Deterministic Parallelism via Liquid Effects [*]

Ming Kawaguchi Patrick Rondon Alexander Bakst Ranjit Jhala
University of California, San Diego
{mwookawa,prondon,abakst,jhala}@cs.ucsd.edu

Abstract

Shared memory multithreading is a popular approach to parallel programming, but also fiendishly hard to get right. We present *Liquid Effects*, a type-and-effect system based on refinement types which allows for fine-grained, low-level, shared memory multithreading while statically guaranteeing that a program is deterministic. Liquid Effects records the effect of an expression as a formula in first-order logic, making our type-and-effect system highly expressive. Further, effects like Read and Write are recorded in Liquid Effects as ordinary uninterpreted predicates, leaving the effect system open to extension by the user. By building our system as an extension to an existing dependent refinement type system, our system gains precise value- and branch-sensitive reasoning about effects. Finally, our system exploits the Liquid Types refinement type inference technique to automatically infer refinement types and effects. We have implemented our type-and-effect checking techniques in CSOLVE, a refinement type inference system for C programs. We demonstrate how CSOLVE uses Liquid Effects to prove the determinism of a variety of benchmarks.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification – Validation; D.1.3 [*Programming Techniques*]: Concurrent Programming – Parallel Programming

General Terms Languages, Reliability, Verification

Keywords Liquid Types, Type Inference, Dependent Types, C, Safe Parallel Programming, Determinism

1. Introduction

How do we program multi-core hardware? While many models have been proposed, the model of multiple sequential threads concurrently executing over a single shared memory remains popular due to its efficiency, its universal support in mainstream programming languages and its conceptual simplicity. Unfortunately, shared memory multithreading is fiendishly hard to get right, due to the inherent non-determinism of thread scheduling. Unless the programmer is exceptionally vigilant, concurrent accesses to shared data can result in non-deterministic behaviors in the program, potentially yielding difficult-to-reproduce "heisenbugs" whose appearance depends on obscurities in the bowels of the operation system's scheduler and are hence notoriously hard to isolate and fix.

Determinism By Default. One way forward is to make parallel programs "deterministic by default" [2] such that, no matter how threads are scheduled, program behavior remains the same, eliminating unruly heisenbugs and allowing the programmer to reason about their parallel programs as if they were sequential. In recent years, many static and dynamic approaches have been proposed for ensuring determinism in parallel programs. While dynamic approaches allow arbitrary data sharing patterns, they also impose non-trivial run-time overheads and hence are best in situations where there is relatively little sharing [5]. In contrast, while static approaches have nearly no run-time overhead, they have been limited to high-level languages like Haskell [7] and Java [2] and require that shared structures be refactored into specific types or classes for which deterministic compilation strategies exist, thereby restricting the scope of sharing patterns.

Liquid Effects. In this paper, we present *Liquid Effects*, a type-and-effect system based on refinement type inference which uses recent advances in SMT solvers to provide the best of both worlds: it allows for fine-grained shared memory multithreading in low-level languages with program-specific data access patterns, and yet statically guarantees that programs are deterministic.

Any system that meets these goals must satisfy several criteria. First, the system must support *precise and expressive effect specifications*. To precisely characterize the effect of program statements, it must precisely reason about complex heap access patterns, including patterns not foreseen by the system's designers, and it must be able to reason about relations between program values like loop bounds and array segment sizes. Second, the system must be *extensible* with user-defined effects to track effects which are domain-specific and thus could not be foreseen by the system's designers. Finally, the system must support *effect inference*, so that the programmer is not overwhelmed by the burden of providing effect annotations.

1. Precise and Expressive Effect Specifications. Our Liquid Effects type-and-effect system expresses the effect of a statement on the heap as a formula in first-order logic that classifies which addresses in the heap are accessed and with what effect they were accessed — for example, whether the data was read or written. Expressing effects as first-order formulas makes our type-and-effect system highly expressive: for example, using the decidable theory of linear arithmetic, it is simple to express heap access patterns like processing chunks of an array in parallel or performing strided accesses in a number of separate threads. Using first-order formulas for effects ensures that our system can express complex access patterns not foreseen by the system's designers, and allows us to incorporate powerful off-the-shelf SMT solvers in our system for reasoning about effects. Further, by building our type-and-effect system as

[*] This work was supported by NSF grants CCF-0644361, CNS-0720802, CCF-0702603, and a gift from Microsoft Research.

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06. . . $5.00

```
void sumBlock (char *a, int i, int len) {
  if (len <= 1) return;

  int hl = len / 2;
  cobegin {
    sumBlock (a, i, hl);
    sumBlock (a, i + hl, len - hl);
  }

  a[i] += a[i + hl];
}

int sum1 (char *a, int len) {
  sumBlock (a, 0, len);

  return a[0];
}
```

Figure 1. Parallel Array Summation With Contiguous Partitions

an extension to an existing dependent refinement type system and allowing effect formulas to reference program variables, our system gains precise value and branch-sensitive reasoning about effects.

***2. Extensibility*.** Our effect formulas specify how locations on the heap are affected using *effect labeling predicates* like Read and Write. These effect labeling predicates are ordinary uninterpreted predicates in first-order logic; our system does not treat effect labeling predicates specially. Thus, we are able to provide extensibility by parameterizing our system over a set of user-defined effect labeling predicates. Further, we allow the user to give *commutativity declarations* specifying which pairs of effects have benign interactions when they simultaneously occur in separate threads of execution. This enables users to track domain-specific effects not envisioned by the designers of the type-and-effect system.

***3. Effect Inference*.** Our type rules can be easily recast as an algorithm for generating constraints over unknown refinement types and effect formulas; these constraints can then be solved using the Liquid Types technique for invariant inference, thus yielding a highly-automatic type-based method for proving determinism.

To illustrate the utility of Liquid Effects, we have implemented our type-and-effect checking techniques in CSOLVE, a refinement type inference system for C programs based on Liquid Types. We demonstrate how CSOLVE uses Liquid Effects to prove the determinism of a variety of benchmarks from the literature requiring precise, value-aware tracking of both built-in and user-defined heap effects while imposing a low annotation burden on the user. As a result, CSOLVE opens the door to efficient, deterministic programming in mainstream languages like C.

2. Overview

We start with a high-level overview of our approach. Figures 1 and 2 show two functions, `sum1` and `sum2` respectively, which add up the elements of an array of size 2^k *in-place* by dividing the work up into independent sub-computations that can be executed in parallel. At the end of both procedures, the first element of the array contains the final sum. In this section, we demonstrate how our system seamlessly combines path-sensitive reasoning using refinement types, heap effect tracking, and SMT-based reasoning to prove that `sum1` and `sum2` are deterministic.

2.1 Contiguous Partitions

The function `sum1` sums array `a` of length `len` using the auxiliary function `sumBlock`, which sums the elements of the contiguous segment of array `a` consisting of `len`-many elements and beginning at index `i`. Given an array segment of length `len`, `sumBlock` computes the sum of the segment's elements by dividing it into two contiguous subsegments which are recursively summed using `sumBlock`. The recursive calls place the sum of each subsegment in the first element of the segment; `sumBlock` computes its final result by summing the first element of each subsegment and stores it in the first element of the entire array segment.

***Cobegin Blocks*.** To improve performance, we can perform the two recursive calls inside `sumBlock` in parallel. This is accomplished using the `cobegin` construct, which evaluates each statement in a block in parallel. Note that the recursive calls both read and write to the array passed to `sumBlock`; thus, to prove that the program is deterministic, we have to show that the part of the array that is written by each call does not overlap with the portion of the array that is read by the other. In the following, we demonstrate how our system is able to show that the recursive calls access disjoint parts of the array and thus that the function `sumBlock` is deterministic.

***Effect Formulas*.** We compute the region of the array accessed by `sumBlock` as an *effect formula* that describes which pointers into the array are accessed by a call to `sumBlock` with parameters `i` and `len`. We note that the final line of `sumBlock` writes element `i` of `a`. We state the effect of this write as the formula [1]

$$\Sigma_{\texttt{a[i]}} \doteq \nu = \texttt{a} + \texttt{i}.$$

We interpret the above formula, $\Sigma_{\texttt{a[i]}}$, as "the pointer ν accessed by the expression $\texttt{a}[\texttt{i}]$ is equal to the pointer obtained by incrementing pointer `a` by `i`." The final line of `sumBlock` also reads element $\texttt{i} + \texttt{hl}$ of `a`. The effect of this read is stated as a similar formula:

$$\Sigma_{\texttt{a[i+hl]}} \doteq \nu = \texttt{a} + \texttt{i} + \texttt{hl}.$$

The total effect of the final line of `sumBlock` stating which portion of the heap it accesses is then given by the disjunction of the two effect formulas:

$$\Sigma_{\texttt{a[i]}} \oplus \Sigma_{\texttt{a[i+hl]}} \doteq \nu = \texttt{a} + \texttt{i} \lor \nu = \texttt{a} + \texttt{i} + \texttt{hl}.$$

The above formula says that the heap locations accessed in the final line of `sumBlock` are exactly the locations corresponding to $\texttt{a}[\texttt{i}]$ and $\texttt{a}[\texttt{i} + \texttt{hl}]$. Having determined the effect of the final line of `sumBlock` as a formula over the local variables `a`, `i`, and `hl`, we make two observations to determine the effect of the entire `sumBlock` function as a formula over its arguments `a`, `i`, and `len`. First, we observe that, in the case where $\texttt{len} = 2$,

$$\texttt{a}[\texttt{i} + \texttt{hl}] = \texttt{a}[\texttt{i} + \texttt{len} - 1].$$

From this, we can see inductively that a call to `sumBlock` with index `i` and length `len` will access elements $\texttt{a}[\texttt{i}]$ to $\texttt{a}[\texttt{i} + \texttt{len} - 1]$. Thus, we can soundly ascribe `sumBlock` the effect

$$\Sigma_{\texttt{sumBlock}} \doteq \texttt{a} + \texttt{i} \leq \nu \land \nu < \texttt{a} + \texttt{i} + \texttt{len}.$$

That is, `sumBlock` will access all elements in the given array segment.

***Determinism via Effect Disjointness*.** We are now ready to prove that the recursive calls within `sumBlock` access disjoint regions of the heap and thus that `sumBlock` behaves deterministically when the calls are executed in parallel. We determine the effect of the first recursive call by *instantiating* `sumBlock`'s effect, *i.e.,* by replacing formals `a`, `i`, and `len` with actuals `a`, `i`, and `hl`, respectively:

$$\Sigma_{\text{Call 1}} \doteq \texttt{a} + \texttt{i} \leq \nu \land \nu < \texttt{a} + \texttt{i} + \texttt{hl}.$$

[1] Here we associate effects with program expressions like $\texttt{a}[\texttt{i}]$. However, in the technical development and our implementation, effects are associated with heap locations in order to soundly account for aliasing (section 4).

```
declare effect Accumulate;
declare Accumulate commutes with Accumulate;

void accumLog (char *l, int j)
  effect
   (&l[j], Accumulate(v) && !Read(v) && !Write(v));

void sumStride (char *a, int stride, char *log) {
  foreach (i, 0, THREADS) {
    for (int j = i; j < stride; j += THREADS) {
        a[j] += a[j + stride];
        accumLog (log, j);
    }
  }
}

int sum2 (char *a, int len) {
  log = (char *) malloc (len);

  for (int stride = len/2; stride > 0; stride /= 2)
    sumStride (a, stride, log);

  return a[0];
}
```

Figure 2. Parallel Array Summation With Strided Partitions

The resulting effect states that the first recursive call only accesses array elements $\mathtt{a}[\mathtt{i}]$ through $\mathtt{a}[\mathtt{i} + \mathtt{hl} - 1]$. We determine the effect of the second recursive call with a similar instantiation of `sumBlock`'s effect:

$$\Sigma_{\text{Call 2}} \doteq \mathtt{a} + \mathtt{i} + \mathtt{hl} \leq \nu \wedge \nu < \mathtt{a} + \mathtt{i} + \mathtt{len} - \mathtt{hl}.$$

All that remains is to show that the effects $\Sigma_{\text{Call 1}}$ and $\Sigma_{\text{Call 2}}$ are disjoint. We do so by asking an SMT solver to prove the unsatisfiability of the conjoined *access intersection* formula

$$\Sigma_{\text{Unsafe}} = \Sigma_{\text{Call 1}} \wedge \Sigma_{\text{Call 2}},$$

whose inconsistency establishes that the intersection of the sets of pointers ν accessed by both recursive calls is empty.

2.2 Complex Partitions

In the previous example, function `sum1` computed the sum of an array's elements by recursively subdividing the array into halves and operating on each half separately before combining the results. We were able to demonstrate that `sum1` is deterministic by showing that the concurrent recursive calls to `sumBlock` operate on disjoint contiguous segments of the array. We now show that our system is capable of proving determinism even when the array access patterns are significantly more complex.

Function `sum2` uses the auxiliary function `sumStride`, which sums an array's elements using the following strategy: First, the array is divided into two contiguous segments. Each element in the first half is added to the corresponding element in the second half, and the element in the first half is replaced with the result. The problem is now reduced to summing only the first half of the array, which proceeds in the same fashion, until we can reduce the problem no further, at which point the sum of all the array elements is contained in the first element of the array.

Foreach Blocks. To increase performance, the pairwise additions between corresponding elements in the first and second halves of the array are performed in parallel, with the level of parallelism determined by a compile-time constant `THREADS`. The parallel threads are spawned using the `foreach` construct. The statement

```
foreach (i, l, u) s;
```

executes statement `s` once with each possible binding of `i` in the range $[\mathtt{l}, \mathtt{u})$. Further, all executions of statement `s` are performed in parallel. The `foreach` loop within `sumStride` spawns `THREADS` many threads. Thread `i` processes elements $\mathtt{i}$, $\mathtt{i} + \mathtt{THREADS}$, $\mathtt{i} + 2 * \mathtt{THREADS}$, *etc.*, of array `a`. While this strided decomposition may appear contrived, it is commonly used in GPU programming to maximize memory bandwidth by enabling "adjacent" threads to access adjacent array cells via *memory coalescing* [1].

To prove that the `foreach` loop within `sumStride` is deterministic, we must show that no location that is written in one iteration is either read from or written to in another iteration. (We ignore the effect of the call to the function `accumLog` for now and return to it later.) Note that this differs from the situation with `sum1`: in order to demonstrate that `sum1` is deterministic, it was not necessary to consider read and write effects separately; it was enough to know that recursive calls to `sumBlock` operated on disjoint portions of the heap. However, the access pattern of a single iteration of the `foreach` loop in `sumStride` is considerably more complicated, and reasoning about noninterference between loop iterations will require tracking not only which locations are affected but precisely how they are affected — that is, whether they are read from or written to.

Labeled Effect Formulas. We begin by computing the effect of each iteration of the loop on the heap, tracking which locations are accessed as well as which effects are performed at each location. We first compute the effect of each iteration of the inner `for` loop, which adds the value of $\mathtt{a}[\mathtt{j} + \mathtt{stride}]$ to $\mathtt{a}[\mathtt{j}]$. We will use the unary *effect labeling predicates* $\mathrm{Read}(\nu)$ and $\mathrm{Write}(\nu)$ to record the fact that a location in the heap has or has not been read from or written to, respectively. We capture the effect of the read of $\mathtt{a}[\mathtt{j} + \mathtt{stride}]$ with the effect formula

$$\Sigma_1 \doteq \mathrm{Read}(\nu) \wedge \neg\mathrm{Write}(\nu) \wedge \nu = \mathtt{a} + \mathtt{j} + \mathtt{stride}.$$

Note that we have used the effect label predicates Read and Write to record not only that we have accessed $\mathtt{a}[\mathtt{j} + \mathtt{stride}]$ but also that this location was only read from, not written to. We record the effect of the write to $\mathtt{a}[\mathtt{j}]$ with a similar formula:

$$\Sigma_2 \doteq \mathrm{Write}(\nu) \wedge \neg\mathrm{Read}(\nu) \wedge \nu = \mathtt{a} + \mathtt{j}$$

As before, the overall effect of these two operations performed sequentially, and thus the effect of a single iteration of the inner `for` loop is the disjunction of the two effect formulas above:

$$\Sigma_{\mathtt{j}} \doteq \Sigma_1 \vee \Sigma_2$$

Iteration Effect Formulas. Having computed the effect of a single iteration of `sumStride`'s inner `for` loop, the next step in proving that `sumStride` is deterministic is to compute the effect of each iteration of the outer `foreach` loop, with our goal being to show that the effects of any two distinct iterations must be disjoint. We begin by generalizing the effect formula we computed to describe the effect of a single iteration of the inner `for` loop to an effect formula describing the effect of the entire `for` loop. The effect formula describing the effect of the entire `for` is an effect formula that does not reference the loop induction variable `j`, but is implied by the conjunction of the loop body's single-iteration effect and any loop invariants that apply to the loop induction variable `j`.

Note that the induction variable `j` enjoys the loop invariant

$$\mathtt{j} < \mathtt{stride} \wedge \mathtt{j} \equiv \mathtt{i} \bmod \mathtt{THREADS}.$$

Given this invariant for `j`, we can now summarize the effect of the `for` loop as an effect formula which does not reference the loop induction variable `j` but is implied by the conjunction of the above invariant on $\texttt{j}$ and the single-iteration effect formula $\Sigma_{\texttt{j}}$:

$$\begin{aligned}\Sigma_{\texttt{for}} \doteq\ & (\mathrm{Write}(\nu) \Rightarrow (\nu - \texttt{a}) \equiv \texttt{i} \bmod \texttt{THREADS}) \\ & \wedge (\mathrm{Write}(\nu) \Rightarrow \nu < \texttt{a} + \texttt{stride}) \\ & \wedge (\mathrm{Read}(\nu) \Rightarrow \nu \geq \texttt{a} + \texttt{stride})\end{aligned} \tag{1}$$

We have now computed the effect of `sumStride`'s inner `for` loop, and thus the effect of a single iteration of `sumStride`'s outer `foreach` loop.

Effect Projection. We define an effect projection operator $\pi(\Sigma, E)$ which returns the restriction of the effect formula Σ to the effect label E. In essence, the effect projection is the formula implied by the conjunction of Σ and $E(\nu)$. For example, the projection of the Write effect of iteration $\texttt{i}$ of the `foreach` loop represents the addresses written by thread $\texttt{i}$.

Determinism via Iteration Effect Disjointness. To prove that the `foreach` loop is deterministic, we must show that values that are written in one iteration are not read or overwritten in another. First, we show that no two `foreach` iterations (*i.e.,* threads) write to the same location, or, in other words, that distinct iterations write through disjoint sets of pointers. We establish this fact by asking the SMT solver to prove the unsatisfiability of the *write-write intersection* formula:

$$\begin{aligned}\Sigma_{\mathsf{Unsafe}} = \pi(\Sigma_{\texttt{for}}, \mathrm{Write}) \wedge \pi(\Sigma_{\texttt{for}}, \mathrm{Write})[\texttt{i} \mapsto \texttt{i}'] \\ \wedge\, \texttt{i} \neq \texttt{i}' \wedge 0 \leq \texttt{i}, \texttt{i}' < \texttt{THREADS}.\end{aligned}$$

The formula is indeed unsatisfiable as, from Equation 1,

$$\begin{aligned}\pi(\Sigma_{\texttt{for}}, \mathrm{Write}) \doteq\ & (\nu - \texttt{a}) \equiv \texttt{i} \bmod \texttt{THREADS} \\ & \wedge\, \nu < \texttt{a} + \texttt{stride}\end{aligned} \tag{2}$$

and so, after substituting $\texttt{i}'$ the query formula is

$$(\nu - \texttt{a}) \equiv \texttt{i} \bmod \texttt{THREADS} \wedge (\nu - \texttt{a}) \equiv \texttt{i}' \bmod \texttt{THREADS}$$

which is unsatisfiable when $\texttt{i} \neq \texttt{i}'$.

Next, we prove that no heap location is read in one iteration (*i.e.,* thread) and written in another. As before, we verify that the intersection of the pointers read in one iteration with those written in another iteration is empty. Concretely, we ask the SMT solver to prove the unsatisfiability of the *write-read intersection* formula

$$\Sigma_{\mathsf{Unsafe}} = \pi(\Sigma_{\texttt{for}}, \mathrm{Write}) \wedge \pi(\Sigma_{\texttt{for}}, \mathrm{Read})[\texttt{i} \mapsto \texttt{i}'] \wedge \texttt{i} \neq \texttt{i}'$$

where, from Equation 1, the locations *read* in a single iteration of the `foreach` loop are described by the Read projection

$$\pi(\Sigma_{\texttt{for}}, \mathrm{Read}) \doteq \nu \geq \texttt{a} + \texttt{stride}.$$

After substituting the the write effect from Equation 2, we can see that the write-read-intersection formula is unsatisfiable as

$$\nu < \texttt{a} + \texttt{stride} \wedge \nu \geq \texttt{a} + \texttt{stride}$$

is inconsistent. Thus, by proving the disjointness of the write-write and write-read effects, we have proved that the `foreach` loop inside `sumStride` is deterministic.

2.3 User-Defined Effects

By expressing our effect labeling predicates as ordinary uninterpreted predicates in first-order logic, we open the door to allowing the user to define their own effects. Such user-defined effects are first-class: we reason about them using the same mechanisms as the built-in predicates Read and Write, and effect formulas over user-defined predicates are equally as expressive as those over the built-in predicates.

We return to the `sumStride` function to demonstrate the use of user-defined effects. The `for` loop in `sumStride` tracks how many times the entry $\texttt{a}[\texttt{j}]$ is written by incrementing the `j`-th value in the counter array `log` using the externally-defined function `accumLog`. We assume that the `accumLog` function is implemented so that its operation is atomic. Thus, using `accumLog` to increment the count of the same element in two distinct iterations of the `foreach` loop does not cause non-determinism.

Specifying Effects. We create a new user-defined effect predicate to track the effect of the `accumLog` function using the statement

```
declare effect Accumulate;
```

This extends the set of effect predicates that our system tracks to

$$\mathbb{E} \doteq \{\mathrm{Read}, \mathrm{Write}, \mathrm{Accumulate}\}.$$

We then annotate the prototype of the `accumLog` function to specify `accumLog` causes the Accumulate effect to occur on the `j`-th entry of its parameter array `l`:

```
void accumLog (char *l, int j)
  effect
   (&l[j], Accumulate(v) && !Read(v) && !Write(v));
```

Specifying Commutativity. We specify that our user effect Accumulate does not cause nondeterminism even when it occurs on the same location in two separate threads using the *commutativity annotation*

```
declare Accumulate commutes with Accumulate;
```

This annotation extends the set of pairs of effects that commute with each other — that is, which can occur in either order without affecting the result of the computation. In particular, the commutativity annotation above extends the set of commutable pairs to

$$\mathbb{C} \doteq \{(\mathrm{Read}, \mathrm{Read}), (\mathrm{Accumulate}, \mathrm{Accumulate})\}.$$

The extended set formally specifies that in addition to pairs of Read effects (included by default), pairs of of Accumulate effects also commute, and hence may be allowed to occur simultaneously.

Generalized Effect Disjointness. Finally, we generalize our method for ensuring determinism. We check the disjointness of two effect formulas Σ_1 and Σ_2 by asking the SMT solver to prove the unsatisfiability of the *effect intersection* formula:

$$\Sigma_{\mathsf{Unsafe}} = \exists (E_1, E_2) \in (\mathbb{E} \times \mathbb{E} \setminus \mathbb{C}). \pi(\Sigma_1, E_1) \wedge \pi(\Sigma_2, E_2).$$

That is, for each possible combination of effect predicates that have not been explicitly declared to commute, we check that the two effect sets are disjoint when projected on those effects.

Thus, by declaring an Accumulate effect which is commutative, we are able to verify the determinism of `sum2`.

Effect Inference. In the preceding, we gave explicit loop invariants and heap effects where necessary. Later in this paper, we explain how we use Liquid Type inference [28] to reduce the annotation burden on the programmer by automatically inferring the loop invariants and heap effects given above.

Outline. The remainder of this paper is organized as follows: In section 3, we give the syntax of LC_E programs, informally explain their semantics, and give the syntax of LC_E's types. We define the type system of LC_E in section 4. In section 5, we give an overview of the results of applying an implementation of a typechecker for LC_E to a series of examples from the literature. We review related work in section 6.

3. Syntax and Semantics

In this section, we present the syntax of LC_E programs, informally discuss their semantics, and give the syntax of LC_E types.

v	$::=$		**Values**
	$\mid$	x	variable
	$\mid$	n	integer
	$\mid$	$\&n$	pointer
a	$::=$		**Pure Expressions**
	$\mid$	v	value
	$\mid$	$a_1 \circ a_2$	arithmetic operation
	$\mid$	$a_1 +_p a_2$	pointer arithmetic
	$\mid$	$a_1 \sim a_2$	comparison
e	$::=$		**Expressions**
	$\mid$	a	pure expression
	$\mid$	$*v$	heap read
	$\mid$	$*v_1 := v_2$	heap write
	$\mid$	**if** v **then** e_1 **else** e_2	if-then-else
	$\mid$	$f(\overline{v})$	function call
	$\mid$	**malloc**(v)	memory allocation
	$\mid$	**let** $x = e_1$ **in** e_2	let binding
	$\mid$	**letu** $x =$ **unfold** v **in** e	location unfold
	$\mid$	**fold** l	location fold
	$\mid$	$e_1 \parallel e_2$	parallel composition
	$\mid$	**for each** x **in** v_1 **to** v_2 $\{e\}$	parallel iteration
F	$::=$	$f\ (\overline{x_i})\ \{\ e\ \}$	**Function Declarations**
P	$::=$	$\overline{F}\ e$	**Programs**

Figure 3. Syntax of LC_E programs

b	$::=$		**Base Types**
	$\mid$	$\texttt{int}(i)$	integer
	$\mid$	$\texttt{ref}(l, i)$	pointer
p	$::=$	$\phi(\overline{v})$	**Refinement Formulas**
τ	$::=$	$\{\nu : b \mid p\}$	**Refinement Types**
i	$::=$		**Indices**
	$\mid$	n	constant
	$\mid$	n^{+m}	sequence
c	$::=$	$\overline{i : \tau}$	**Blocks**
l	$::=$		**Heap Locations**
	$\mid$	$\tilde{l}$	abstract location
	$\mid$	l_j	concrete location
h	$::=$		**Heap Types**
	$\mid$	ε	empty heap
	$\mid$	$h * l \mapsto c$	extended heap
Σ	$::=$		**Heap Effects**
	$\mid$	ε	empty effect
	$\mid$	$\Sigma * \tilde{l} \mapsto p$	extended effect
σ	$::=$	$(\overline{x_i : \tau_i})/h_1 \rightarrow \tau/h_2/\Sigma$	**Function Schemes**

Figure 4. Syntax of LC_E types

3.1 Programs

The syntax of LC_E programs is shown in Figure 3. Most of the expression forms for expressing sequential computation are standard or covered extensively in previous work [28]; the expression forms for parallel computation are new to this work.

Values. The set of LC_E values v includes program variables, integer constants n, and constant pointer values $\&n$ representing addresses in the heap. With the exception of the null pointer value $\&0$, constant pointer values do not appear in source programs.

Expressions. The set of pure LC_E expressions a includes values v, integer arithmetic expressions $a_1 \circ a_2$, where $\circ$ is one of the standard arithmetic operators $+$, $-$, $*$, *etc.*, pointer arithmetic expressions $a_1 +_p a_2$, and comparisons $a_1 \sim a_2$ where $\sim$ is one of the comparison operators $=$, $<$, $>$, *etc.* Following standard practice, zero and non-zero values represent falsity and truth, respectively.

The set of LC_E expressions e includes the pure expressions a as well as all side-effecting operations. The expression forms for pointer read and write, if-then-else, function call, memory allocation, and let binding are all standard. The location unfold and fold expressions are used to support type-based reasoning about the heap using strong updates As they have no effect on the dynamic semantics of LC_E programs, we defer discussion of location unfold and fold expressions to section 4.

Parallel Expressions. The set of LC_E expressions includes two forms for expressing parallel computations. The first, parallel composition $e_1 \parallel e_2$, evaluates expressions e_1 and e_2 in parallel and returns when both subexpressions have evaluated to values.

The second parallel expression form is the parallel iteration expression form **for each** x **in** v_1 **to** v_2 $\{e\}$, where v_1 and v_2 are integer values. A parallel iteration expression is evaluated by evaluating the body expression e once for each possible binding of variable x to an integer in the range $[v_1, v_2)$. All iterations are evaluated in parallel, and the parallel iteration expression returns when all iterations have finished evaluating. Both forms of parallel expressions are evaluated solely for their side effects.

Functions and Programs. A function declaration $f\ (\overline{x_i})\ \{\ e\ \}$ declares a function f with arguments x_i whose body is the expression e. The return value of the function is the value of the expression. An LC_E program consists of a sequence of function declarations F followed by an expression e which is evaluated in the environment containing the previously-declared functions.

3.2 Types

The syntax of LC_E types is shown in Figure 4.

Base Types. The base types b of LC_E include integer types $\texttt{int}(i)$ and pointer types $\texttt{ref}(l, i)$. The integer type $\texttt{int}(i)$ describes an integer whose value is in the set described by the index i; the set of indices is described below. The pointer type $\texttt{ref}(l, i)$ describes a pointer value to the heap location l whose offset from the beginning of location l is an integer belonging to the set described by index i.

Indices. The language of LC_E base types uses *indices* i to approximate the values of integers and the offsets of pointers from the starts of the heap locations where they point. There are two forms of indices. The first, the singleton index n, describes the set of integers $\{n\}$. The second, the sequence index n^{+m}, represents the sequence of offsets $\{n + lm\}_{l=0}^{\infty}$.

Refinement Types. The LC_E refinement types τ are formed by joining a base type b with a refinement formula p to form the refinement type $\{\nu : b \mid p\}$. The distinguished *value variable* ν refers to the value described by this type. The *refinement formula* p is a logical formula $\phi(\nu, \overline{v})$ over LC_E values $\overline{v}$ and the value variable ν. The type $\{\nu : b \mid p\}$ describes all values v of base type b that satisfy the formula $p[\nu \mapsto v]$. Our refinements are first-order formulas with equality, uninterpreted functions, and linear arithmetic.

When it is unambiguous from the context, we use b to abbreviate $\{\nu : b \mid \text{true}\}$.

Blocks. A block c describes the contents of a heap location as a sequence of bindings from indices i to refinement types τ. Each binding $i : \tau$ states that a value of type τ is contained in the block at each offset n from the start of the block which is described by the index i. Bindings of the form $m : \tau$, where types are bound to singleton indices, refer to exactly one element within the block; as such, we allow the refinement formulas within the same block to refer to the value at offset m using the syntax $@m$. We require all indices in a block to be disjoint.

Heap Types. Heap types h statically describe the contents of run-time heaps as sets of bindings from heap locations l to blocks c. A heap type is either the empty heap ε or a heap h extended with a binding from location l to block c, written $h * l \mapsto c$. The location l is either an *abstract location* $\tilde{l}$, which corresponds to arbitrarily many run-time heap locations, or a *concrete location* l_j, which corresponds to exactly one run-time heap location. The distinction between abstract and concrete locations is used to implement a form of sound strong updates on heap types in the presence of aliasing, unbounded collections, and concurrency; we defer detailed discussion to section 4. Locations may be bound at most once in a heap.

Heap Effects. A heap effect Σ describes the effect of an expression on a set of heap locations as a set of bindings from heap locations l to effect formulas p. A heap effect is either the empty effect ε or an effect Σ extended with a binding from location $\tilde{l}$ to an *effect formula* p over the in-scope program variables and the value variable ν, written $\Sigma * \tilde{l} \mapsto p$. Intuitively, an effect binding $\tilde{l} \mapsto p$ describes the effect of an expression on location $\tilde{l}$ as the set of pointers v such that the formula $p[\nu \mapsto v]$ is valid. Note that we only bind abstract locations in heap effects.

Effect Formulas. The formula portion of an effect binding can describe the effect of an expression with varying degrees of precision, from simply describing whether the expression accesses the location at all, to describing which offsets into a location are accessed, to describing which offsets into a location are accessed and how they are accessed (*e.g.* whether they are read or written).

For example, if we use the function $\mathrm{BB}(\nu)$ to refer to the beginning of the block where ν points, we can record the fact that expression e accesses the first ten items in the block at location $\tilde{l}$ (for either reading or writing) with the effect binding

$$\tilde{l} \mapsto \mathrm{BB}(\nu) \leq \nu \wedge \nu < \mathrm{BB}(\nu) + 10,$$

i.e., by stating that all pointers used by e to accesses location $\tilde{l}$ satisfy the above formula. To record that an expression does not access location $\tilde{l}$ at all, we ascribe it the effect formula false.

We add further expressiveness to our language of effect formulas by enriching it with effect-specific predicates used to describe the particular effects that occurred within a location. We use the unary predicates $\mathrm{Read}(v)$ and $\mathrm{Write}(v)$ to indicate that pointer v was read from or written to, respectively. Using these predicates, we can precisely state how an expression depends upon or affects the contents of a heap location.

For example, we can specify that an expression does not write to a location $\tilde{l}$ using the effect binding

$$\tilde{l} \mapsto \neg\mathrm{Write}(\nu).$$

On the other hand, we may specify precisely which offsets into $\tilde{l}$ are written by using Write to guard a predicate describing a set of pointers into $\tilde{l}$. For example, the effect binding

$$\tilde{l} \mapsto \mathrm{Write}(\nu) \Rightarrow (\mathrm{BB}(\nu) \leq \nu \wedge \nu < \mathrm{BB}(\nu) + 10)$$

describes an expression which writes to the first ten elements of the block at $\tilde{l}$.

Effect predicates like Read and Write may only appear in heap effect bindings; they may not appear in type refinements.

Function Schemes. A *LC_E function scheme*

$$(\overline{x_i : \tau_i})/h_1 \ \rightarrow \ \tau/h_2/\Sigma$$

is the type of functions which take arguments x_i of corresponding types τ_i in a heap of type h_1, return values of type τ in a heap of type h_2, and incur side-effects described by the heap effect Σ. The types of function parameters may depend on the other function parameters, and the type of the input heap may depend on the parameters. The types of the return value, output heap, and heap effect may depend on the types of the parameters. We implicitly quantify over all the location names appearing in a function scheme.

4. Type System

In this section, we present the typing rules of LC_E. We begin with a discussion of LC_E's type environments. We then discuss LC_E's expression typing rules, paying particular attention to how the rules carefully track side effects. Finally, we describe how we use tracked effects to ensure determinism.

Type Environments. Our typing rules make use of three types of environments. A *local environment*, Γ, is a sequence of *type bindings* of the form $x : \tau$ recording the types of in-scope variables and *guard formulas* p recording any branch conditions under which an expression is evaluated. A global environment, Φ, is a sequence of function type bindings of the form $f : S$ which maps functions to their refined type signatures. Finally, an effect environment Π is a pair $(\mathbb{E}, \mathbb{C})$. The first component in the pair, $\mathbb{E}$, is a set of effect label predicates. The second component in the pair, $\mathbb{C}$, is a set of pairs of effect label predicates such that $(E_1, E_2) \in \mathbb{C}$ states that effect E_1 *commutes* with E_2. By default, $\mathbb{E}$ includes the built-in effects, Read and Write, and $\mathbb{C}$ includes the pair $(\mathrm{Read}, \mathrm{Read})$, which states that competing reads to the same address in different threads produce a result that is independent of their ordering. In our typing rules, we implicitly assume a single global effect environment Π.

Local environments are well-formed if each bound type or guard formula is well-formed in the environment that precedes it. An effect environment $(\mathbb{E}, \mathbb{C})$ is well-formed as long as $\mathbb{C}$ is symmetric and only references effects in the set $\mathbb{E}$, *i.e.,* $\mathbb{C} \subseteq \mathbb{E} \times \mathbb{E}$, and $(E_1, E_2) \in \mathbb{C}$ iff $(E_2, E_1) \in \mathbb{C}$.

$$\begin{aligned}
\Gamma &::= \epsilon \mid x : \tau ; \Gamma \mid p ; \Gamma && \text{(Local Environment)} \\
\Phi &::= \epsilon \mid \mathtt{f} : \sigma ; \Phi && \text{(Global Environment)} \\
\mathbb{E} &::= \{\mathrm{Read}, \mathrm{Write}\} \mid E ; \mathbb{E} \\
\mathbb{C} &::= (\mathrm{Read}, \mathrm{Read}) \mid (E, E) ; \mathbb{C} \\
\Pi &::= (\mathbb{E}, \mathbb{C}) && \text{(Effect Environment)}
\end{aligned}$$

4.1 Typing Judgments

We now discuss the rules for ensuring type well-formedness, subtyping, and typing expressions. Due to space constraints, we only present the formal definitions of the most pertinent rules, and defer the remainder to the accompanying technical report [16].

Subtyping Judgments. The rules for subtyping refinement types are straightforward: type τ_1 is a subtype of τ_2 in environment Γ, written $\Gamma \vdash \tau_1 <: \tau_2$, if 1) τ_1's refinement implies τ_2 under the assumptions recorded as refinement types and guard predicates in Γ and 2) τ_1's index is included in τ_2's when both are interpreted as sets.

Heap type h_1 is a subtype of h_2 in environment Γ, written $\Gamma \vdash h_1 <: h_2$, if both heaps share the same domain and, for each location l, the block bound to l in h_1 is a subtype of the block bound to l in h_2. Subtyping between blocks is pairwise subtyping between the types bound to each index in each block; we use substitutions to place bindings to offsets $@i$ in the environment.

Because effect formulas are first-order formulas, subtyping between effects is simply subtyping between refinement types: an effect Σ_1 is a subtype of Σ_2 in environment Γ, written $\Gamma \vdash \Sigma_1 <:$

Σ_2, if each effect formula bound to a location in Σ_1 implies the formula bound to the same location in Σ_2 using the assumptions in Γ.

Type Well-Formedness. The type well-formedness rules of LC_E ensure that all refinement and effect formulas are well-scoped and that heap types and heap effects are well-defined maps over locations. These rules are straightforward; we briefly discuss them below, deferring their formal definitions [16].

A heap effect Σ is well-formed with respect to environment Γ and heap h, written $\Gamma, h \vdash \Sigma$, if it binds only abstract locations which are bound in h, binds a location at most once, and binds locations to effect formulas which are well-scoped in Γ. Further, an effect formula p is well-formed in an effect environment $(\mathbb{E}, \mathbb{C})$ if all effect names in p are present in $\mathbb{E}$.

A refinement type τ is well-formed with respect to Γ, written $\Gamma \vdash \tau$, if its refinement predicate references only variables contained in Γ. A heap type h is well-formed in environment Γ, written $\Gamma \vdash h$, if it binds any location at most once, binds a corresponding abstract location for each concrete location it binds, and contains only well-formed refinement types.

A "world" consisting of a refinement type, heap type, and heap effect is well-formed with respect to environment Γ, written $\Gamma \vdash \tau/h/\Sigma$, if, with respect to the environment Γ, the type and heap type are well-formed and the heap effect Σ is well-formed with respect to the heap h.

The rule for determining the well-formedness of function type schemes is standard.

Pure Expression Typing. The rules for typing a pure expression a in an environment Γ, written $\Gamma \vdash a : \tau$, are straightforward, and are deferred to [16]. These rules assign each pure expression a refinement type that records the exact value of the expression.

Expression Typing and Effect Tracking. Figure 5 shows the rules for typing expressions which explicitly manipulate heap effects to track or compose side-effects. Each expression is typed with a refinement type, a refinement heap assigning types to the heap's contents after evaluating the expression, and an effect which records how the expression accesses each heap location as a mapping from locations to effect formulas. We use the abbreviation `void` to indicate the type $\{\nu : \mathtt{int}(0) \mid \mathrm{true}\}$.

Pointer dereference expressions $*v$ are typed by the rule [T-READ]. The value v is typed to find the location and index into which it points in the heap; the type of the expression is the type at this location and index. The rule records the read's effect in the heap by creating a new binding to v's abstract location, $\tilde{l}$, and uses the auxiliary function *logEffect* to create an effect formula which states that the only location that is accessed by this dereference is exactly that pointed to by v (*i.e.,* $\nu = v$), that the location is read (*i.e.,* $\mathrm{Read}(\nu)$), and that no other effect occurs.

The rules for typing heap-mutating expressions of the form $*v_1 := v_2$, [T-SUPD] and [T-WUPD], are similar to [T-READ]. The two rules for mutation differ only in whether they perform a strong update on the type of the heap, *i.e.,* writing through a pointer with a singleton index strongly updates the type of the heap, while writing through a pointer with a sequence index does not.

Expressions are sequenced using the **let** construct, typed by rule [T-LET]. The majority of the rule is standard; we discuss only the portion concerning effects. The rule types expressions e_1 and e_2 to yield their respective effects Σ_1 and Σ_2. The effect of the entire **let** expression is the composition of the two effects, $\Sigma_1 \oplus \Sigma_2$, defined in Figure 5. We check that Σ_2 is well-formed in the initial environment to ensure that the variable x does not escape its scope.

Rule [T-PAR] types the parallel composition expression $e_1 \parallel e_2$. Expressions e_1 and e_2 are typed to obtain their effects Σ_1 and Σ_2. We then use the OK judgment, defined in Figure 5, to verify that effects Σ_1 and Σ_2 commute, *i.e.,* that the program remains deterministic regardless of the interleaving of the two concurrently-executing expressions. We give $e_1 \parallel e_2$ the effect $\Sigma_1 \oplus \Sigma_2$. We require that the input heap for a parallel composition expression must be *abstract*, that is, contain only bindings for parallel compositions; this forces the expressions which are run in parallel to unfold any locations they access, which in turn enforces

Typing Rules $\boxed{\Phi, \Gamma, h \vdash e : \tau/h_2/\Sigma}$

$$\frac{\Gamma \vdash v : \mathtt{ref}(l_j, i) \qquad h = h_1 * l_j \mapsto \dots, i{:}\tau, \dots}{\Phi, \Gamma, h \vdash *v : \tau/h/\tilde{l} \mapsto \mathit{logEffect}(v, \mathrm{Read})}\ [\text{T-READ}]$$

$$\frac{\begin{array}{c} h = h_1 * l_j \mapsto \dots, n{:}b, \dots \\ \Gamma \vdash v_1 : \mathtt{ref}(l_j, n) \qquad \Gamma \vdash v_2 : b \\ h' = h_1 * l_j \mapsto \dots, n{:}\{\nu : b \mid \nu = v_2\}, \dots \end{array}}{\Phi, \Gamma, h \vdash *v_1 := v_2 : \mathtt{void}/h'/\tilde{l} \mapsto \mathit{logEffect}(v_1, \mathrm{Write})}\ [\text{T-SUPD}]$$

$$\frac{\begin{array}{c} \Gamma \vdash v_1 : \mathtt{ref}(l_j, n^{+m}) \\ \Gamma \vdash v_2 : \hat{\tau} \qquad h = h_1 * l_j \mapsto \dots, n^{+m}{:}\hat{\tau}, \dots \end{array}}{\Phi, \Gamma, h \vdash *v_1 := v_2 : \mathtt{void}/h/\tilde{l} \mapsto \mathit{logEffect}(v_1, \mathrm{Write})}\ [\text{T-WUPD}]$$

$$\frac{\begin{array}{c} \Phi, \Gamma, h \vdash e_1 : \tau_1/h_1/\Sigma_1 \\ \Phi, \Gamma; x{:}\tau_1, h_1 \vdash e_2 : \hat{\tau}_2/\hat{h}_2/\hat{\Sigma}_2 \qquad \Gamma \vdash \hat{\tau}_2/\hat{h}_2/\hat{\Sigma}_2 \end{array}}{\Phi, \Gamma, h \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \hat{\tau}_2/\hat{h}_2/\Sigma_1 \oplus \hat{\Sigma}_2}\ [\text{T-LET}]$$

$$\frac{\begin{array}{c} \Phi, \Gamma, h \vdash e_1 : \tau_1/\hat{h}'/\hat{\Sigma}_1 \qquad \Phi, \Gamma, h \vdash e_2 : \tau_2/\hat{h}'/\hat{\Sigma}_2 \\ \Phi, \Gamma, \hat{h}' \vdash \hat{\Sigma}_{\{1,2\}} \qquad \Phi, \Gamma \vdash OK(\hat{\Sigma}_1, \hat{\Sigma}_2) \qquad h \text{ abstract} \end{array}}{\Phi, \Gamma, h \vdash e_1 \parallel e_2 : \mathtt{void}/\hat{h}'/\hat{\Sigma}_1 \oplus \hat{\Sigma}_2}\ [\text{T-PAR}]$$

$$\frac{\begin{array}{c} \Gamma \vdash v_1 : \mathtt{int}(i) \qquad \Gamma \vdash v_2 : \mathtt{int}(i) \\ \Gamma_1 = \Gamma; x{:}\{\nu : \mathtt{int}(i) \mid v_1 \le \nu < v_2\} \\ \Phi, \Gamma_1, h \vdash e : \tau/h/\Sigma \qquad y \text{ fresh} \\ \Gamma_2 = \Gamma_1; y{:}\{\nu : \mathtt{int}(i) \mid v_1 \le \nu < v_2 \wedge \nu \ne x\} \\ \Phi, \Gamma_2 \vdash OK(\Sigma, \Sigma[x \mapsto y]) \\ \Phi, \Gamma, h \vdash \tau/h/\hat{\Sigma}' \qquad \Gamma_1 \vdash \Sigma <: \hat{\Sigma}' \qquad h \text{ abstract} \end{array}}{\Phi, \Gamma, h \vdash \textbf{for each } x \textbf{ in } v_1 \textbf{ to } v_2\ \{e\} : \mathtt{void}/h/\hat{\Sigma}'}\ [\text{T-FOREACH}]$$

$$\frac{\begin{array}{c} \Gamma \vdash v : \{\nu : \mathtt{ref}(\tilde{l}, i_y) \mid \nu \ne 0\} \\ h = h_0 * \tilde{l} \mapsto \overline{n_k{:}\tau_k}, \overline{i^+{:}\tau^+} \\ \theta = [\overline{@n_k \mapsto x_k}] \qquad \overline{x_k} \text{ fresh} \qquad \Gamma_1 = \Gamma; \overline{x_k{:}\theta\tau_k} \\ l_j \text{ fresh} \qquad h_1 = h * l_j \mapsto \overline{n_k{:}\{\nu = x_k\}}, \overline{i^+{:}\theta\tau^+} \\ \Phi, \Gamma_1; x{:}\{\nu : \mathtt{ref}(l_j, i_y) \mid \nu = v\}, h_1 \vdash e : \hat{\tau}_2/\hat{h}_2/\hat{\Sigma} \\ \Gamma_1 \vdash h_1 \qquad \Gamma \vdash \hat{\tau}_2/\hat{h}_2/\hat{\Sigma} \\ \Sigma = \hat{\Sigma} \oplus \{(\tilde{l} \mapsto \mathit{logEffect}(\mathrm{BB}(\nu) + n_k, \mathrm{Read}))\}_{n_k} \end{array}}{\Phi, \Gamma, h \vdash \textbf{letu } x = \textbf{unfold } v \textbf{ in } e : \hat{\tau}_2/\hat{h}_2/\Sigma}\ [\text{T-UNFOLD}]$$

$$\frac{h = h_0 * \tilde{l} \mapsto \hat{c}_1 * l_j \mapsto c_2 \qquad \Gamma \vdash c_2 <: \hat{c}_1}{\Phi, \Gamma, h \vdash \textbf{fold } L : \mathtt{void}/h_0 * \tilde{l} \mapsto \hat{c}_1/\varepsilon}\ [\text{T-FOLD}]$$

Effects Checking $\boxed{\Gamma \vdash OK(\Sigma_1, \Sigma_2)}$

$$\frac{\begin{array}{c} \Psi(E_1, E_2) = \pi(p_1, E_1) \wedge \pi(p_2, E_2) \\ \forall (E_1, E_2) \in (\mathbb{E} \times \mathbb{E}) \setminus \mathbb{C}.\mathsf{Unsat}([\![\Gamma]\!] \wedge \Psi(E_1, E_2)) \end{array}}{\Gamma \vdash OK(p_1, p_2)}$$

$$\frac{\forall l \in \mathit{Dom}(\Sigma_1 \oplus \Sigma_2). \Gamma \vdash OK(\mathrm{LU}(\Sigma_1, l), \mathrm{LU}(\Sigma_2, l))}{\Gamma \vdash OK(\Sigma_1, \Sigma_2)}$$

Effects Operations

$$\Sigma_1 \oplus \Sigma_2 \doteq \{l \mapsto \mathrm{LU}(\Sigma_1, l) \vee \mathrm{LU}(\Sigma_2, l)\}_{l \in \bigcup \mathit{Dom}(\Sigma_{1,2})}$$

$$\mathrm{LU}(\Sigma, l) \doteq \begin{cases} \Sigma(l) & l \in \mathit{Dom}(\Sigma) \\ \text{false} & \text{o.w.} \end{cases}$$

$$\pi(p, E) \doteq p[E \mapsto \mathrm{Check}] \wedge \mathrm{Check}(\nu)$$

$$\mathit{logEffect}(v, E) \doteq E(\nu) \wedge \nu = v \wedge_{E' \in \mathbb{E} \setminus \{E\}} \neg E'(\nu)$$

Figure 5. Typing Rules

several invariants that will prevent the type system from unsoundly assuming invariants in one thread which may be broken by the heap writes performed in another. We elaborate on this below.

Rule [T-FOREACH] types the **foreach** parallel loop expression. The loop induction variable i ranges over values between v_1 and v_2, exclusive; we type the body expression e in the environment enriched with an appropriate binding for i and compute the resulting heap and per-iteration effect Σ. We require that the heap is loop-invariant. To ensure that the behavior of the **foreach** loop is deterministic, we must check that the effects of distinct iterations do not interfere. Hence, we check non-interference at two arbitrary, distinct iterations by adding a binding for a fresh name j to the environment with the invariant that $i \neq j$, and verifying with OK that the effect at an iteration i, Σ, commutes with the effect at a distinct iteration j, $\Sigma[i \mapsto j]$. We return Σ', which subsumes Σ but is well-formed in the original environment, ensuring that the loop induction variable i does not escape its scope. As with [T-PAR], we require that the input heap is abstract, for the same reasons.

***Strong Updates and Concurrency*.** The rules [T-UNFOLD] and [T-FOLD] are used to implement a local non-aliasing discipline for performing strong updates on the types of heap locations when only one pointer to the location is accessed at a time. The mechanism is similar to freeze/thaw and adopt/focus [34, 10]. These rules are treated in previous work [28]; we now discuss how these rules handle effects and briefly recap their handling of strong updates.

Rule [T-UNFOLD] types the **letu** construct for unfolding a pointer to an abstract location, $\tilde{l}$, to obtain a pointer to a corresponding concrete location, l_j, bound to the variable x. [T-UNFOLD] constructs the block bound to l_j by creating a skolem variable x_j for each binding of a singleton index n_j to a type τ_j; the binding is a singleton index in a concrete location, so it corresponds to exactly one datum on the heap. We apply an appropriate substitution to the elements within the block, then typecheck the body expression e with respect to the extended heap and enriched environment. Well-formedness constraints ensure that an abstract location is never unfolded twice in the same scope and that x does not escape its scope.

We allow two concurrently-executing expressions to unfold, and thus simultaneously access, the same abstract location. When a location is unfolded, our type system records the refinement types bound to each of its singleton indices in the environment. If we do not take care, the refinement types bound to singleton indices and recorded in the environment may be invalidated by an effect (*e.g.* a write) occurring in a simultaneously-executing expression.Thus, an expression which unfolds a location *implicitly* depends on the data whose invariants are recorded in its environment at the time of unfolding. To capture these implicit dependencies, when a pointer v is unfolded, we conservatively record a pseudo-read [33] at each singleton offset n_k within the block into which v points by recording in the heap effect that the pointer $\mathrm{BB}(v)+n_k$ is read, where $\mathrm{BB}(v)$ is the beginning of the memory block where v points. Then, the invariant recorded in the environment at the unfolding is preserved, as any violating writes would get caught by the determinism check.

Rule [T-FOLD] removes a concrete location from the heap so that a different pointer to the same abstract location may be unfolded. The rule ensures that the currently-unfolded concrete location's block is a subtype of the corresponding abstract location's so that we can be sure that the abstract location's invariant holds when it is unfolded later. The **fold** expression has no heap effect.

***Program Typing*.** The remaining expression forms do not manipulate heap effects; as their typing judgments are straightforward and covered in previous work [28], we defer their definition [16]. We note that the rule for typing **if** expressions records the value of the branch condition in the environment when typing each branch. The judgments for typing function declarations and programs in LC_E are straightforward and are deferred to [16].

Program	LOC	Quals	Annot	Changes	T (s)
Reduce	39	7	7	N/A	8.8
SumReduce	39	1	3	0	1.8
QuickSort	73	3	4	N/A	5.9
MergeSort	95	6	7	0	32.4
IDEA	222	7	5	3	59.7
K-Means	458	7	10	16	63.7

Table 1. (LOC) is the number of source code lines as reported by *sloccount*, (Quals) is the number of logical qualifiers required to prove memory safety and determinism, (Annot) is the number of annotated function signatures plus the number of effect and commutativity declarations. (Changes) is the number of program modifications, (T) is the time in seconds taken for verification.

4.2 Handling Effects

We now describe the auxiliary definitions used by the typing rules of subsection 4.1 for checking effect noninterference.

***Combining Effects*.** In Figure 5, we define the $\oplus$ operator for combining effects. Our decision to encode effects as formulas in first-order logic makes the definition of this operator especially simple: for each location l, the combined effect of Σ_1 and Σ_2 on location l is the disjunction of of the effects on l recorded in Σ_1 and Σ_2. We use the auxiliary function $\mathrm{LU}(\Sigma, l)$ to look up the effect formula for location l in Σ; if there is no binding for l in Σ, the false formula, indicating no effect, is returned instead.

***Effects Checking*.** We check the noninterference of heap effects Σ_1, Σ_2 using the OK judgment, defined in Figure 5. The effects are noninterfering if the formulas bound to locations in both heap effects are pairwise noninterfering. Two effect formulas p_1 and p_2 are noninterfering as follows. For each pair of non-commuting effects E_1 and E_2 in the set $\mathbb{E}$, we project the set of pointers in p_1 (resp., p_2) which are accessed with effect E_1 (resp., E_2) using the effect projection operator π. Now, if the conjunction of the projected formulas is unsatisfiable, the effect formulas are noninterfering.

***User-Defined Effects and Commutativity*.** We allow our set of effects $\mathbb{E}$ to be extended with additional effect label predicates by the user. To specify how these effects interact, commutativity annotations can be provided that indicate which effects do not result in nondeterministic behavior when run simultaneously. Then, the user may override the effect of any function by providing an effect Σ as an annotation, allowing additional flexibility to specify domain-specific effects or override the effect system if needed.

5. Evaluation

We have implemented the techniques in this paper as an extension to CSOLVE, a refinement type-based verifier for C. CSOLVE takes as input a C program and a set of logical qualifiers, or formulas over the value variable ν, to use in refinement and effect inference. CSOLVE then checks the program both for memory safety errors (*e.g.* out-of-bounds array accesses, null pointer dereferences) and non-determinism. If CSOLVE determines that the program is free from such errors, it outputs an annotation file containing the types of program variables, heaps, functions, and heap effects. If CSOLVE cannot determine that the program is safe, it outputs an error message with the line number of the potential error and the inferred types of the program variables in scope at that location.

***Type and Effect Inference*.** CSOLVE infers refinement types and effect formulas using the predicate abstraction-based Liquid Types [28] technique; we give a high-level overview here. We note that there are expressions whose refinement types and heap effects cannot be constructed from the types and heap effects of subexpressions or from bindings in the heap and environment but instead

```
<region r1,r2,r3 | r1:* # r3:*, r2:* # r3:*> void
merge(DPJArrayInt<r1> a, DPJArrayInt<r2> b, DPJArrayInt<r3> c)
  reads r1:*, r2:* writes r3:* {
  if (a.length <= merge_size)
    seq_merge(a, b, c);
  else {
    int ha=a.length/2, sb=split(a.get(ha),b);
    final DPJPartitionInt<r1> a_split=new DPJPartitionInt<r1>(a,hd);
    final DPJPartitionInt<r2> a_split=new DPJPartitionInt<r2>(b,sb);
    final DPJPartitionInt<r3> c_split=new DPJPartitionInt<r3>(c,ha+sb);
    cobegin { merge(a_split.get(0),b_split.get(0),c_split.get(0));
              merge(a_split.get(1),b_split.get(1),c_split.get(1)); }}}
```

Figure 6. DPJ merge function

```
qualif Q(V: ptr) : &&[_ <= V; V {<, >=} (_ + _ + _)]

void merge(int* ARRAY LOC(L) a,
           int* ARRAY LOC(L) b,
           int la, int lb, int* ARRAY c) {
  if (la <= merge_size){
    seq_merge(a, b, la, lb, c);
  } else {
    int ha = la / 2, sb = split(a[ha],b,lb);
    cobegin {
      merge(a,b,ha,sb,c);
      merge(a+ha,b+sb,la-ha,lb-sb,c+ha+sb); }}}
```

Figure 7. CSOLVE merge function

must be *synthesized*. For example, the function typing rule requires us to synthesize types, heaps, and heap effects for functions. This is comparable to inferring pre- and post-conditions for functions (and similarly, loop invariants), which reduces to determining appropriate formulas for refinement types and effects. To make inference tractable, we require that formulas contained in synthesized types, heaps, and heap effects are *liquid*, *i.e.,* conjunctions of *logical qualifier* formulas provided by the user. These qualifiers are templates for predicates that may appear in types or effects. We read our inference rules as an algorithm for constructing subtyping constraints which, when solved, yield a refinement typing for the program.

Methodology. To evaluate our approach, we drew from the set of Deterministic Parallel Java (DPJ) benchmarks reported in [2]. For those which were parallelized using the methods we support, namely heap effects and commutativity annotations, we verified determinism and memory safety using our system. Our goals for the evaluation were to show that our system was as expressive as DPJ's for these benchmarks while requiring less program restructuring.

Results. The results of running CSOLVE on the following benchmarks are presented in Table 1. `Reduce` is the program given in section 2. `SumReduce` initializes an array using parallel iteration, then sums the results using a parallel divide-and-conquer strategy. `MergeSort` divides an array into quarters, recursively sorting each in parallel. It then merges pairs of sorted quarters in parallel, and finally merges the two sorted halves to yield a single sorted array. The final merges are performed recursively in parallel. `QuickSort` is a standard in-place quicksort adapted from a sequential version included with DPJ. We parallelized the algorithm by partitioning the input array around its median, then recursively sorting each partition in parallel. `K-Means` was adapted from the STAMP benchmarks [23], the C implementation that was ported for DPJ. `IDEA` is an encryption/decryption kernel ported to C from DPJ.

Changes. Some benchmarks required modifications in order to conform to the current implementation of CSOLVE; this does not indicate inherent limitations in the technique. These include: multi-dimensionalizing flattened arrays, writing stubs for matrix **malloc**, expanding structure fields (`K-Means`), and soundly abstracting non-linear operations and `#define`-ing configuration parameters to constants to work around a bug in the SMT solver (`IDEA`).

Annotations. CSOLVE requires two classes of annotations to compute base types for the program. First, the annotation `ARRAY` is required to distinguish between pointers to singletons and pointers to arrays. Second, as base type inference is intraprocedural, we must additionally specify which pointer parameters may be aliased by annotating them with a location parameter of the form `LOC(L)`.

Qualitative Comparison. Since the annotations used by CSOLVE and DPJ are very different, a quantitative comparison between them is not meaningful. Instead, we illustrate the kinds of annotations used by each tool and qualitatively compare them. Figures 6, 7 contain DPJ, and CSOLVE code, resp., implementing the recursive `Merge` routine from `MergeSort`, including region annotations and required wrapper classes. In the latter, `Merge` takes two arrays `a` and `b` and recursively splits them, sequentially merging them into array `c` at a target size. In particular, each statement in the cobegin writes to a different contiguous interval of `c`.

In DPJ, verifying this invariant requires: First, wrapping all arrays with an API class `DPJArrayInt`, and then wrapping each contiguous subarray with another class `DPJArrayPartitionInt`. This must be done because DPJ does not support precise effects over partitions of native Java arrays. Second, the method must be explicitly declared to be polymorphic over named regions `r1`, `r2`, and `r3`, corresponding to the memory locations in which the formals reside. Finally, `Merge` must be explicitly annotated with the appropriate effects, *i.e.,* it reads `r1` and `r2` and writes `r3`.

In CSOLVE, verifying this invariant requires: First, specifying that `a` and `b` are potentially aliased arrays (the annotation `LOC(L) ARRAY`). Second, we specify the qualifiers used to synthesize refinement types and heap effects via the line starting with `qualif`. This specification says that a predicate of the given form may appear in a type or effect, with the wildcard _ replaced by any program identifier in scope for that type or effect. Using the given qualifier, CSOLVE infers that `a` and `b`'s location is only read, and that `c`'s location is only written at indices described by the formula $c \leq \nu < c + la + lb$, which suffices to prove determinism.

Unlike DPJ, CSOLVE does not require invasive changes to code (*e.g.* explicit array partitioning), and hence supports domain- and program-specific sharing patterns (*e.g.* memory coalescing from Figure 2). However, this comes at the cost of providing qualifiers. In future work, abstract interpretation may help lessen this burden.

6. Related Work

We limit our discussion to static mechanisms for checking and enforcing determinism, in particular the literature that pertains to reasoning about heap disjointness: type-and-effect systems, semantic determinism checking, and other closely related techniques.

Named Regions. Region-based approaches assign references (or objects) to distinct segments of the heap, which are explicitly named and manipulated by the programmer. This approach was introduced in the context of adding impure computations to a functional language, and developed as a means of controlling the side-effects incurred by segments of code in sequential programs [20, 21]. Effects were also investigated in the context of safe, programmer-controlled memory management [31, 15, 19]. Ideas from this work led to the notion of abstract heap locations [34, 13, 10] and our notion of fold and unfold.

Ownership Types. In the OO setting, regions are closely related to *ownership types* which use the class hierarchy of the program to separate the heap into disjoint, nested regions [9, 29]. In addition to isolation, ownership types can be used to track effects [8], and to reason about data races and deadlocks [6, 4, 22].

Such techniques can be used to enforce determinism [30], but regions and ownership relations are not enough to enforce fine-grained separation. Instead, we must precisely track relations between program variables. We are inspired by DPJ [2], which shows how some sharing patterns can be captured in a dependent region system. However, we show the full expressiveness of refinement type inference and SMT solvers can be brought to bear to enable complex, low-level sharing with static determinism guarantees.

Checking Properties of Multithreaded Programs. Several authors have looked into type-and-effect systems for checking other properties of multithreaded programs. For example, [11, 24] show how types can be used to prevent races, [12] describes an effect discipline that encodes Lipton's Reduction method for proving atomicity. Our work focuses on the higher-level semantic property of determinism. Nevertheless, it would be useful to understand how race-freedom and atomicity could be used to establish determinism. Others [27, 3] have looked at proving that different blocks of operations *commute*. In future work, we could use these to automatically generate effect labels and commutativity constraints.

Program Logics and Abstract Interpretation. There is a vast literature on the use of logic (and abstract interpretation) to reason about sets of addresses (*i.e.,* the heap). The literature on logically reasoning about the heap includes the pioneering work in TVLA [35], separation logic [26], and the direct encoding of heaps in first-order logic [17, 18], or with explicit sets of addresses called dynamic frames [14]. These logics have also been applied to reason about concurrency and parallelism: [25] looks at using separation logic to obtain (deterministic) parallel programs, and [35] uses abstract interpretation to find loop invariants for multithreaded, heap-manipulating Java programs. The above look at analyzing disjointness for linked data structures. The most closely related to our work is [32], which uses intra-procedural numeric abstract interpretation to determine the set of array indices used by different threads, and then checks disjointness over the domains.

Our work differs in that we show how to consolidate all the above lines of work into a uniform location-based heap abstraction (for separation logic-style disjoint structures) with type refinements that track finer-grained invariants. Unlike [25], we can verify access patterns that require sophisticated arithmetic reasoning, and unlike [32] we can check separation between disjoint structures, and even indices drawn from compound structures like arrays, lists and so on. Our type system allows "context-sensitive" reasoning about (recursive) procedures. Further, first-order refinements allow us to verify domain-specific sharing patterns via first class effect labels [21].

References

[1] Nvidia cuda programming guide.

[2] S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA*, 2009.

[3] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ASPLOS*, 2009.

[4] Z. R. Anderson, D. Gay, R. Ennals, and E. A. Brewer. Sharc: checking data sharing strategies for multithreaded c. In *PLDI*, 2008.

[5] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[6] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, 2002.

[7] M. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal: Nested data parallelism in haskell. In *Euro-Par*, 2001.

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

[9] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *ECOOP*, 2001.

[10] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.

[11] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI*, pages 219–232, 2000.

[12] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003.

[13] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, 2002.

[14] B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A programming model for concurrent object-oriented programs. *TOPLAS*, 2008.

[15] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX*, 2002.

[16] M. Kawaguchi, P. Rondon, A. Bakst, and R. Jhala. Liquid effects: Technical report. `http://goto.ucsd.edu/~rjhala/liquid`.

[17] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL*, 2008.

[18] S. K. Lahiri, S. Qadeer, and D. Walker. Linear maps. In *PLPV*, 2011.

[19] C. Lattner and V. S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, 2005.

[20] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects, 2002.

[21] D. Marino and T. D. Millstein. A generic type-and-effect system. In A. Kennedy and A. Ahmed, editors, *TLDI*, pages 39–50. ACM, 2009.

[22] J.-P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro. Dynamically checking ownership policies in concurrent c/c++ programs. In *POPL*, pages 457–470, 2010.

[23] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[24] P. Pratikakis, J. S. Foster, and M. W. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*, 2006.

[25] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP*, pages 348–362, 2009.

[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[27] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *TOPLAS*, 19(6), 1997.

[28] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.

[29] M. Smith. Towards an effects system for ownership domains. In *In ECOOP Workshop - FTfJP 2005*, 2005.

[30] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *TOPLAS*, 30, 2008.

[31] M. Tofte and J.-P. Talpin. A theory of stack allocation in polymorphically typed languages, 1993.

[32] M. T. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic verification of determinism for structured parallel programs. In *SAS*, 2010.

[33] J. Voung, R. Chugh, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using data race detection. In *PLDI*, 2008.

[34] D. Walker and J. Morrisett. Alias types for recursive data structures. 2000.

[35] E. Yahav and M. Sagiv. Verifying safety properties of concurrent heap-manipulating programs. *TOPLAS*, 32(5), 2010.