dynamic region access checks could be safely elided.

Figures 8, 9, and 10 show the total time spent by all CPUs and GPUs in each phase of the application. The topmost component of each bar shows the overhead added by the dynamic checks. In each figure the problem size stays the same as the number of processors increases (strong scaling). Figure 10 includes multiple problem sizes to show how overhead is affected by changing problem size (weak scaling). For cases where there is an existing implementation to compare against we have included a dotted line indicating baseline performance. In a few cases (Figures 9 and 10(a)), the checking overhead is the difference between better and worse performance than the baseline. The overall performance relative to the baseline implementations is discussed in [2].

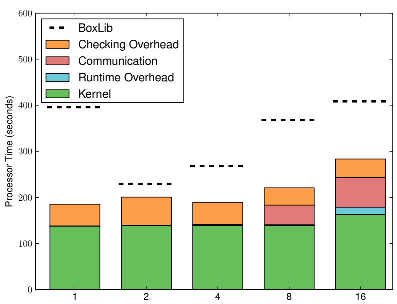
In addition to total processor overhead, we also measured performance gain from eliding checks in terms of wall-clock time. Since most region accesses occur in leaf tasks, the checks parallelize well. Wall-clock performance gains from eliding memory checks ranged from 1-10%, 1-15%, and 2-71% for Circuit, Fluid, and AMR respectively. Performance gains for AMR were larger than the other applications because AMR was already memory bound and the additional checks intensified memory pressure. For the GPU kernels in the Circuit application checking required up to 8 additional registers per thread. While the GPU kernels in Cir-



(a) 4096 cells per dimension



(b) 8192 cells per dimension



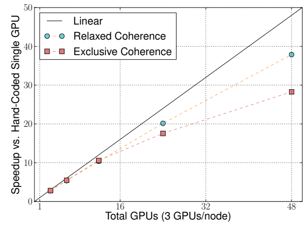
(c) 16384 cells per dimension

Figure 10. Overhead in AMR application.

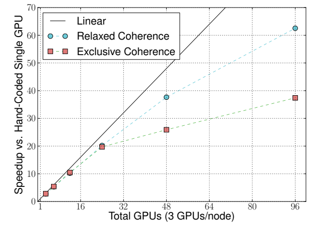
cuit were not bound by available on-chip memory, kernels that are would be susceptible to extreme performance degradation due to the extra registers required for checking. We also measured the overhead of the dynamic checks associated with checking task call privileges but found them to be negligible, demonstrating that runtime non-interference checks are inexpensive in Legion.

9.3 Scalability

Figures 8, 9, and 10 show that the overhead of the Legion runtime is always less than 7% of the total execution time of the applications. In some applications communication does not scale well, but this is a result of the algorithm required by the application, not the Legion runtime. Even in the case of the Circuit application, which exhibits quadratic increases in communication cost, the Legion runtime is able to achieve a 62.5X speedup on 96 GPUs over a hand-coded single GPU implementation[2].



(a) 48 Piece Problem Size



(b) 96 Piece Problem Size

Figure 11. Performance of relaxed coherence modes.

9.4 Performance

To demonstrate the benefit of relaxed coherence modes, we modified the circuit example from Section 2 to use exclusive coherence instead of atomic coherence in the *distribute_charge* task and compared the performance of the two versions. The results are shown in Figure 11. Slowdowns ranged from 34% on 48-GPUs to 67% on 96 GPUs and more importantly scaled with node count. This is a direct consequence of Amdahl's Law. Even though the *distribute_charge* tasks are a small fraction of the total work, the serialization that results from requiring exclusive access to the overlapping ghost regions limits the scalability of the application. Relaxed coherence modes will be crucial in achieving scalability of applications with aliased data on distributed memory machines.

10. Related Work

Legion is most directly related to Sequoia [1, 10]. Sequoia is a static language with a single unified hierarchy of tasks and data; Legion is more dynamic with separate task and region hierarchies.

Deterministic Parallel Java (DPJ) is the only other region-based parallel system of which we are aware[4]. While there are similarities between DPJ's effects and Legion privileges, there are differences stemming from DPJ's static approach. Regions in Legion are first-class and can be created, partitioned, packed, and unpacked dynamically, allowing programmers to compute data organization at runtime; like Sequoia, DPJ partitioning schemes are static. Legion allows programmers to create multiple partitions of the same region to give different views onto the same data, which is not possible in DPJ. DPJ supports both exclusive and atomic tasks which are similar to Legion's coherence modes, but only allows specification at the coarser granularity of tasks and not individual regions.

Chapel [7] and X10 [8] also provide some Legion-like facilities. Chapel's locales and X10's places provide the programmer with a mechanism for expressing locality, similar to regions in Legion. However, locales and places are not used for independence analysis to discover parallelism. In contrast, Jade uses annotations to describe data disjointness, and like Legion leverages the disjointness information to discover parallelism, but lacks a region system to name and organize unbounded collections of objects [17].

Hierarchical Place Trees (HPT) [20] is a generalization of the Sequoia and X10 program models. HPT presents hierarchical places in which to put data; places are mapped onto physical locations in the memory hierarchy. HPT has no equivalent to partitioning in Legion, leaving the burden on the programmer to ensure that data is moved correctly through the place hierarchy and to ensure the safety of parallel task execution.

Many efforts use static region systems for memory management (e.g., [12, 18]). Our system is more closely related to dynamic region systems used for expressing locality for performance [11]. Titanium is an SPMD parallel language with a region system where regions are tightly bound to specific processors [21].

There have been many type and effect systems for ownership types [6] including ones that leverage nested regions for describing relationships (e.g., [9]). However, ownership type and effect systems are primarily used for reasoning about determinism in object oriented languages and don't capture the range of disjointness properties in Legion. Reasoning about disjoint data is the strong suit of separation logic [16]. While we have borrowed some separation logic notation, we chose to use a privileges system because separation logic does not easily support reasoning about the interleaving of operations to aliased regions of memory.

11. Conclusion

Modern architectures have dramatically increased in complexity in recent years. To program this class of machines, new programming systems will be required that are capable of reasoning about the structure of data and how it should be partitioned. We have presented the static and dynamic semantics for the Legion programming system, showing how a combination of static and dynamic checks can be used to support region-level privileges and coherence, even in the presence of dynamically partitioned and aliased data. We have also given a novel compositional parallel semantics, permitting a precise treatment of relaxed coherence modes; in particular we have shown the Legion design is sound even with relaxed coherence. These semantics make possible a novel hierarchical scheduling algorithm that is crucial for scaling on large distributed machines. Finally, we have demonstrated that our system enables static elision of many dynamic checks leading to large performance improvements on real world applications.

References

[1] M. Bauer, J. Clark, E. Schkufza, and A. Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in Sequoia. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2011.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Supercomputing (SC)*, 2012.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] R. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[5] R. Bocchino et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, 2003.

[7] B.L. Chamberlain et al. Parallel programmability and the chapel language. *Int'l Journal of HPC Applications*, 2007.

[8] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[9] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.

[10] K. Fatahalian et al. Sequoia: Programming the Memory Hierarchy. In *Supercomputing*, November 2006.

[11] D. Gay and A. Aiken. Language support for regions. In *PLDI*, 2001.

[12] D. Grossman et al. Region-based memory management in cyclone. In *PLDI*, 2002.

[13] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, 2005.

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 1998.

[15] M. Lijewski, A. Nonaka, and J. Bell. Boxlib. `https://ccse.lbl.gov/BoxLib/index.html`, 2011.

[16] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in CS*, 2002.

[17] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 1998.

[18] M. Tofte and J.P. Talpin. Region-based memory management. In *POPL*, 1994.

[19] J.S. Vetter et al. Keeneland: Bringing heterogeneous gpu computing to the computational science community. *Computing in Science Engineering*, pages 90 –95, 2011.

[20] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Workshop on Languages and Compilers for Parallel Computing*, 2009.

[21] K. Yelick et al. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing*, 1998.

A. Core Legion Circuit Code

```
−−                    ⟨voltage,current,charge,capacitance,piece ID⟩
type CircuitNode        = ⟨int,int,int,int,int⟩
−−                    ⟨ owned node, owned or ghost node, resistance, current⟩
type CircuitWire⟨rn,rg⟩  = ⟨CircuitNode@rn, CircuitNode@(rn,rg),int,int⟩

type NodeList⟨rl,rn⟩       = ⟨ CircuitNode@rn, NodeList⟨rl,rn⟩@rl ⟩
type WireList⟨rl,rw,rn,rg⟩= ⟨ CircuitWire⟨rn,rg⟩@rw, WireList⟨rl,rw,rn,rg⟩@rl ⟩
function extern_metis[rl,rn,rw](node_list : NodeList⟨rl,rn⟩@rl,
        wires_list : WireList⟨rl,rw,rn,rn⟩@rl), reads(rl,rn,rw), writes(rn) : bool

type CircuitPiece⟨rl,rw,rn⟩ = rr[rpw,rpn,rg]
                    ⟨ WireList⟨rl,rpw,rpn,rg⟩@rl, NodeList⟨rl,rpn⟩@rl ⟩
                    where rpn ≤ rn and rg ≤ rn and rpw ≤ rw and
                        rn ∗ rw and rl ∗ rn and rl ∗ rw

−− Multicoloring helper for aliased partitions
type multicoloring⟨rn⟩ = ⟨ coloring(rn), coloring(rn) ⟩

−− Simulation initialization and invocation
function simulate_circuit[rl,rw,rn] ( all_nodes : NodeList⟨rl,rn⟩@rl,
                    all_wires : WireList⟨rl,rw,rn,rn⟩@rl, steps : int ),
    reads(rn,rw,rl), writes(rn,rw,rl) : bool =
  −− use METIS to decide how to partition circuit
  let _ : bool = extern_metis[rl,rn,rw](all_nodes, all_wires) in

  −− create colorings to describe METIS results to Legion
  let owned_node_map : coloring(rn) = owned_node_coloring[rl,rn](all_nodes) in
  let ghost_node_map : multicoloring⟨rn⟩ =
                          ghost_node_coloring[rl,rw,rn,rn](all_wires) in
  let wire_map : coloring(rw) = wire_coloring[rl,rw,rn,rn](all_wires) in

  −− Disjoint partition for the owned nodes of each piece
  partition rn using owned_node_map as rn0,rn1 in
  −− Aliased partition for ghost nodes of each piece
  partition rn using ghost_node_map.1 as rg0 in
  partition rn using ghost_node_map.2 as rg1 in
  −− Disjoint partition for the owned wires of each piece
  partition rw using wire_map as rw0,rw1 in

  −− Create region relationships for circuit pieces
  let lists0 : ⟨WireList⟨rl,rw0,rn0,rg0⟩@rl,NodeList⟨rl,rn0⟩@rl⟩ =
       ⟨ build_piece_wire_list[rl,rw,rn,rw0,rn0,rg0](all_wires),
        build_piece_node_list[rl,rn,rn0](all_nodes) ⟩ in
  let piece0 : CircuitPiece⟨rl,rw,rn⟩ =
       pack lists0 as CircuitPiece⟨rl,rw,rn⟩[rw0,rn0,rg0] in
  let lists1 : ⟨WireList⟨rl,rw1,rn1,rg1⟩@rl,NodeList⟨rl,rn1⟩@rl⟩ =
       ⟨ build_piece_wire_list[rl,rw,rn,rw1,rn1,rg1](all_wires),
        build_piece_node_list[rl,rn,rn1](all_nodes) ⟩ in
  let piece1 : CircuitPiece⟨rl,rw,rn⟩ =
       pack lists1 as CircuitPiece⟨rl,rw,rn⟩[rw1,rn1,rg1] in

  −− do actual (parallel) simulation
    execute_time_steps[rl,rw,rn](piece0,piece1,steps)

−− Time Step Loop
function execute_time_steps[rl,rw,rn] ( p0 : CircuitPiece⟨rl,rw,rn⟩,
    p1 : CircuitPiece⟨rl,rw,rn⟩, steps : int ) , reads(rn,rw,rl), writes(rn,rw) : bool =
  if steps < 1 then true else
  unpack p0 as piece0 : CircuitPiece⟨rl,rw,rn⟩[rw0,rn0,rg0] in
  unpack p1 as piece1 : CircuitPiece⟨rl,rw,rn⟩[rw1,rn1,rg1] in
  let _ : bool = calc_new_currents[rl,rw0,rn0,rg0](piece0.1) in
  let _ : bool = calc_new_currents[rl,rw1,rn1,rg1](piece1.1) in
  let _ : bool = distribute_charge[rl,rw0,rn0,rg0](piece0.1) in
  let _ : bool = distribute_charge[rl,rw1,rn1,rg1](piece1.1) in
  let _ : bool = update_voltage[rl,rn0](piece0.2) in
  let _ : bool = update_voltage[rl,rn1](piece1.2) in
    execute_time_steps[rl,rw,rn](p0,p1,steps−1)
```

Listing 2. Top-Level Application Code

```
function calc_new_currents[rl,rw,rn,rg] ( ptr_list : WireList⟨rl,rw,rn,rg⟩@rl ),
    reads(rl,rw,rn,rg), writes(rw) : bool =
  if isnull(ptr_list) then true else
  let wire_node : WireList⟨rl,rw,rn,rg⟩ = read(ptr_list) in
  let wire : CircuitWire⟨rn,rg⟩ = read(wire_node.1) in
  let in_node : CircuitNode = read(wire.1) in
  let out_node : CircuitNode = read(wire.2) in
  let current : int = (in_node.1 − out_node.1) / wire.3 in
  let new_wire : CircuitWire⟨rn,rg⟩ = ⟨wire.1,wire.2,wire.3,current⟩ in
  let _ : CircuitWire⟨rn,rg⟩@rw = write(wire_node.1, new_wire) in
    calc_new_currents[rl,rw,rn,rg](wire_node.2)

function distribute_charge[rl,rw,rn,rg] ( ptr_list : WireList⟨rl,rw,rn,rg⟩@rl ),
    reads(rl,rw,rn), reduces(reduce_charge,rn,rg), atomic(rn,rg) : bool =
  if isnull(ptr_list) then true else
  let wire_node : WireList⟨rl,rw,rn,rg⟩ = read(ptr_list) in
  let wire : CircuitWire⟨rn,rg⟩ = read(wire_node.1) in
  let _ : CircuitNode@rn = reduce(reduce_charge, wire.1, wire.4) in
  let _ : CircuitNode@(rn,rg) = reduce(reduce_charge, wire.2, wire.4) in
    distribute_charge[rl,rw,rn,rg](wire_node.2)

function update_voltage[rl,rn] ( ptr_list : NodeList⟨rl,rn⟩@rl ),
    reads(rl,rn), writes(rn) : bool =
  if isnull(ptr_list) then true else
  let node_ptr : CircuitNode@rn = read(ptr_list).1 in
  let _ : CircuitNode@rn =
    −− update voltage on a node
    let node : CircuitNode = read(node_ptr) in
    let voltage : int = (node.3 / node.4) in
    let new_node : CircuitNode = ⟨ voltage, node.2, node.3, node.4, node.5 ⟩ in
      write(node_ptr, new_node)
  in
  let next : NodeList⟨rl,rn⟩@rl = read(ptr_list).2 in
    update_voltage[rl,rn](next)

−− Reduction function for distribute charge
function reduce_charge ( node : CircuitNode, current : int ) : CircuitNode =
    let new_charge : int = node.3 + current in
      ⟨ node.1, node.2, new_charge, node.4, node.5 ⟩
```

Listing 3. Leaf Computation Tasks

```
function owned_node_coloring[rl,rn] ( node_list: NodeList⟨rl,rn⟩@rl ),
        reads(rl,rn) : coloring(rn) =
    if isnull(node_list) then
        newcolor rn
    else                          −− tuple fields accessed by .(field number)
        let list_elem : NodeList⟨rl,rn⟩ = read(node_list) in
        let part_coloring : coloring(rn) = owned_node_coloring[rl,rn](list_elem.2) in
        let node_ptr : CircuitNode@rn = list_elem.1 in
        let node : CircuitNode = read(node_ptr) in
        let piece_id_from_metis: int = node.5 in
            color(part_coloring, node_ptr, piece_id_from_metis)

function ghost_node_coloring[rl,rw,rn,rg] ( wire_list: WireList⟨rl,rw,rn,rg⟩@rl ),
        reads(rl,rw,rn,rg) : multicoloring⟨rn⟩ =
    if isnull(wire_list) then
        ⟨ newcolor rn, newcolor rn ⟩
    else                          −− tuple fields accessed by .(field number)
        let list_elem : WireList⟨rl,rw,rn,rg⟩ = read(wire_list) in
        let part_coloring : multicoloring⟨rn⟩ =
                        ghost_node_coloring[rl,rw,rn,rg](list_elem.2) in
        let wire_ptr : CircuitWire⟨rn,rg⟩@rw = list_elem.1 in
        let wire : CircuitWire⟨rn,rg⟩ = read(wire_ptr) in
        let in_node : CircuitNode = read(wire.1) in
        let out_node : CircuitNode = read(wire.2) in
        let in_piece_id : int = in_node.5 in
        let out_piece_id : int = out_node.5 in
        let id_not_equal : bool =
            if in_piece_id ⟨ out_piece_id then true else
            if out_piece_id ⟨ in_piece_id then true else false
        in
            if id_not_equal then
                if in_piece_id ⟨ 2 then
                    ⟨ color(part_coloring.1, downregion(wire.2, rn), 1), part_coloring.2 ⟩
                else
                    ⟨ part_coloring.1, color(part_coloring.2, downregion(wire.2, rn), 1) ⟩
            else
                ⟨ part_coloring.1, part_coloring.2 ⟩

function wire_coloring[rl,rw,rn,rg] ( wire_list: WireList⟨rl,rw,rn,rg⟩@rl ),
        reads(rl,rw,rn) : coloring(rw) =
    if isnull(wire_list) then
        newcolor rw
    else                          −− tuple fields accessed by .(field number)
        let list_elem : WireList⟨rl,rw,rn,rg⟩ = read(wire_list) in
        let part_coloring : coloring(rw) = wire_coloring[rl,rw,rn,rg](list_elem.2) in
        let wire_ptr : CircuitWire⟨rn,rg⟩@rw = list_elem.1 in
        let wire : CircuitWire⟨rn,rg⟩ = read(wire_ptr) in
        let node_ptr : CircuitNode@rn = wire.1 in
        let node : CircuitNode = read(node_ptr) in
        let piece_id_from_metis: int = node.5 in
            color(part_coloring, wire_ptr, piece_id_from_metis)
```

Listing 4. Coloring Functions

```
function build_piece_node_list[rl,rn,rpn] ( all_nodes : NodeList⟨rl,rn⟩@rl ),
      reads(rl), writes(rl) : NodeList⟨rl,rpn⟩@rl =
    if isnull(all_nodes) then
        null NodeList⟨rl,rpn⟩@rl
    else
        let list_elem : NodeList⟨rl,rn⟩ = read(all_nodes) in
        let part_list : NodeList⟨rl,rpn⟩@rl =
                        build_piece_node_list[rl,rn,rpn](list_elem.2) in
        let node_ptr : CircuitNode@rpn = downregion(list_elem.1, rpn) in
            if isnull(node_ptr) then
                part_list
            else
                let new_elem_ptr : NodeList⟨rl,rpn⟩@rl = new NodeList⟨rl,rpn⟩@rl in
                let new_elem : NodeList⟨rl,rpn⟩ = ⟨ node_ptr, part_list ⟩ in
                let _ : NodeList⟨rl,rpn⟩@rl = write(new_elem_ptr, new_elem) in
                    new_elem_ptr

function build_piece_wire_list[rl,rw,rn,rpw,rpn,rpg]
                        ( all_wires : WireList⟨rl,rw,rn,rn⟩@rl ),
      reads(rl,rpw), writes(rl,rpw) : WireList⟨rl,rpw,rpn,rpg⟩@rl =
    if isnull(all_wires) then
        null WireList⟨rl,rpw,rpn,rpg⟩@rl
    else
        let list_elem : WireList⟨rl,rw,rn,rn⟩ = read(all_wires) in
        let part_list : WireList⟨rl,rpw,rpn,rpg⟩@rl =
                    build_piece_wire_list[rl,rw,rn,rpw,rpn,rpg](list_elem.2) in
        let wire_ptr : CircuitWire⟨rn,rn⟩@rpw = downregion(list_elem.1, rpw) in
            if isnull(wire_ptr) then
                part_list
            else
                let old_wire : CircuitWire⟨rn,rn⟩ = read(wire_ptr) in
                let new_wire_ptr : CircuitWire⟨rpn,rpg⟩@rpw =
                                new CircuitWire⟨rpn,rpg⟩@rpw in
                let new_wire : CircuitWire⟨rpn,rpg⟩ = ⟨ downregion(old_wire.1, rpn),
                                downregion(old_wire.2, rpn, rpg),
                                old_wire.3, old_wire.4 ⟩ in
                let _ : CircuitWire⟨rpn,rpg⟩@rpw = write(new_wire_ptr, new_wire) in
                let new_elem_ptr : WireList⟨rl,rpw,rpn,rpg⟩@rl =
                                new WireList⟨rl,rpw,rpn,rpg⟩@rl in
                let new_elem : WireList⟨rl,rpw,rpn,rpg⟩ = ⟨ new_wire_ptr, part_list ⟩ in
                let _ : WireList⟨rl,rpw,rpn,rpg⟩@rl = write(new_elem_ptr, new_elem) in
                    new_elem_ptr
```

Listing 5. List-Building Helper Functions

B. Proofs

This appendix contains the full type system and operational semantics for Core Legion (recall Figure 2) and the proofs of all results. A subset of the type and operational semantics rules for Core Legion are given in Figure 4. The remaining type rules are given in Figure 12, and the remaining operational semantics rules are given in Figure 13.

B.1 Proof of Theorem 1

Recall from Section 6 that the proof of Theorem 1 is by induction on the structure of a compound expression. In this appendix, we give the details of the proof for each type of expression in the Core Legion language.

Proof.

- Case [E-Bool]. By assumption,

$$M, L, H, S, C \vdash bv \mapsto bv, []$$

and

$$\Gamma, \Phi, \Omega \vdash bv : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The only type rule that can apply is [T-Bool], so $T = bool$. Therefore $bv \sim_H M[\![bool]\!]$ and $[] \sim H$ and $[] :_M \Phi$ all follow trivially.

- Case [E-Int]. By assumption,

$$M, L, H, S, C \vdash iv \mapsto iv, []$$

and

$$\Gamma, \Phi, \Omega \vdash iv : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The only type rule that can apply is [T-Int], so $T = int$. Therefore $iv \sim_H M[\![int]\!]$ and $[] \sim H$ and $[] :_M \Phi$ all follow trivially.

- Case [E-Null]. By assumption,

$$M, L, H, S, C \vdash \text{null } T@r \mapsto null, []$$

and

$$\Gamma, \Phi, \Omega \vdash \text{null } T@r : T'$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The only type rule that can apply is [T-Null], so $T' = T@r$. Therefore $null \sim_H M[\![T@r]\!]$, $[] \sim H$ and $[] :_M \Phi$ all follow trivially.

- Case [E-MakeTuple]. By assumption,

$$M, L, H, S, C \vdash \langle e_1, e_2\rangle \mapsto \langle v_1, v_2\rangle, E'$$

and

$$\Gamma, \Phi, \Omega \vdash \langle e_1, e_2\rangle : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The only type rule that can apply is [T-MakeTuple], so $T = \langle T_1, T_2\rangle$.

For the subexpression e_1, from the form of rule [E-MakeTuple], we know that

$$M, L, H, S, C \vdash e_1 \mapsto v_1, E_1$$

From the form of rule [T-MakeTuple], we know that

$$\Gamma, \Phi, \Omega \vdash e_1 : T_1$$

Then, by induction, we have $v_1 \sim_H M[\![T_1]\!]$, $E_1 \sim H$ and $E_1 :_M \Phi$.
For the subexpression e_2, from the form of rule [E-MakeTuple], we know that

$$M, L, H, S', C \vdash e_2 \mapsto v_2, E_2$$

where $S' = apply(S, E_1)$. By Lemma 1, $S' = \text{apply}(S, E) \sim H$. From the form of rule [T-MakeTuple], we know that

$$\Gamma, \Phi, \Omega \vdash e_2 : T_2$$

Then, by induction, we have $v_2 \sim_H M[\![T_2]\!]$, $E_2 \sim H$ and $E_2 :_M \Phi$.
From the preceding steps we can conclude that $\langle v_1, v_2\rangle \sim_H M[\![\langle T_1, T_2\rangle]\!]$. Finally, from [E-MakeTuple] we also have

$$valid\_interleave(S, C, E', E_1, E_2)$$

and Lemma 3, $E' \sim H$ and by Lemma 5 $E' :_M \Phi$.

- Case [E-Tuple1]. By assumption,

$$M, L, H, S, C \vdash e.1 \mapsto v_1, E$$

and

$$\Gamma, \Phi, \Omega \vdash e.1 : T_1$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. From the form of [E-Tuple1] we know that

$$M, L, H, S, C \vdash e \mapsto \langle v_1, v_2\rangle, E$$

The only type rule that can apply is [T-Tuple1], and from the form of the rule we know

$$\Gamma, \Phi, \Omega \vdash e : \langle T_1, T_2\rangle$$

By induction, we have $\langle v_1, v_2\rangle \sim_H M[\![\langle T_1, T_2\rangle]\!]$, $E \sim H$ and $E :_M \Phi$. It follows immediately that $v_1 \sim_H M[\![T_1]\!]$ also holds.

- Case [E-Tuple2] is symmetric to case [E-Tuple1].
- Case[E-New]. By assumption,

$$M, L, H, S, C \vdash \text{new } T@r \mapsto l, []$$

and

$$\Gamma, \Phi, \Omega \vdash \text{new } T@r : T@r$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The hypothesis of the [E-New] rule is that $l \in M(r)$ and $H(l) = M[\![T]\!]$, which establishes $l \sim_H M[\![T]\!]$. The other two conclusions, $[] \sim H$ and $[] :_M \Phi$, follow trivially.

$$\Gamma, \Phi, \Omega \vdash bv : \mathit{bool} \quad \text{(T-BOOL)}$$

$$\Gamma, \Phi, \Omega \vdash iv : \mathit{int} \quad \text{(T-INT)}$$

$$\Gamma, \Phi, \Omega \vdash \text{null } T@r : T@r \quad \text{(T-NULL)}$$

$$\frac{\Gamma, \Phi, \Omega \vdash e_1 : T_1 \quad \ldots \quad \Gamma, \Phi, \Omega \vdash e_n : T_n}{\Gamma, \Phi, \Omega \vdash \langle e_1, \ldots, e_n \rangle : \langle T_1, \ldots, T_n \rangle} \text{(T-MAKETUPLE)}$$

$$\frac{\Gamma, \Phi, \Omega \vdash e : \langle T_1, \ldots, T_n \rangle}{\Gamma, \Phi, \Omega \vdash e.i : T_i} \quad \text{(T-TUPLE)}$$

$$\frac{\Gamma(id) = T}{\Gamma, \Phi, \Omega \vdash id : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, \Phi, \Omega \vdash e_1 : T_1 \quad \Gamma[T_1/id], \Phi, \Omega \vdash e_2 : T_2}{\Gamma, \Phi, \Omega \vdash \ : \text{let } id : T_1 = e_1 \text{ in } e_2 : T_2} \quad \text{(T-LET)}$$

$$\frac{\Gamma, \Phi, \Omega \vdash e_1 : \mathit{int} \quad \Gamma, \Phi, \Omega \vdash e_2 : \mathit{int}}{\Gamma, \Phi, \Omega \vdash e_1 + e_2 : \mathit{int}} \quad \text{(T-ADD)}$$

$$\frac{\Gamma, \Phi, \Omega \vdash e_1 : \mathit{int} \quad \Gamma, \Phi, \Omega \vdash e_2 : \mathit{int}}{\Gamma, \Phi, \Omega \vdash e_1 < e_2 : \mathit{bool}} \quad \text{(T-COMPARE)}$$

$$\frac{\Gamma, \Phi, \Omega \vdash e : T@(r_1, \ldots, r_n)}{\Gamma, \Phi, \Omega \vdash \text{isnull}(e) : \mathit{bool}} \quad \text{(T-ISNULL)}$$

$$\frac{\Gamma, \Phi, \Omega \vdash e_1 : \mathit{bool} \quad \Gamma, \Phi, \Omega \vdash e_2 : T \quad \Gamma, \Phi, \Omega \vdash e_3 : T}{\Gamma, \Phi, \Omega \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{(T-IFELSE)}$$

Figure 12. Legion Core Type System, Remaining Rules

- Case [E-Var]. By assumption,

$$M, L, H, S, C \vdash id \mapsto v, []$$

and

$$\Gamma, \Phi, \Omega \vdash id : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The hypothesis of the [E-Var] rule is $L(id) = v$. From $L \sim_H M[\![\Gamma]\!]$ we have $L(id) \sim_H M[\![\Gamma(id)]\!]$. The hypothesis of the [T-Var] rule is $\Gamma(id) = T$. Putting these three facts together we conclude $v \sim_H M[\![T]\!]$. The other two conclusions, $[] \sim H$ and $[] :_M \Phi$, follow trivially.

- Case [E-IsNull-F]. By assumption,

$$M, L, H, S, C \vdash \text{isnull}(e) \mapsto \mathit{false}, E$$

and

$$\Gamma, \Phi, \Omega \vdash \text{isnull}(e) : \mathit{bool}$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. From the form of [E-IsNull-F] we know that $M, L, H, S, C \vdash e \mapsto l, E$, and and from the form of the type rule [T-IsNull] we know $\Gamma, \Phi, \Omega \vdash e : T@(r_1, \ldots, r_n)$. Clearly $\mathit{false} \sim_H M[\![\mathit{bool}]\!]$, and $E \sim H$ and $E :_M \Phi$ follow by induction.

- Case [E-IsNull-T] is almost identical to case [E-IsNull-F], with l replaced by *null* and *false* replaced by *true*.

- Case [E-Let]. By assumption,

$$M, L, H, S, C \vdash \text{let } id : T_1 = e_1 \text{ in } e_2 \mapsto v_2, E'$$

and

$$\Gamma, \Phi, \Omega \vdash \text{let } id : T_1 = e_1 \text{ in } e_2 : T_2$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$.
For the subexpression e_1, from [E-Let], we know that

$$M, L, H, S, C \vdash e_1 \mapsto v_1, E_1$$

From type rule [T-Let], we know that

$$\Gamma, \Phi, \Omega \vdash e_1 : T_1$$

Then, by induction, we have $v_1 \sim_H M[\![T_1]\!]$, $E_1 \sim H$ and $E_1 :_M \Phi$.
For the subexpression e_2, from [E-Let], we have

$$M, L', H, S', C \vdash e_2 \mapsto v_2, E_2$$

where $L' = L[v_1/id]$ and $S' = \mathit{apply}(S, E_1)$. From [T-Let], we have $\Gamma[T_1/id], \Phi, \Omega \vdash e_2 : T_2$. By Lemma 1, $S' = \text{apply}(S, E) \sim H$. From $L \sim_H M[\![\Gamma]\!]$ and $v_1 \sim_H M[\![T_1]\!]$ we conclude $L[v_1/id] \sim_H M[\![\Gamma[T_1/id]]\!]$. Then, by induction, we have $v_2 \sim_H M[\![T_2]\!]$, $E_2 \sim H$ and $E_2 :_M \Phi$.
Finally, from [E-Let] we also have

$$\mathit{valid\_interleave}(S, C, E', E_1, E_2)$$

Then, by Lemma 3, $E' \sim H$ and by Lemma 5 $E' :_M \Phi$.

- Case [E-Add]. By assumption,

$$M, L, H, S, C \vdash e_1 + e_2 \mapsto v', E'$$

$$M, L, H, S, C \vdash bv \mapsto bv, [] \quad \text{(E-BOOL)}$$

$$M, L, H, S, C \vdash iv \mapsto iv, [] \quad \text{(E-INT)}$$

$$M, L, H, S, C \vdash \text{null } T@r \mapsto null, [] \quad \text{(E-NULL)}$$

$$\frac{M, L, H, S, C \vdash e \mapsto \langle v_1, \ldots, v_n \rangle, E}{M, L, H, S, C \vdash e.i \mapsto v_i, E} \quad \text{(E-TUPLE)}$$

$$\frac{\begin{array}{l} M, L, H, S, C \vdash e_1 \mapsto v_1, E_1 \qquad S_1 = apply(S, E_1) \\ \ldots \\ M, L, H, S_{n-1}, C \vdash e_n \mapsto v_n, E_n \\ valid\_interleave(S, C, E', E_1, \ldots, E_n) \end{array}}{M, L, H, S, C \vdash \langle e_1, \ldots, e_n \rangle \mapsto \langle v_1, \ldots, v_n \rangle, E'} \quad \text{(E-MAKETUPLE)}$$

$$\frac{L(id) = v}{M, L, H, S, C \vdash id \mapsto v, []} \quad \text{(E-VAR)}$$

$$\frac{\begin{array}{l} M, L, H, S, C \vdash e_1 \mapsto v_1, E_1 \\ L' = L[v_1/id] \\ S' = \text{apply}(S, E_1) \\ M, L', H, S', C \vdash e_2 \mapsto v_2, E_2 \\ valid\_interleave(S, C, E', E_1, E_2) \end{array}}{M, L, H, S, C \vdash \text{let } id : T = e_1 \text{ in } e_2 \mapsto v_2, E'} \quad \text{(E-LET)}$$

$$\frac{\begin{array}{l} M, L, H, S, C \vdash e_1 \mapsto v_1, E_1 \\ S' = \text{apply}(S, E_1) \\ M, L, H, S', C \vdash e_2 \mapsto v_2, E_2 \\ v' = v_1 + v_2 \\ valid\_interleave(S, C, E', E_1, E_2) \end{array}}{M, L, H, S, C \vdash e_1 + e_2 \mapsto v', E'} \quad \text{(E-ADD)}$$

$$\frac{\begin{array}{l} M, L, H, S, C \vdash e_1 \mapsto v_1, E_1 \\ S' = \text{apply}(S, E_1) \\ M, L, H, S', C \vdash e_2 \mapsto v_2, E_2 \\ v' = \begin{cases} true, & \text{if } v_1 < v_2. \\ false, & \text{otherwise.} \end{cases} \\ valid\_interleave(S, C, E', E_1, E_2) \end{array}}{M, L, H, S, C \vdash e_1 < e_2 \mapsto v', E'} \quad \text{(E-COMPARE)}$$

$$\frac{M, L, H, S, C \vdash e \mapsto l, E}{M, L, H, S, C \vdash \text{isnull}(e) \mapsto false, E} \quad \text{(E-ISNULL-F)}$$

$$\frac{M, L, H, S, C \vdash e \mapsto null, E}{M, L, H, S, C \vdash \text{isnull}(e) \mapsto true, E} \quad \text{(E-ISNULL-T)}$$

$$\frac{\begin{array}{l} M, L, H, S, C \vdash e_1 \mapsto false, E_1 \\ S' = \text{apply}(S, E_1) \\ M, L, H, S', C \vdash e_3 \mapsto v_3, E_3 \end{array}}{M, L, H, S, C \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mapsto v_3, E_1 +\!\!+ E_3} \quad \text{(E-IFELSE-F)}$$

$$\frac{\begin{array}{l} M, L, H, S, C \vdash e_1 \mapsto true, E_1 \\ S' = \text{apply}(S, E_1) \\ M, L, H, S', C \vdash e_2 \mapsto v_2, E_2 \end{array}}{M, L, H, S, C \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mapsto v_2, E_1 +\!\!+ E_2} \quad \text{(E-IFELSE-T)}$$

Figure 13. Legion Core Operational Semantics, Remaining Rules

and

$$\Gamma, \Phi, \Omega \vdash e_1 + e_2 : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The only type rule that can apply is [T-Add], so $T = int$.
For the subexpression e_1, from [E-Add] we know

$$M, L, H, S, C \vdash e_1 \mapsto v_1, E_1$$

From [T-Add], we know that

$$\Gamma, \Phi, \Omega \vdash e_1 : int$$

Then, by induction, we have $v_1 \sim_H M[\![T_1]\!]$, $E_1 \sim H$ and $E_1 :_M \Phi$.

For the subexpression e_2, from [E-Add], we know

$$M, L, H, S', C \vdash e_2 \mapsto v_2, E_2$$

where $S' = apply(S, E_1)$. By Lemma 1, $S' = \text{apply}(S, E) \sim H$. From [T-Add] we know

$$\Gamma, \Phi, \Omega \vdash e_2 : int$$

Then, by induction, we have $v_2 \sim_H M[\![T_2]\!]$, $E_2 \sim H$ and $E_2 :_M \Phi$.
It is immediate that $v_1 + v_2 \sim_H M[\![int]\!]$. Finally, from [E-Add] we also have

$$valid\_interleave(S, C, E', E_1, E_2)$$

and Lemma 3, $E' \sim H$ and by Lemma 5, $E' :_M \Phi$.

- Case [E-Compare] is isomorphic to [E-Add].
- Case [E-IfElse-T]. By assumption,

$$M, L, H, S, C \vdash \text{ if } e_1 \text{ then } e_2 \text{ else } e_3 \mapsto v_2, E_1 +\!\!+ E_2$$

and

$$\Gamma, \Phi, \Omega \vdash \text{ if } e_1 \text{ then } e_2 \text{ else } e_3 : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$.
For the subexpression e_1, from [E-IfElse-T] we know

$$M, L, H, S, C \vdash e_1 \mapsto \mathit{true}, E_1$$

From [T-IfElse], we know that

$$\Gamma, \Phi, \Omega \vdash e_1 : \mathit{bool}$$

Clearly $\mathit{true} \sim_H M[\![\mathit{bool}]\!]$, and by induction we have $E_1 \sim H$ and $E_1 :_M \Phi$.
For the subexpression e_2, from [E-IfElse-T] we know

$$M, L, H, S', C \vdash e_2 \mapsto v_2, E_2$$

where $S' = \mathit{apply}(S, E_1)$. By Lemma 1, $S' = \text{apply}(S, E) \sim H$. From [T-IfElse-T] we know

$$\Gamma, \Phi, \Omega \vdash e_2 : T$$

Then, by induction, we have $v_2 \sim_H M[\![T]\!]$, $E_2 \sim H$ and $E_2 :_M \Phi$.
Finally, by Lemma 2, $E_1 +\!\!+ E_2 \sim H$ and by Lemma 4, $E_1 +\!\!+ E_2 :_M \Phi$.
- Case [E-IfElse-F] is nearly the same as [E-IfElse-T] with *false* replacing *true* and e_3 replacing e_2.
- Case [E-Newcolor]. By assumption,

$$M, L, H, S, C \vdash \text{newcolor } r \mapsto K, []$$

where $K = \{(l_1, iv_1), \ldots, (l_p, iv_p)\}$ such that $l_i \in M(r)$ for all i and $l_1, \ldots, l_p$ are pairwise distinct locations. In addition, we have the assumptions

$$\Gamma, \Phi, \Omega \vdash \text{newcolor } r : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The only type rule that can apply is [T-Newcolor], so we can infer that $T = \text{coloring}(r)$. From the structure of K it follows that $K \sim_H M[\![\text{coloring(r)}]\!]$. The other two conclusions, $[] \sim H$ and $[] :_M \Phi$, follow trivially.
- Case [E-Color]. By assumption,

$$M, L, H, S, C \vdash \text{color}(e_1, e_2, e_3) \mapsto K', E'$$

and

$$\Gamma, \Phi, \Omega \vdash \text{color}(e_1, e_2, e_3) : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The only type rule that can apply is [T-Color], so we know $T = \text{coloring}(r)$.

For the subexpression e_1, from [E-Color] we know

$$M, L, H, S, C \vdash e_1 \mapsto K, E_1$$

From [T-Color], we know that

$$\Gamma, \Phi, \Omega \vdash e_1 : \text{coloring}(r)$$

By induction it follows that $K \sim_H M[\![\text{coloring(r)}]\!]$ and $E_1 \sim H$ and $E_1 :_M \Phi$.
For the subexpression e_2, from [E-Color] we know

$$M, L, H, S', C \vdash e_2 \mapsto l, E_2$$

where $S' = \text{apply}(S, E_1)$. From [T-Color], we know that

$$\Gamma, \Phi, \Omega \vdash e_2 : T'@r$$

By Lemma 1, $S' \sim H$. By induction, $l \sim_H M[\![T'@r]\!]$ and $E_2 \sim H$ and $E_2 :_M \Phi$.
For the subexpression e_3, from [E-Color] we know

$$M, L, H, S'', C \vdash e_3 \mapsto v, E_3$$

where $S'' = \text{apply}(S', E_2)$. From [T-Color], we know that

$$\Gamma, \Phi, \Omega \vdash e_3 : \mathit{int}$$

By Lemma 1, $S'' \sim H$. By induction, $v \sim_H M[\![\mathit{int}]\!]$ and $E_3 \sim H$ and $E_3 :_M \Phi$.
Now K' is built from K by extending K with a mapping for location l. First, because $l \sim_H M[\![T'@r]\!]$ and $K \sim_H M[\![\text{coloring}(r)]\!]$ and because K' excludes any previous mapping for l (guaranteeing the the coloring is a function), we conclude $K' \sim_H M[\![\text{coloring}(r)]\!]$. From [E-Color], we also know

$$\mathit{valid\_interleave}(S, C, E', E_1, E_2, E_3)$$

and by applying Lemma 3 twice we can conclude $E' \sim H$. Similarly, by applying Lemma 5 twice we can conclude $E' :_M \Phi$.

- Case [E-UpRgn]. By assumption,

$$M, L, H, S, C \vdash \text{upregion}(e, r_1, \ldots, r_n) \mapsto v, E$$

and

$$\Gamma, \Phi, \Omega \vdash \text{upregion}(e, r_1, \ldots, r_n) : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The type rule must be [T-UpRgn], so we also know $T = T'@(r_1, \ldots, r_n)$.
From [E-UpRgn] we know

$$M, L, H, S, C \vdash e \mapsto v, E$$

and from [T-UpRgn] we know

$$\Gamma, \Phi, \Omega \vdash e : T'@(r'_1, \ldots, r'_k)$$

By induction, we can conclude that $v \sim_H M[\![T'@(r'_1, \ldots, r'_k)]\!]$, $E \sim H$ and $E :_M \Phi$.
To show $v \sim_H M[\![T'@(r_1, \ldots, r_n)]\!]$, we first observe that from $v \sim_H M[\![T'@(r'_1, \ldots, r'_k)]\!]$ we know there is an i such that $v \in M(r'_i)$ and $H(v) = \hat{T}'$. From [T-UpRgn] we know $\forall i.\exists j.r'_i \leq r_j \in \Omega^*$. From Lemma 6, we have $M \sim \Omega^*$, so there must be some r_j such that $M(r'_i) \subseteq M(r_j)$. Then $v \in M(r_j)$ and $v \sim_H M[\![T'@(r_1, \ldots, r_n)]\!]$.

- Case [E-DownRgn]. By assumption,

$$M, L, H, S, C \vdash \text{downregion}(e, r_1, \ldots, r_n) \mapsto v, E$$

and

$$\Gamma, \Phi, \Omega \vdash \text{downregion}(e, r_1, \ldots, r_n) : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$. The type rule must be [T-DownRgn], so we also know $T = T'@(r_1, \ldots, r_n)$.
From [E-DownRgn] we know

$$M, L, H, S, C \vdash e \mapsto l, E$$

and from [T-DownRgn] we know

$$\Gamma, \Phi, \Omega \vdash e : T'@(r'_1, \ldots, r'_k)$$

By induction, we can conclude that $l \sim_H M[\![T'@(r'_1, \ldots, r'_k)]\!]$, $E \sim H$ and $E :_M \Phi$.
To show $v \sim_H M[\![T'@(r_1, \ldots, r_n)]\!]$ there are two cases. If $v = null$ then $null \sim_H M[\![T'@(r_1, \ldots, r_n)]\!]$ follows by definition. Otherwise, $v = l$ and $l \in M(r_i)$ for some i. We also know $H(v) = \hat{T}'$ because $l \sim_H M[\![T'@(r'_1, \ldots, r'_k)]\!]$. Therefore $v \sim_H M[\![T'@(r_1, \ldots, r_n)]\!]$.

- Case [E-Read]. By assumption,

$$M, L, H, S, C \vdash \text{read}(e) \mapsto v, E +\!\!+ [\mathit{read}(l, \mathit{excl}, v, 0)]$$

and

$$\Gamma, \Phi, \Omega \vdash \text{read}(e) : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$.
From [E-Read] we have

$$M, L, H, S, C \vdash e \mapsto l, E$$

and from [T-Read] we know

$$\Gamma, \Phi, \Omega \vdash e : T@(r_1, \ldots, r_n)$$

By induction, we can conclude that $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$, $E \sim H$ and $E :_M \Phi$.
To show $v \sim_H M[\![T]\!]$ there are two cases. If $l \notin C$, then $v = S'(l)$. Since $S' \sim H$ from Lemma 1, we have $S'(l) \sim_H H(l)$. If $l \in C$, we have $v = v' \sim_H H(l)$ from [E-Read] directly. In both cases, $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$ gives us $H(l) = \hat{T}$ and therefore $v \sim_H M[\![T]\!]$. Next, $E +\!\!+ [\mathit{read}(l, \mathit{excl}, v, 0)] \sim H$ follows immediately from $E \sim H$ as any read operation is consistent (the restriction is only on writes and reductions).
Finally, we must show $E +\!\!+ [\mathit{read}(l, \mathit{excl}, v, 0)] :_M \Phi$. Since we have already shown $E :_M \Phi$, it suffices to show $[\mathit{read}(l, \mathit{excl}, v, 0)] :_M \Phi$. By $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$, there is some r_i such that $l \in M(r_i)$. Furthermore, from [T-Read] we have $\mathit{reads}(r_i) \in \Phi^*$, which shows $[\mathit{read}(l, \mathit{excl}, v, 0)] :_M \Phi^*$. The result follows from Lemma 7.

- Case [E-Write]. By assumption,

$$M, L, H, S, C \vdash \text{write}(e_1, e_2) \mapsto l, E' +\!\!+ [\mathit{write}(l, \mathit{excl}, v, 0)]$$

and

$$\Gamma, \Phi, \Omega \vdash \text{write}(e_1, e_2) : T@(r_1, \ldots, r_n)$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$.
For subexpression e_1, From [E-Write] we have

$$M, L, H, S, C \vdash e_1 \mapsto l, E_1$$

and from [T-Write] we know

$$\Gamma, \Phi, \Omega \vdash e_1 : T@(r_1, \ldots, r_n)$$

. By induction, we can conclude that $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$, $E_1 \sim H$ and $E_1 :_M \Phi$.
For subexpression e_2, From [E-Write] we have

$$M, L, H, S', C \vdash e_2 \mapsto v, E_2$$

where $S' = \text{apply}(S, E_1)$. From [T-Write] we know $\Gamma, \Phi, \Omega \vdash e_2 : T$. Using Lemma 1 it follows that $S' \sim H$. Then by induction, we can conclude that $v \sim_H M[\![T]\!]$, $E_2 \sim H$ and $E_2 :_M \Phi$.
We have already shown $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$, so it only remains to show $E' +\!\!+ [\mathit{write}(l, \mathit{excl}, v, 0)] \sim H$ and $E' +\!\!+ [\mathit{write}(l, \mathit{excl}, v, 0)] :_M \Phi$. From [E-Write] we have

$$\mathit{valid\_interleave}(S, C, E', E_1, E_2)$$

and therefore by Lemma 3 we have $E' \sim H$. To show $[\mathit{write}(l, \mathit{excl}, v, 0)] \sim H$ we must show $v \sim_H M[\![H(l)]\!]$, which follows from $v \sim_H M[\![T]\!]$ and $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$ and $S \sim H$. Thus, $E' +\!\!+ [\mathit{write}(l, \mathit{excl}, v, 0)] \sim H$.
To show $E' +\!\!+ [\mathit{write}(l, \mathit{excl}, v, 0)] :_M \Phi$, we first observe that $\mathit{valid\_interleave}(S, C, E', E_1, E_2)$ and Lemma 5 imply $E' :_M \Phi$. To show $[\mathit{write}(l, \mathit{excl}, v, 0)] :_M \Phi$, we observe that because $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$, there is some r_i such that $l \in M(r_i)$. Furthermore, from [T-Write] we have $\mathit{writes}(r_i) \in \Phi^*$, which shows $[\mathit{write}(l, \mathit{excl}, v, 0)] :_M \Phi^*$. The result follows from Lemma 7.

- Case[E-Reduce]. By assumption,

$$M, L, H, S, C \vdash \text{reduce}(id, e_1, e_2) \mapsto l, E' +\!\!+ [\mathit{reduce}_{id}(l, \mathit{excl}, v, 0)]$$

and

$$\Gamma, \Phi, \Omega \vdash \text{reduce}(id, e_1, e_2) : T_1@(r_1, \ldots, r_n)$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$.
For subexpression e_1, From [E-Reduce] we have

$$M, L, H, S, C \vdash e_1 \mapsto l, E_1$$

and from [T-Reduce] we know

$$\Gamma, \Phi, \Omega \vdash e_1 : T@(r_1, \ldots, r_n)$$

. By induction, we can conclude that $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$, $E_1 \sim H$ and $E_1 :_M \Phi$.
For subexpression e_2, From [E-Reduce] we have

$$M, L, H, S', C \vdash e_2 \mapsto v, E_2$$

where $S' = \text{apply}(S, E_1)$. From [T-Reduce] we know

$$\Gamma, \Phi, \Omega \vdash e_2 : T_2$$

Using Lemma 1 it follows that $S' \sim H$. Then by induction, we can conclude that $v \sim_H M[\![T_2]\!]$, $E_2 \sim H$ and $E_2 :_M \Phi$.
We have already shown $l \sim_H M[\![T_1@(r_1, \ldots, r_n)]\!]$, so it only remains to show $E' +\!\!+ [\mathit{reduce}_{id}(l, \mathit{excl}, v, 0)] \sim H$ and $E' +\!\!+ [\mathit{reduce}_{id}(l, \mathit{excl}, v, 0)] :_M \Phi$. From [E-Reduce] we have $\mathit{valid\_interleave}(S, C, E', E_1, E_2)$ and therefore by Lemma 3 we have $E' \sim H$. To show $[\mathit{reduce}_{id}(l, \mathit{excl}, v, 0)] \sim H$ we must show three things. First, we must show $M[\![\Gamma]\!](id) = (\hat{T}_1, \hat{T}_2), \emptyset, \emptyset \to \hat{T}_1$ which follows from the assumption $\Gamma(id) = (T_1, T_2), \emptyset, \emptyset \to T_1$ in [T-Reduce]. Second, we must show $H(l) = \hat{T}_1$, which follows from $l \sim_H M[\![T_1@(r_1, \ldots, r_n)]\!]$. Finally, we have already shown $v \sim_H M[\![\hat{T}_2]\!]$. Thus,

$$E' +\!\!+ [\mathit{reduce}_{id}(l, \mathit{excl}, v, 0)] \sim H$$

To show

$$E' +\!\!+ [\mathit{reduce}_{id}(l, \mathit{excl}, v, 0)] :_M \Phi$$

we first observe that $\mathit{valid\_interleave}(S, C, E', E_1, E_2)$ and Lemma 5 imply $E' :_M \Phi$. To show $[\mathit{reduce}_{id}(l, \mathit{excl}, v, 0)] :_M \Phi$, we observe that because $l \sim_H M[\![T@(r_1, \ldots, r_n)]\!]$, there is some r_i such that $l \in M(r_i)$. Furthermore, from [T-Reduce] we have $\mathit{reduce}_{id}(r_i) \in \Phi^*$, which shows $[\mathit{reduce}_{id}(l, \mathit{excl}, v, 0)] :_M \Phi^*$. The result follows from Lemma 7.

- Case [E-Pack]. By assumption,

$$M, L, H, S, C \vdash \text{pack } e_1 \text{ as } T_1[r_1, \ldots, r_k] \mapsto v', E$$

and

$$\Gamma, \Phi, \Omega \vdash \text{pack } e_1 \text{ as } T_1[r_1, \ldots, r_k] : T_1$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$.
From [E-Pack] we have $M, L, H, S, C \vdash e_1 \mapsto v, E$ and from [T-Pack] we know $\Gamma, \Phi, \Omega \vdash e_1 : T_2[r_1/r'_1, \ldots, r_k/r'_k]$.
By induction, we can conclude that $v \sim_H M[\![T_2[r_1/r'_1, \ldots, r_k/r'_k]]\!]$, $E \sim H$ and $E :_M \Phi$.
The only thing remaining to show is $v' \sim_H M[\![T_1]\!]$. From [E-Pack] we have $v' = \langle\langle \rho_1, \ldots, \rho_k, v \rangle\rangle$ and $M(r_i) = \rho_i$. From [T-Pack] we have $T_1 = \exists r'_1, \ldots, r'_n.\ T_2$ where Ω_1 and $\Omega_1[r_1/r'_1, \ldots, r_k/r'_k] \subseteq \Omega^*$.
Let $M[\![T_1]\!] = \exists r'_1, \ldots, r'_n.\ \hat{T}_2$ where $\hat{\Omega}_1$. We require that $v \sim_H \hat{T}_2[\rho_1/r'_1, \ldots, \rho_k/r'_k]$ and $\{(r'_i, \rho_i)\} \sim \hat{\Omega}_1$. The right hand side of the first condition can be rewritten as $\hat{T}_2[M(r_1)/r'_1, \ldots, M(r_k)/r'_k]$, which is equal to $M[\![T_2[r_1/r'_1, \ldots, r_k/r'_k]]\!]$.
Similarly, observe that

$$\hat{\Omega}_1[\rho_1/r'_1, \ldots, \rho_k/r'_k] = M[\![\Omega_1[r_1/r'_1, \ldots, r_k/r'_k]]\!]$$

With two applications of Lemma 8, we have

$$\{(r'_i, \rho_i)\} \sim \hat{\Omega}_1 \Leftrightarrow M \sim \Omega_1[r_1/r'_1, \ldots, r_k/r'_k]$$

The latter constraints are a subset of Ω^*, and must be satisfied if $M \sim \Omega^*$, which we have from Lemma 6.

- Case [E-Unpack]. By assumption,

$$M, L, H, S, C \vdash \text{unpack } e_1 \text{ as } id : T_1[r_1, \ldots, r_k] \text{ in } e_2 \mapsto v_2, E'$$

and

$$\Gamma, \Phi, \Omega \vdash \text{unpack } e_1 \text{ as } id : T_1[r_1, \ldots, r_k] \text{ in } e_2 : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$.
For subexpression e_1, from [E-UnPack] we have

$$M, L, H, S, C \vdash e_1 \mapsto \langle\langle \rho_1, \ldots, \rho_k, v_1 \rangle\rangle, E$$

and from [T-Unpack] we know

$$\Gamma, \Phi, \Omega \vdash e_1 : T_1 = \exists r'_1, \ldots, r'_n . T_2 \text{ where } \Omega_1$$

By induction, we can conclude that $\langle\langle \rho_1, \ldots, \rho_k, v \rangle\rangle \sim_H M[\![T_1]\!]$ and $E \sim H$ and $E :_M \Phi$.
For subexpression e_2, from [E-UnPack] we have

$$M', L', H, S', C \vdash e_2 \mapsto v_2, E_2$$

where $S' = \text{apply}(S, E_2)$, $L' = L[v_1/id]$ and $M' = M[\rho_1/r_1, \ldots, \rho_k/r_k]$. From [T-Unpack] we have

$$\Gamma', \Phi, \Omega' \vdash e_2 : T_3$$

where $\Gamma' = \Gamma[T_2[r_1/r'_1, \ldots, r_k/r'_k]/id]$ and $\Omega' = \Omega \cup \Omega_1[r_1/r'_1, \ldots, r_k/r'_k]$. To establish the induction hypothesis for e_2, we must show $M' \sim \Omega'$ and $L' \sim_H M'[\![\Gamma']\!]$ and $S' \sim H$.
We can show $M' \sim \Omega'$ by showing both $M' \sim \Omega$ and $M' \sim \Omega_1[r_1/r'_1, \ldots, r_k/r'_k]$. The former follows directly from $M \sim \Omega$ as M' does not change any existing mappings in M. The latter requires an argument similar to the [E-Pack] case. We let $M[\![T_1]\!] = \exists r'_1, \ldots, r'_n.\ \hat{T}_2$ where $\hat{\Omega}_1$,

and conclude that $\{\rho_1/r'_1, \ldots, \rho_k/r'_k\} \sim \hat{\Omega}_1$. From Lemma 8 we have $\emptyset \sim \hat{\Omega}_1[\rho_1/r'_1, \ldots, \rho_k/r'_k]$. Finally, we observe that

$$M'[\![\Omega_1[r_1/r'_1, \ldots, r_k/r'_k]]\!] = \hat{\Omega}_1[\rho_1/r'_1, \ldots, \rho_k/r'_k]$$

and conclude $M' \sim \Omega'$ with another application of Lemma 8.
To show $L' \sim_H M'[\![\Gamma']\!]$, we must show $L'(x) \sim_H M'[\![\Gamma'(x)]\!]$ for all identifiers x in the domain of Γ'. For any $id' \neq id$, the statement is equivalent to showing $L(x) \sim_H M'[\![\Gamma(x)]\!]$, which must hold because M' does does not change any existing binding in M. For id, we have $v_1 \sim_H \hat{T}_2[\rho_1/r'_1, \ldots, \rho_k/r'_k]$ and again observe that since $r'_1, \ldots, r'_k$ do not appear in M, the right hand side is equal to $M'[\![T_2[r_1/r'_1, \ldots, r_k/r'_k]]\!]$.
Finally, as in previous cases, $S' \sim H$ follows from Lemma 1. Thus, we can conclude that $v_2 \sim_H M'[\![T_3]\!]$ and $E_2 \sim H$ and $E_2 :_M \Phi$.
Because T_3 cannot include any of the unpacked regions in it, $v_2 \sim_H M[\![T_3]\!]$. From [E-Unpack] we have

$$valid\_interleave(S, C, E', E_1, E_2)$$

and using Lemma 3 we prove $E' \sim H$. To finish the case, we note that from

$$E_1 :_M \Phi$$

$$E_2 :_M \Phi$$

$$valid\_interleave(S, C, E', E_1, E_2)$$

and using Lemma 5 we can conclude $E' :_M \Phi$.

- Case [E-Part]. By assumption,

$$M, L, H, S, C \vdash \text{partition } r_p \text{ using } e_1 \text{ as } r_1, \ldots, r_k \text{ in } e_2 \mapsto v, E'$$

and

$$\Gamma, \Phi, \Omega \vdash \text{partition } r_p \text{ using } e_1 \text{ as } r_1, \ldots, r_k \text{ in } e_2 : T$$

and $M \sim \Omega$ and $L \sim_H M[\![\Gamma]\!]$ and $S \sim H$.
For subexpression e_1, from [E-Part] we have

$$M, L, H, S, C \vdash e_1 \mapsto K, E_1$$

and from [T-Part] we have

$$\Gamma, \Phi, \Omega \vdash e_1 : \text{coloring}(r_p)$$

By induction, we conclude that $K \sim_H M[\![\text{coloring}(r_p)]\!]$ and $E_1 \sim H$ and $E_1 :_M \Phi$. For subexpression e_2, from [E-Part] we have

$$M', L, H, S', C \vdash e_2 \mapsto v, E_2$$

where $M' = M[\rho_1/r_1, \ldots, \rho_k/r_k]$ and $S' = \text{apply}(S, E_1)$.
From [T-Part] we have

$$\Gamma, \Phi, \Omega' \vdash e_2 : T$$

where $\Omega' = \Omega \wedge \bigwedge_{i \in [1,k]} r_i \leq r_p \wedge \bigwedge_{1 \leq i < j \leq k} r_i * r_j$. To establish the induction hypothesis for e_2, we must show $M' \sim \Omega'$ and $L \sim_H M'[\![\Gamma]\!]$ and $S' \sim H$.
For $M' \sim \Omega'$, we observe that all of the constraints are either present previously (hold in $M \sim \Omega$) or are established explicitly by the partition operation. Note that the type of the coloring carries the name of the region being partitioned and that all introduced region names are not in use in M. For $L \sim_H M'[\![\Gamma]\!]$, we simply observe that $L \sim_H M[\![\Gamma]\!]$ holds and M' is an extension of M. Finally, as in previous cases, $S' \sim H$ follows from Lemma 1.
Therefore, we can conclude by induction $v \sim_H M'[\![T]\!]$ and $E_2 \sim H$ and $E_2 :_M \Phi$. Because T may not mention any of the regions introduced in the partitioning, we have $v \sim_H M[\![T]\!]$. From [E-Part] we have

$$valid\_interleave(S, C, E', E_1, E_2)$$

and using Lemma 3 we prove $E' \sim H$. To finish the case, we note that from

$$E_1 :_M \Phi$$

$$E_2 :_M \Phi$$

$$valid\_interleave(S, C, E', E_1, E_2)$$

and using Lemma 5 we can conclude $E' :_M \Phi$.

- Case [E-Call]. From [E-Call] we have

$$M, L, H, S, C \vdash id[r_1, \ldots, r_k](e_1, \ldots, e_n) \mapsto v_{n+1}, E''$$

and from [T-Call] we have

$$\Gamma, \Phi, \Omega \vdash id[r_1, \ldots, r_k](e_1, \ldots, e_n) : T_r[r_1/r'_1, \ldots, r_k/r'_k]$$

For subexpression e_1, from [E-Call] we have

$$M, L, H, S, C \vdash e_1 \mapsto v_1, E_1$$

and from [T-Call] we have

$$\Gamma, \Phi, \Omega \vdash e_1 : T_1[r_1/r'_1, \ldots, r_k/r'_k]$$

By induction, $v_1 \sim_H M[\![T_1[r_1/r'_1, \ldots, r_k/r'_k]]\!]$, $E_1 \sim H$ and $E_1 :_M \Phi$.
For the other subexpressions that are arguments to the call the reasoning is very similar; we show only the last argument e_n. First,

$$S_n = \text{apply}(E_{n-1}, \text{apply}(E_{n-2}, \ldots \text{apply}(E_1, S) \ldots)$$

An induction on the number of applications of *apply* using Lemma 1 shows that $S_n \sim H$. We also have

$$M, L, H, S, C \vdash e_n \mapsto v_n, E_n$$

from [E-Call] and

$$\Gamma, \Phi, \Omega \vdash e_n : T_n[r_1/r'_1, \ldots, r_k/r'_k]$$

from [T-Call]. By induction, $v_n \sim_H M[\![T_n[r_1/r'_1, \ldots, r_k/r'_k]]\!]$, $E_n \sim H$ and $E_n :_M \Phi$.
Now, from [E-Call] we have

$$valid\_interleave(S, C, E', E_1, \ldots, E_n)$$

and furthermore

$$M' = \{(r'_1, M(r_1)), \ldots (r'_k, M(r_k))\}$$

$$L' = \{(a_1, v_1), \ldots, (a_n, v_n)\}$$

$$S' = \text{apply}(S, E')$$

We also know from [E-Call] that

$$M', L', H, S', C' \vdash e_{n+1} \mapsto v_{n+1}, E_{n+1}$$

where e_{n+1} is the body of the called function. From [T-Program] we know

$$\Gamma', \Phi', \emptyset \vdash e_{n+1} : T_r$$

where $\Gamma'(a_i) = T_i$. To apply the induction hypothesis to e_{n+1} we want to show

1. $S' \sim H$,
2. $M' \sim \emptyset$, and
3. $L' \sim_H M'[\![\Gamma']\!]$

For (1), using $E_i \sim H$ for $1 \leq i \leq n$ and applying Lemma 3 $n-1$ times we can conclude that $E' \sim H$. Then because $S \sim H$ and $S' = \text{apply}(S, E')$ by Lemma 1 we have $S' \sim H$.
Part (2) is immediate by the definition of mapping consistency.
For (3), in previous steps we have shown

$$v_i \sim_H M[\![T_i[r_1/r'_1, \ldots, r_k/r'_k]]\!]$$

for $1 \leq i \leq n$. Since $M[\![r_i]\!] = M'[\![r'_i]\!]$ it follows that $M[\![T_i[r_1/r'_1, \ldots, r_k/r'_k]\!] = M'[\![T_i]\!]$ and therefore $v_i \sim_H M'[\![T_i]\!]$, from which the result follows.
Thus, by induction, we conclude that $v_{n+1} \sim_H M'[\![T_r]\!]$, $E_{n+1} \sim H$ (under M') and $E_{n+1} :_{M'} \Phi'$. Because

$$M[\![T_r[r_1/r'_1, \ldots, r_k/r'_k]\!] = M'[\![T_r]\!]$$

we can conclude $v_{n+1} \sim_H M[\![T_r[r_1/r'_1, \ldots, r_k/r'_k]\!]$.
To show that $E_{n+1} \sim H$ under M' implies $E_{n+1} \sim H$ under M, note that $H(\cdot)$ does not depend on the mapping (because it includes only physical, not logical, regions) so the types of locations are unaffected by the mapping (see Figure 6). Now, the environment $M'[\![\Gamma']\!]$ is used to look up reduction functions, but the reduction function signatures are restricted to have no regions named in their arguments and so

$$M[\![\Gamma']\!](id) = M'[\![\Gamma']\!](id)$$

Finally, from

$$valid\_interleave(S, C, E'', E', E_{n+1})$$

and Lemma 3 we conclude $E'' \sim H$.
For the last consistency property of the function call, we first observe that $E_{n+1} :_{M'} \Phi'$ is equivalent to

$$E_{n+1} :_M \Phi[r_1/r'_1, \ldots, r_k/r'_k]$$

because, again, E_{n+1} does not mention logical region names and $M'[\![r']\!] = M[\![r]\!]$. From [T-Call] we have

$$\Phi[r_1/r'_1, \ldots, r_k/r'_k] \subseteq \Phi^*$$

and so $E_{n+1} :_M \Phi^*$. Applying Lemma 7 we conclude that $E_{n+1} :_M \Phi$. Finally, from Lemma 5 we conclude $E'' :_M \Phi$.

□

B.2 Proof of Lemma 9

Proof. The proof is by induction on the structure of the derivation of e. The clobber set is only manipulated in one case and only examined in one other case. In all other cases, the clobber set is unchanged and used only in the derivation of subexpressions. The case where the clobber set is manipulated is [E-Call], but the use of a union operation still allows the inductive hypothesis to be applied to the subexpression e_{n+1}. The examination of the clobber set occurs in [E-Read] where it is used to choose the result v of the read from the heap. For each use of the [E-Read] rule, there are three possible cases:

- $l \in C$, for which $v = S'(l)$ in the derivation using C. Since $C \subseteq C'$, we have $l \in C'$, and $S'(l)$ is the correct result for the C' derivation as well.
- $l \notin C'$, and therefore $l \notin C$. Both derivations may use any value $v' : H(l)$.
- $l \in C, l \notin C'$, for which $v = S'(l)$ must be chosen for the derivation using C. From Theorem 1 and Lemma 1, we have $E \sim H$ and $S' = apply(S, E) \sim H$, so $S'(l) \sim_H H(l)$, and v is a legal choice for v' in the derivation using C'.

□

B.3 Proof of Theorem 2

We need one additional lemma, which follows directly from the definition of *apply*:

Lemma 13. Let S be an initial store, and E_1 and E_2 be memory traces. Then:

$$apply(S, E_1 +\!\!+ E_2) = apply(apply(S, E_1), E_2)$$

The proof of the theorem:

Proof. First, we must show that $any\_interleave(E', E_1, \ldots, E_n)$. This is done by induction on the length of E_1, followed by E_2, and so on. Next, we must show that E' is coherent for both L_{excl} and L_{atomic}. This is done by matching the structure of E' with $E_1, \ldots, E_n$ and the use of Lemma 13. With the question of coherence addressed, the sequential equivalence of E' is trivial for both the exclusive and atomic cases. □

B.4 Proof of Lemma 10

Proof. We consider the three conditions needed to be a valid interleaving.

1. If $E'_a +\!\!+ [\epsilon_1] +\!\!+ [\epsilon_2] +\!\!+ E'_b$ is an interleaving of $E_1, \ldots, E_n$, the proof must first "pop off" all of E'_a, leaving $[\epsilon_1] +\!\!+ [\epsilon_2] +\!\!+ E'_b$ for the interleaved subtrace. The next two steps must pop ϵ_1 and ϵ_2, leaving E'_b. They cannot be in the same subtrace because $\epsilon_1 \# \epsilon_2$, so we could also pop ϵ_2 and then ϵ_1, resulting in a proof that $E'_a +\!\!+ [\epsilon_2] +\!\!+ [\epsilon_1] +\!\!+ E'_b$ is also an interleaving of the constituent traces.
2. For coherence, we must show the value read (if any) by ϵ_1 does not change, the value read (if any) by ϵ_2 does not change, and that any read in E'_b sees no change. With the help of Lemma 13, we can eliminate the common E'_a from all sequences by using $S' = apply(S, E'_a)$. Similarly, to show that

   $$apply(S', [\epsilon_1] +\!\!+ [\epsilon_2] +\!\!+ E) = apply(S', [\epsilon_2] +\!\!+ [\epsilon_1] +\!\!+ E)$$

   for any E, we need only to show that it holds for $E = []$. The proof is now reduced to showing the following:
   (a) $\epsilon_1 = read(l, c, v, t) \Rightarrow S'(l) = apply(S', \epsilon_2)(l)$ - since ϵ_1 is a read, either ϵ_2 must be a read (in which case $apply(S, \epsilon_2) = S'$) or must be to a different location, so the result of applying just it to S' cannot change the value in location l.
   (b) $\epsilon_2 = read(l, c, v, t) \Rightarrow apply(S', \epsilon_1)(l) = S'(l)$ - this is the same as above, with ϵ_1 and ϵ_2 switched.
   (c) $apply(S', \epsilon_1 +\!\!+ \epsilon_2) = apply(S', \epsilon_2 +\!\!+ \epsilon_1)$ - we must show that the two store operations $apply(\bullet, \epsilon_1)$ and $apply(\bullet, \epsilon_2)$ commute. Modifications to two different locations commute, as do two reads (which make no change to the store). Finally, two reductions using the same function to the same location also commute.
3. The proof that sequential equivalence is maintained comes down to showing the final store is identical after the swap of the two operations, and this is merely a special case of the third piece of the coherence proof.

□

B.5 Proof of Lemma 11

Proof. We will use Theorem 2 and Lemma 10 to make repeated swaps of adjacent operations to transform E' into $E_1 +\!\!+ \ldots +\!\!+ E_n$. The symmetry of Lemma 10 is then used to show that the validity of $E_1 +\!\!+ \ldots +\!\!+ E_n$ under Theorem 2 implies the validity of E'.

The swapping algorithm as follows: Associate with each operation in E' the index of that operation in $E_1 +\!\!+ \ldots +\!\!+ E_n$. Any consecutive pair of operations ϵ_1 and ϵ_2 for which $index(\epsilon_1) > index(\epsilon_2)$ is a *misordered pair*. For any such pair, ϵ_1 and ϵ_2 will be from different constituent traces (if they were from the same trace, their misordering would mean that E' was not an interleaving of $E_1, \ldots, E_n$). Every pair of traces is non-interfering, so $\epsilon_1 \# \epsilon_2$ and they can be swapped while preserving validity of the trace. At most $n^2/2$ swaps are needed to eliminate all misordered pairs. □

B.6 Proof of Lemma 12

Proof. The proof is by contradiction. Assume $E_1 \not\# E_2$. Then there must be some ϵ_1 in E_1 and some ϵ_2 in E_2 such that $\epsilon_1 \not\# \epsilon_2$. This requires that ϵ_1 and ϵ_2 be operations to the same location l and not both be reads or both reductions using the same function name. Assume $\epsilon_1 = read(l, c_1, v_1, t_1)$ and $\epsilon_2 = write(l, c_2, v_2, t_2)$. The definition of $E_1 :_M \Phi_1$ tells us that there must be some r_1 satisfying $l \in M(r_1) \wedge reads(r_1) \in \Phi_1$. Similarly, there must be some r_2 satisfying $l \in M(r_2) \wedge writes(r_2) \in \Phi_2$. Letting $\phi_1 = reads(r_1)$ and $\phi_2 = writes(r_2)$, we have $M(r_1) \cap M(r_2) \neq \emptyset$ (the intersection contains at least l), so $\phi_1 \not\#_M \phi_2$ and $\Phi_1 \not\#_M \Phi_2$. There are seven other cases to consider (using $id_1 \neq id_2$ for the reduce-reduce case), but all yield the same result. □

B.7 Proofs of Theorems 3 and 4

As discussed in Section 7, Theorem 3 follows directly Lemmas 11 and 12. The proof of Theorem 4 is parallel, using the $\#^A$ operator instead of $\#$.

B.8 Proof of Theorem 5

Proof. Let E_1 be the result of an evaluation of e_1, with E'_1 being the memory trace from the evaluation of e'_1 within the expression tree. E_1 will be formed from a tree of valid interleavings, each of which must include all the memory operations of each constituent trace. By induction over the number of intermediate subexpressions between e_1 and e'_1, we can show that any memory operation ϵ'_1 that is in E'_1 must also be in E_1. Similarly, any memory operation ϵ'_2 that is in E'_2 must also be in E_2. We have $E_1 \# E_2$ from Lemma 12, and therefore $\epsilon_1 \# \epsilon_2$ for all $\epsilon_1 \in E_1$ and $\epsilon_2 \in E_2$. Since all ϵ'_1 are in E_1 and all ϵ'_2 are in E_2, we have $\epsilon'_1 \# \epsilon'_2$ and therefore $E'_1 \# E'_2$. □