A Type and Effect System for Determinism in Multithreaded Programs

Yi Lu[1], John Potter[1], Chenyi Zhang[2], and Jingling Xue[1]

[1] University of New South Wales
{ylu,potter,jingling}@cse.unsw.edu.au
[2] University of Queensland
chenyi@uq.edu.au

Abstract. There has been much recent interest in supporting deterministic parallelism in imperative programs. Structured parallel programming models have used type systems or static analysis to enforce determinism by constraining potential interference of lexically scoped tasks. But similar support for multithreaded programming, where threads may be ubiquitously spawned with arbitrary lifetimes, especially to achieve a modular and manageable combination of determinism and nondeterminism in multithreaded programs, remains an open problem.
This paper proposes a simple and intuitive approach for tracking thread interference and capturing both determinism and nondeterminism as computational effects. This allows us to present a type and effect system for statically reasoning about determinism in multithreaded programs. Our general framework may be used in multithreaded languages for supporting determinism, or in structured parallel models for supporting threads. Even more sophisticated concurrency models, such as actors, are often implemented on top of an underlying threading model, thus the underlying ideas presented here should be of value in reasoning about the correctness of such implementations.

1 Introduction

Concurrent programming is increasingly pervasive in mainstream software development as we attempt to exploit the full power of modern multicore processors. For developers working in heavily used languages such as C, C++, C# and Java, threads provide the dominant model for concurrent programming. Threads are a straightforward adaptation of the sequential model of computation to concurrent programs; programming languages require little or no syntactic changes to support them and modern computers and operating systems have evolved to efficiently support them. However, *they discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism* [13]. Threads may interact and share data in a myriad of ways; unanticipated thread interleavings and side effects can lead to subtle bugs and wrong results [24]. It is widely acknowledged that multithreaded programming is difficult and error-prone [21].

Deterministic code is easier to debug and verify than nondeterministic code for both testing and formal methods [13, 25]. Moreover, many computational algorithms are required to be deterministic—a given input is always expected to produce the same output—even when used in nondeterministic contexts. For example, most scientific computing, encryption/decryption, sorting, program analysis, and processor simulation algorithms exhibit deterministic behavior [1]. There has been much recent interest in supporting deterministic algorithms in multithreaded imperative programs. Programming models and type systems that support deterministic parallelism are emerging and have been proved to be useful in practice. Structured deterministic parallel models [1, 26, 2] enforce determinism statically at compile time by disallowing interference between concurrent computations (conflicting in their read and write sets); they typically rely on lexically scoped task parallelism (e.g., the fork/join style [12]) to localise and constrain potential interference of parallel tasks. While effective for applications where parallelism is often regular, structured parallelism may restrict programming styles thereby precluding many useful concurrency patterns which require irregular parallelism—*the world of client applications is not nearly as well structured and regular* [24]. Moreover, it may be difficult to support these models in mainstream programming languages where threads are used, because structured tasks can suffer interference from independent threads which may be ubiquitously spawned with arbitrary lifetimes. Reasoning statically about threads is a challenging problem, because we have to capture potential interference amongst all concurrent threads [13].

This paper proposes a simple and intuitive approach for type checking determinism in multithreaded imperative programs. By tracking and controlling what side effects may occur in parallel and how they may interfere with one another, our type system can check that deterministic behaviour is preserved in otherwise nondeterministic contexts. Programmers can freely assert that sections of code are deterministic, and have those assertions guaranteed by the type checker. Thus deterministic algorithms will assuredly produce deterministic results even in a nondeterministic context, making multithreaded programs more predictable and understandable. Our framework is general and may be used in existing multithreaded programming languages for supporting determinism, or in previous structured deterministic parallel models for supporting threads (e.g. Deterministic Parallel Java's region-based effect systems [1, 2] can be extended with our approach to support multithreaded programming).

We formalise our approach in a novel type and effect system, called *Deterministic Effects*, as an extension to Nielson et al.'s effect systems [18, 19]. Our small step operational semantics describes the concurrent behaviour of programs; crucially, the operational semantics preserves the relative nesting of thread creations. We prove that programmer-specified determinism for type correct multithreaded programs is indeed guaranteed. Deterministic effects can help improve the design and understanding of multithreaded software, such as in specifying or documenting concurrent behaviour for safety and optimisation. We envisage their use in interface specifications, facilitating modular development of largescale concurrent programs. We also envisage static analyses for discovering determinism of expressions in existing multithreaded programs.

2 Deterministic Effects at a Glance

In this section, we introduce the basic ideas of deterministic effects with simple examples by showing how to track computational effects (specifically, side effects, though other effects may be possible, see Section 4.3) that may occur concurrently, how to capture determinism in nondeterministic code, and how to enforce programmer-specified deterministic expressions.

Deterministic effects capture the conflict between any two effects which may occur concurrently, by tracking the effects of forked threads in a form that mimics the tree structure of thread creation and then comparing these effects with the effects of any subsequent expressions to detect potential interference in a flow-sensitive manner. In general, earlier forks have more opportunity for interference than later forks. In Section 2.1, we introduce the syntax and demonstrate the use of deterministic effects to forbid all possible thread interference in a program. Programs that are well-typed have purely deterministic behaviours.

However, there are algorithms that may not have deterministic input-output behaviour. Real world applications are more likely composed of a mixture of deterministic and nondeterministic computations [2, 25]. Achieving a manageable combination of determinism and nondeterminism is an important open problem in multithreaded programming. In Section 2.2, by reasoning about thread interference, we show how to infer determinism or nondeterminism of each expression in a possibly nondeterministic program. We extend our syntax with a programmer-specified deterministic construct, which is essentially the same as those used in [22, 5], except that they check determinism dynamically at runtime while we enforce it statically at compile time (see discussion in Section 4.2). With this more liberal model, we show that we can allow arbitrary mixing of deterministic and nondeterministic computations in a safe way—guaranteeing desired determinism without unnecessarily restricting nondeterministic code.

2.1 Effects, Noninterference and Determinism

Our effect system extends Nielson et al.'s framework [18] for capturing interference between threads. We use a simple lambda calculus, similar to that used in [19, 9, 21, 17], to provide the essential features of concurrent threads sharing access to memory including memory references and fork expressions. We describe the form of effects, and provide some simple examples to illustrate how to reason about thread interference. Memory reads and writes give rise to conventional sequential side effects based on regions determined by labels associated with memory allocation expressions. Threads are spawned by fork expressions, and their effects are represented separately from sequential side effects. By tracking the effects of threads spawned by a computation separately from its sequential continuation, we are able to use a flow-sensitive analysis to identify when the threads may interfere with the continuation. Despite its apparent simplicity, this approach appears to be novel.

The basic syntax is given by:

$$v ::= c \mid \textbf{fn}\ x \Rightarrow e$$
$$e ::= v \mid x \mid e\ e \mid \textbf{ref}_\pi\ e \mid !e \mid e := e \mid \textbf{fork}\ e$$

The expression syntax e includes *values* v (a family of *integer constants* c and *functions*), a family of *variables* x, *function application* and the usual imperative operations on reference cells (*allocation*, *dereference* and *assignment*). The allocation expression $\textbf{ref}_\pi\ e$ creates a new reference in memory and initialises it to the value of e, and the *label* (or *abstract location*) π uniquely identifies the creation point/allocation site. Common language constructs such as let, sequence or recursion are not directly defined, because they are easily encoded. The let expression $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ becomes $(\textbf{fn}\ x \Rightarrow e_2)\ e_1$, and the sequence expression $e_1; e_2$ is $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ where x does not occur free in e_2. In examples, we will use explicit let and sequence expressions as shorthands. Like [18], which we extend, conditionals are omitted, as their treatment is standard. Recursive functions can be encoded by references as shown in [9, 17] and in Section 4.1. Threads are introduced by **fork** e, which spawns a new thread for the evaluation of e; the value of e is never used (this expression simply returns a zero when evaluated [9, 3]), so a thread is only used for side effects.

The syntax for types and effects is given by:

$$\tau ::= int \mid \tau \xrightarrow{\varphi} \tau \mid ref_\rho\ \tau \qquad \varepsilon ::= !\rho \mid \rho :=$$
$$\varphi ::= \{\varepsilon\} \mid \{fork\ \varphi\} \mid \varphi \cup \varphi \mid \emptyset \qquad \rho ::= \{\pi\} \mid \rho \cup \rho \mid \emptyset$$

The syntax for types includes a primitive integer type, as well as function and reference type. The function type is annotated with effect information, φ, which captures concurrent effects that may occur during the execution of the function body. The reference type $ref_\rho\ \tau$ records the type τ of values that can be stored in the reference cell, and a *region* ρ where the cell may be created. The region syntax is used to denote finite sets of creation points or labels, which is a standard technique to distinguish cells created at different program points. The union operator $\cup$ is assumed to be commutative, associative and idempotent, with $\emptyset$ as its identity; $\{\pi_1\} \cup \ldots \cup \{\pi_n\}$ may be written as $\{\pi_1, \ldots \pi_n\}$. A *side effect* ε identifies a read $!\rho$ or a write $\rho :=$ to a region ρ.

A *fork effect*, *fork* φ, captures the effect of a forked thread. This allows us to syntactically distinguish the effect of the current expression from other fork effects arising from earlier forks, when determining thread interference. For example, the effect $\{!\{\pi_1\}, fork\ \{\{\pi_2\} :=\}\}$ describes a read effect $!\{\pi_1\}$ on the current thread, and a write effect $\{\pi_2\} :=$ on a thread spawned by the current thread, as identified by the keyword *fork*. The order of side effects and fork effects in an effect set (φ) is not important, but we track expression compositions flow-sensitively to identify potential interference between components (see examples below). In addition to π, we sometimes use ϖ, "var pi", to denote labels for program points in our examples.

We illustrate our ideas with a simple value setter example:

$$\begin{array}{l}\textbf{let}\ \ \mathit{val} = \textbf{ref}_\pi\ 0\ \ \textbf{in}\ \ \textbf{let}\ \ \mathit{set} = (\textbf{fn}\ x \Rightarrow \mathit{val} := x)\ \ \textbf{in}\\ \quad \textbf{fork}\ (\mathit{set}\ 2);\quad !\mathit{val} \qquad\qquad\qquad //\ \mathit{error}\end{array}$$

The types and effects for some of the expressions here, are

$$\begin{array}{ll} \mathit{val} : \mathit{ref}_\pi\ \mathit{int} \qquad\quad \&\ \emptyset & \quad \begin{array}{ll}\mathit{set}\ 2 & : \mathit{int}\ \&\ \{\{\pi\}:=\}\\ \textbf{fork}\ (\mathit{set}\ 2) & : \mathit{int}\ \&\ \{\mathit{fork}\ \{\{\pi\}:=\}\}\\ !\mathit{val} & : \mathit{int}\ \&\ \{!\{\pi\}\}\end{array} \\ \mathit{set} : \mathit{int} \xrightarrow{\{\{\pi\}:=\}} \mathit{int}\ \&\ \emptyset & \end{array}$$

The type of the variable *val* is determined by the allocation expression used to initialise it; the label π records the program point where the reference cell is allocated. Evaluating read-only variables has no memory effect, $\emptyset$. The function type of the variable *set* records its write effect on *val* for use in function application. This is manifested in the effect of the expression *set* 2. The effect of the dereference $!\mathit{val}$ is the simple read effect $\{!\{\pi\}\}$.

In most type and effect systems, **fork** (*set* 2) has the same effect as *set* 2, namely $\{\{\pi\} :=\}$. Hence they may not tell if this code is safe. For example, if we swap the order of the dereference and fork above, as in

$$!\mathit{val};\quad \textbf{fork}\ (\mathit{set}\ 2) \qquad //\ \mathit{ok}$$

there is no interference. This illustrates what we mean by our system being *flow sensitive*: in general, moving forks earlier in a sequential thread is more likely to make a program illegal.

Since our system distinguishes an effect on one thread from that on another, the effect of **fork** (*set* 2) is $\{\mathit{fork}\ \{\{\pi\} :=\}\}$—the fork effect captures the write effect on location $\{\pi\}$ and the *fork* indicates that it occurs on a different thread. For the original version of the example, the fork occurs before the dereference. Our flow-sensitive type and effect system recognises that the write effect $\{\mathit{fork}\ \{\{\pi\} :=\}\}$ for the fork may occur concurrently with the read effect $\{!\{\pi\}\}$ on the current thread, and considers this illegal. The second version of the example with reverse order of effects is legal in our system; we recognise that the read effect has completed before the conflicting fork. In our type system, this flow sensitivity is captured by checking sequential composition of effects for noninterference in Table 2 in Section 3.

Now consider an example with a thread forked inside a function:

$$\begin{array}{l}\textbf{let}\ \ x = \textbf{ref}_\pi\ 0\ \ \textbf{in}\ \ \textbf{let}\ \ y = \textbf{ref}_\varpi\ 0\ \ \textbf{in}\\ \textbf{let}\ \ f = (\textbf{fn}\ z \Rightarrow \textbf{fork}\ (x := 1);\ y := 1)\ \ \textbf{in}\\ \quad x := 2;\quad f\ 0;\quad y := 2 \qquad\qquad //\ \mathit{ok}\end{array}$$

This example shows no interference in our effect system, but may not pass, for example, type and effect systems for enforcing locking disciplines [9, 3, 21] which would conservatively dictate all memory accesses must be protected by a lock. Consider the effects recorded for the last three expressions. $x := 2$ is a sequential expression with effect $\{\{\pi\} :=\}$. It can never interfere with any later expressions.

$f\ 0$ may look like a sequential expression, but the body of f will fork a thread to access the shared variable x. Clearly $f\ 0$ writes to ϖ and creates a thread that writes to π; in our system its concurrent effect is $\{\{\varpi\}{:=}, \mathit{fork}\ \{\{\pi\}:=\}\}$. The effect of the third expression $y := 2$ is $\{\{\varpi\}:=\}$. The last two expressions do not interfere, because we only need to consider the concurrent part of the effect of the earlier expression, which is $\{\mathit{fork}\ \{\{\pi\}:=\}\}$, and the overall effect of the later expression, which is $\{\{\varpi\}:=\}$. (The sequential part of the second expression with effect $\{\{\varpi\}:=\}$ completes before the third expression starts.) The write effects $\{\mathit{fork}\ \{\{\pi\}:=\}\}$ and $\{\{\varpi\}:=\}$ do not interfere with each other since they access different reference cells.

Unsurprisingly, threads that concurrently read the same memory location do not interfere, as shown in the following example:

```
let  x = ref_π 1  in  let  y = ref_ϖ 2  in
let  lim = (fn z => 100−!z)  in
  fork (lim x);   fork (lim y);   fork (lim x)      // ok
```

The function *lim* reverses a count from 100; its type and effect is:

$$\mathit{ref}_{\{\pi,\varpi\}}\ \mathit{int} \xrightarrow{\{!\{\pi,\varpi\}\}} \mathit{int}\ \&\ \emptyset$$

These threads do not interfere with each other because concurrent reads on the same memory region are considered safe. This example also shows the use of regions (rather than single labels) in reference types so that function arguments of reference type may be associated with multiple labels, in the absence of label polymorphism (see Section 4.1).

2.2 Deterministic Effects with Nondeterministism

We have seen how to enforce determinism by forbidding all thread interference. Programs which are well-typed, using our type rules, restrict concurrent computations to be purely deterministic. Now we introduce a more liberal model which allows arbitrary interleaving of threaded expressions, no matter what their potential interference may be. This provides a nondeterministic model of behaviour, typical of how threading works in current languages. However we allow (and enforce) explicit embedding of deterministic expressions. By tracking thread interference within expressions, we can capture the determinism or nondeterminism of expressions as effects.

If a stand-alone expression exhibits no interference in its evaluation, then its behaviour is deterministic. We say in that case the expression is *weakly deterministic*. However, if that deterministic expression is evaluated in a concurrent context in which other threads may interfere with it, the behaviour is no longer guaranteed to be deterministic. In other words, determinism of expression evaluation is not preserved by concurrent composition. This is partly in the nature of shared memory concurrency, but is unfortunate—if we have a deterministic algorithm we want to preserve its determinism even when it is embedded in a larger, possibly nondeterministic computation. Our solution is to allow a programmer to express this desire by declaring an expression to be deterministic using **det** (see extended syntax below). In that case, our system will firstly check that the expression is weakly deterministic. In addition, our system will enforce its determinism by insisting that any concurrent context where that expression appears cannot interfere with it. We say such expressions are *strongly deterministic.*

In this section we show how we can relax our earlier model, to allow both nondeterministic forms of expression, and also enforce strong determinism for expressions so declared with the following extended syntax:

$$e \quad ::= \quad ... \mid \mathbf{det}\ e$$

Based on thread interference, the levels of determinism on expressions are distinguished by our effects system (from the weakest to the strongest): *nondeterministic*, *weakly deterministic* and *strongly deterministic.* A nondeterministic expression is allowed to contain interference, except that such interference does not affect any of its subexpressions which are asserted to be strongly deterministic. An expression is weakly deterministic if it does not contain interference, but may suffer interference from other threads; in other words, it is deterministic by itself, but may not be if used in a nondeterministic context. A strongly deterministic expression is weakly deterministic and must not be interfered with.

We start with a few examples before introducing our deterministic effect system in the next section.

$$\begin{array}{l}\mathbf{let}\ \ x = \mathbf{ref}_{\pi}\ 0\ \ \mathbf{in}\ \ \mathbf{let}\ \ y = \mathbf{ref}_{\varpi}\ 1\ \ \mathbf{in} \\ \quad \mathbf{fork}(x := 1); \quad \mathbf{fork}(x := !y); \quad \mathbf{det}\ y := 2 \qquad //\ \mathit{ok}\end{array}$$

The above program is nondeterministic, as it allows the value of y to interfere with the value of x. This is legal in our type and effect system, since the value of y, which is asserted as strongly deterministic, is not affected by its context. The following program is illegal.

$$\begin{array}{l}\mathbf{let}\ \ x = \mathbf{ref}_{\pi}\ 0\ \ \mathbf{in}\ \ \mathbf{let}\ \ y = \mathbf{ref}_{\varpi}\ 1\ \ \mathbf{in} \\ \quad \mathbf{fork}(x := 1); \quad \mathbf{fork}(x := !y); \quad \mathbf{det}\ y := !x \qquad //\ \mathit{error}\end{array}$$

We can also allow threads to be forked within **det** expressions. In both of the following examples, the fork expression may run concurrently with the last (nondeterministic) expression. In both examples, the value of x is guaranteed to be 2, and that of y may be 0 or 2. In the second example, because the computation for y is declared to be **det** this must be illegal. In that case the nondeterministic write to x interferes with the read of x inside the **det** expression. Note that in the first example, the read of x does not interfere with the write; this illustrates the asymmetry of the noninterference relation.

$$\begin{array}{l}\mathbf{let}\ \ x = \mathbf{ref}_{\pi}\ 0\ \ \mathbf{in}\ \ \mathbf{let}\ \ y = \mathbf{ref}_{\varpi}\ 1\ \ \mathbf{in} \\ \quad \mathbf{det}\ \mathbf{fork}(x := 2); \quad y := !x \qquad\qquad\qquad //\ \mathit{ok}\end{array}$$

$$\begin{array}{l}\mathbf{let}\ \ x = \mathbf{ref}_{\pi}\ 0\ \ \mathbf{in}\ \ \mathbf{let}\ \ y = \mathbf{ref}_{\varpi}\ 1\ \ \mathbf{in} \\ \quad \mathbf{det}\ \mathbf{fork}(y := !x); \quad x := 2 \qquad\qquad\qquad //\ \mathit{error}\end{array}$$

Any purely deterministic program is a special case of this more flexible system, in which the top-level program expression is declared to be **det**.

$$[\text{CONSTANT}] \quad \Gamma \vdash c : \mathit{int} \ \&\ \emptyset.\bot \qquad\qquad [\text{VARIABLE}] \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \ \&\ \emptyset.\bot}$$

$$[\text{FUNCTION}] \quad \frac{\Gamma, x \mapsto \tau_2 \vdash e : \tau_1 \ \&\ \Delta \qquad x \notin \mathit{dom}(\Gamma)}{\Gamma \vdash \mathbf{fn}\ x \texttt{=>}\ e : \tau_2 \xrightarrow{\Delta} \tau_1 \ \&\ \emptyset.\bot}$$

$$[\text{APPLICATION}] \frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\Delta_3} \tau_1 \ \&\ \Delta_1 \quad \Gamma \vdash e_2 : \tau_2 \ \&\ \Delta_2 \quad \Delta_1 \ ;\ \Delta_2 \ ;\ \Delta_3 = \Delta}{\Gamma \vdash e_1\ e_2 : \tau_1 \ \&\ \Delta}$$

$$[\text{REFERENCE}] \quad \frac{\Gamma \vdash e : \tau \ \&\ \Delta}{\Gamma \vdash \mathbf{ref}_\pi\ e : \mathit{ref}_{\{\pi\}}\ \tau \ \&\ \Delta}$$

$$[\text{DEREFERENCE}] \quad \frac{\Gamma \vdash e : \mathit{ref}_\rho\ \tau \ \&\ \Delta_1 \qquad \Delta_1 \ ;\ \{!\rho\}.\bot = \Delta}{\Gamma \vdash !e : \tau \ \&\ \Delta}$$

$$[\text{UPDATE}] \quad \frac{\Gamma \vdash e_1 : \mathit{ref}_\rho\ \tau \ \&\ \Delta_1 \quad \Gamma \vdash e_2 : \tau \ \&\ \Delta_2 \quad \Delta_1 \ ;\ \Delta_2 \ ;\ \{\rho :=\}.\bot = \Delta}{\Gamma \vdash e_1 := e_2 : \tau \ \&\ \Delta}$$

$$[\text{FORK}] \ \frac{\Gamma \vdash e : \tau \ \&\ \varphi.\theta}{\Gamma \vdash \mathbf{fork}\ e : \mathit{int} \ \&\ \{\mathit{fork}\ \varphi\}.\theta} \qquad [\text{DET}] \ \frac{\Gamma \vdash e : \tau \ \&\ \varphi.\bot}{\Gamma \vdash \mathbf{det}\ e : \tau \ \&\ \{\mathit{det}\ \varphi\}.\bot}$$

$$[\text{SUBSUMPTION}] \quad \frac{\Gamma \vdash e : \tau' \ \&\ \Delta' \qquad \tau' <: \tau \qquad \Delta' \sqsubseteq \Delta}{\Gamma \vdash e : \tau \ \&\ \Delta}$$

Table 1. Static Semantics

3 A Deterministic Effect System

In this section, we formalise our approach in a type and effect system by using the syntax introduced in the previous section. Since we now have the additional **det** e expression, we need to extend the syntax for types and effects:

$$\varphi ::= \dots \mid \{\mathit{det}\ \varphi\} \qquad\qquad \Delta ::= \varphi.\theta$$
$$\tau ::= \dots \mid \tau \xrightarrow{\Delta} \tau \qquad\qquad \theta ::= \bot \mid \top$$

The new form of effect for expressions is the *deterministic effect* Δ. We now refer to φ as *base effect*. The additional part of an effect, θ, refers to the inferred level of determinism for expressions—$\bot$ denotes a weakly deterministic effect and $\top$ denotes a nondeterministic effect with ordering, $\bot \leq \top$, so that weakly deterministic effects are more specific. An expression is strongly deterministic if it is syntactically annotated with the keyword **det**—it must have a qualified effect *det* φ which serves as a contract for the type system to ensure that any concurrent context where this expression appears cannot interfere with it.

3.1 Static Semantics

Effect systems [16] generally can be viewed as a type system where typing judgements have a more elaborate form, associating both a type and an effect with expressions. For the syntax defined in Section 2, Table 1 presents our type system for deterministic effects, using judgements of the form

$$\Gamma \vdash e : \tau \ \&\ \Delta$$

[COMPOSITION]	$\dfrac{\varphi_1 \hookrightarrow_\theta \varphi_2 \qquad \theta_1 \le \theta \qquad \theta_2 \le \theta}{\varphi_1.\theta_1 \ ; \ \varphi_2.\theta_2 = (\varphi_1 \cup \varphi_2).\theta}$		
[C-EMP]	$\emptyset \hookrightarrow_\theta \varphi$	[C-DET-S]	$\dfrac{\varphi_1 \hookrightarrow_\bot \varphi_2}{\{det\ \varphi_1\} \hookrightarrow_\bot \varphi_2}$
[C-EFF]	$\{\varepsilon\} \hookrightarrow_\theta \varphi$	[C-FOK-S]	$\dfrac{\varphi_1 \not\twoheadrightarrow_\bot \varphi_2 \qquad \varphi_2 \not\twoheadrightarrow_\bot \varphi_1}{\{fork\ \varphi_1\} \hookrightarrow_\bot \varphi_2}$
[C-UNI]	$\dfrac{\varphi_1 \hookrightarrow_\theta \varphi \qquad \varphi_2 \hookrightarrow_\theta \varphi}{\varphi_1 \cup \varphi_2 \hookrightarrow_\theta \varphi}$	[C-FOK-W]	$\dfrac{\varphi_1 \not\twoheadrightarrow_\top \varphi_2 \qquad \varphi_2 \not\twoheadrightarrow_\top \varphi_1}{\{fork\ \varphi_1\} \hookrightarrow_\top \varphi_2}$
[C-DUN]	$\dfrac{\{det\ \varphi_1\} \hookrightarrow_\theta \varphi \quad \{det\ \varphi_2\} \hookrightarrow_\theta \varphi}{\{det\ \varphi_1 \cup \varphi_2\} \hookrightarrow_\theta \varphi}$	[C-DET-W]	$\dfrac{\varphi_1 \not\twoheadrightarrow_\top \varphi_2 \qquad \varphi_2 \not\twoheadrightarrow_\bot \varphi_1}{\{det\ \{fork\ \varphi_1\}\} \hookrightarrow_\top \varphi_2}$

Table 2. Composition and Sequencing Rules

[N-R/R]	$\{!\rho\} \not\twoheadrightarrow_\bot \varphi$		
[N-W/R]	$\dfrac{\rho_1 \cap \rho_2 = \emptyset}{\{\rho_1 :=\} \not\twoheadrightarrow_\bot \{!\rho_2\}}$	[N-FOL]	$\dfrac{\varphi_1 \not\twoheadrightarrow_\theta \varphi_2}{\{fork\ \varphi_1\} \not\twoheadrightarrow_\theta \varphi_2}$
[N-W/W]	$\dfrac{\rho_1 \cap \rho_2 = \emptyset}{\{\rho_1 :=\} \not\twoheadrightarrow_\bot \{\rho_2 :=\}}$	[N-FOR]	$\dfrac{\varphi_1 \not\twoheadrightarrow_\theta \varphi_2}{\varphi_1 \not\twoheadrightarrow_\theta \{fork\ \varphi_2\}}$
[N-EFF]	$\{\varepsilon_1\} \not\twoheadrightarrow_\top \{\varepsilon_2\}$	[N-UNL]	$\dfrac{\varphi_1 \not\twoheadrightarrow_\theta \varphi \qquad \varphi_2 \not\twoheadrightarrow_\theta \varphi}{\varphi_1 \cup \varphi_2 \not\twoheadrightarrow_\theta \varphi}$
[N-DEL]	$\dfrac{\varphi_1 \not\twoheadrightarrow_\theta \varphi_2}{\{det\ \varphi_1\} \not\twoheadrightarrow_\theta \varphi_2}$	[N-UNR]	$\dfrac{\varphi \not\twoheadrightarrow_\theta \varphi_1 \qquad \varphi \not\twoheadrightarrow_\theta \varphi_2}{\varphi \not\twoheadrightarrow_\theta \varphi_1 \cup \varphi_2}$
[N-DER]	$\dfrac{\varphi_1 \not\twoheadrightarrow_\bot \varphi_2}{\varphi_1 \not\twoheadrightarrow_\theta \{det\ \varphi_2\}}$	[N-EMP]	$\emptyset \not\twoheadrightarrow_\theta \varphi$

Table 3. Noninterference Rules

where τ is the type associated with the expression e relative to a type environment Γ which provides the type for each free variable as in a standard type system. The environment Γ is empty for top-level expressions which may therefore have no free variables; environment extension is written as $\Gamma, x \mapsto \tau$. The deterministic effect Δ describes the base effects that may take place during evaluation as well as its level of determinism (weakly deterministic or nondeterministic).

The rules in Table 1 infer typing judgements for each syntactic form of expression, together with the standard [SUBSUMPTION] rule using the definitions for subtyping and subeffects in Table 4. In Table 1, the inferred base effect φ is either empty, composed of effects of subexpressions, or generates new base effects. The rules [CONSTANT], [FUNCTION] and [VARIABLE] all have empty effect. The primitive side effects ($\{\varepsilon\}.\bot$) generated in [DEREFERENCE] and [UPDATE], as well as empty effects, are weakly deterministic.

In [FUNCTION], function abstraction records the (potential) deterministic effect of its body as an effect annotation in its type; it extends the environment for typing the body with a type for its variable (our types need not be unique).

[T-RFL]	$\tau <: \tau$		
[T-REF]	$\dfrac{\rho_1 \subseteq \rho_2}{ref_{\rho_1}\ \tau <: ref_{\rho_2}\ \tau}$	[F-R/W]	$\dfrac{\rho_1 \subseteq \rho_2}{\{!\ \rho_1\} \sqsubseteq \{\rho_2 :=\}}$
[T-FUN]	$\dfrac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2' \qquad \Delta \sqsubseteq \Delta'}{\tau_1 \xrightarrow{\Delta} \tau_2 <: \tau_1' \xrightarrow{\Delta'} \tau_2'}$	[F-W/W]	$\dfrac{\rho_1 \subseteq \rho_2}{\{\rho_1 :=\} \sqsubseteq \{\rho_2 :=\}}$
[E-SUB]	$\dfrac{\varphi_1 \sqsubseteq \varphi_2 \qquad \theta_1 \leq \theta_2}{\varphi_1.\theta_1 \sqsubseteq \varphi_2.\theta_2}$	[F-FOK]	$\varphi \sqsubseteq \{fork\ \varphi\}$
[E-DET]	$\dfrac{\varphi_1 \sqsubseteq \varphi_2}{\varphi_1.\bot \sqsubseteq \{det\ \varphi_2\}.\theta}$	[F-FOR]	$\dfrac{\varphi_1 \sqsubseteq \varphi_2}{\{fork\ \varphi_1\} \sqsubseteq \{fork\ \varphi_2\}}$
[F-EMP]	$\emptyset \sqsubseteq \varphi$	[F-DET]	$\dfrac{\varphi_1 \sqsubseteq \varphi_2}{\{det\ \varphi_1\} \sqsubseteq \{det\ \varphi_2\}}$
[F-R/R]	$\dfrac{\rho_1 \subseteq \rho_2}{\{!\ \rho_1\} \sqsubseteq \{!\ \rho_2\}}$	[F-UNI]	$\dfrac{\varphi_1 \sqsubseteq \varphi_1' \qquad \varphi_2 \sqsubseteq \varphi_2'}{\varphi_1 \cup \varphi_2 \sqsubseteq \varphi_1' \cup \varphi_2'}$

Table 4. Subtype and Subeffect Rules

The rules [APPLICATION], [REFERENCE], [DEREFERENCE], [UPDATE] and [FORK] inherit and sequentially compose deterministic effects from their subexpressions. For example, the overall deterministic effect of [APPLICATION] is the sequential composition of the deterministic effects of its function e_1 and argument e_2 evaluations, followed by the deterministic effect of the function body itself extracted from the function type. The composition of $\Delta_1;\ \Delta_2;\ \Delta_3 = \Delta$ is a shorthand for the composition of $\Delta_1;\ \Delta_2 = \Delta'$ and $\Delta';\ \Delta_3 = \Delta$. Sequential composition of effects is defined in [COMPOSITION] of Table 2 and requires sequential composability of the component effects. This is where the test for noninterference of concurrent effects arises in our system. In [REFERENCE], the labelled creation point π is recorded in the type of the introduced reference, so that later dereferences and assignments can track its use.

Only four rules are involved in base effect generation: [DEREFERENCE], [UPDATE], [FORK] and [DET]. The reference accessing expressions, dereference and update, compose the effect of the evaluation of their subexpressions, with the side effects generated by memory access (indicated by the read $!\rho$ or write $\rho :=$ in the rules). The effect of **fork** e is the effect of e qualified with the effect keyword *fork*. As discussed already, this distinguishes effects of concurrent threads from effects of the current thread; no other form of expression generates a fork effect. The rule [DET] asserts that an expression **det** e is strongly deterministic, provided that e is weakly deterministic and its inferred base effect *det* φ ensures the expression cannot be interfered with by concurrent threads.

The type rules rely on effect composition, which in turn, as we shall see below, relies on a test of sequential composability which is asymmetric in its first and second effects. Consequently our type system is flow sensitive. In simple memory effect systems, the effect of $e_1; e_2$ and of $e_2; e_1$ are the same—simply the union of the two effects. For us, it is possible that $e_1; e_2$ is legal, whereas $e_2; e_1$ is not, as illustrated by the two versions of the first example in Section 2.1.

Sequential composition of deterministic effects, defined in Table 2, and the associated notion of noninterference, in Table 3, lie at the heart of the type system. We define the ordering $\bot \leq \top$, meaning that a deterministic expression can be considered as a nondeterministic expression. To reduce the number of rules, the symbol for sequential composition is parameterised with determinism: $\Delta_1 \hookrightarrow_\theta \Delta_2$ where θ could be either $\bot$ or $\top$. Now we define both *deterministic* and *nondeterministic compositions* in the [COMPOSITION] rule. If the composition is deterministic (i.e. the resulting effect is the union of effects $\varphi_1 \cup \varphi_2$ and $\bot$), then both expressions must be at least weakly deterministic and the $\hookrightarrow_\bot$ symbol is used to denote *strong sequential composability*. Otherwise, the $\hookrightarrow_\top$ symbol is used to denote *weak sequential composability* which allows interference between φ_1 and φ_2 and generates a nondeterministic combined effect $(\varphi_1 \cup \varphi_2).\top$.

The key role of [COMPOSITION] is to check that deterministic effects (thus expressions) are indeed sequentially composable. The rest of rules in Table 2 define strong and weak sequential composability as a binary relation over base effects φ. [C-EMP], [C-EFF], [C-UNI] and [C-DUN] are generic rules for both strong and weak sequential composability. By rule [C-EFF] we assert that side effects $\{\varepsilon\}$ associated solely with the current thread, are sequentially composable with any subsequent effect. This captures the idea that terminated computations cannot interfere with later ones. [C-UNI] allows the combination of effects on the left provided that each of them is sequentially composable with the effect on the right. The second union rule [C-DUN] preserves the *det* qualification over composition. Strong sequencing rules forbid thread interference between two effects; therefore it is safe to remove the *det* qualification in the strong sequencing rule [C-DET-S]. In another strong sequencing rule [C-FOK-S], composing a fork effect $\{\mathit{fork}\ \varphi_1\}$ with another φ_2 requires the effect φ_1 not to interfere ($\nrightarrow_\bot$) with the overall effect of φ_2 and vice versa, because the thread may be running concurrently with the second expression. On the other hand, the weak sequencing rule [C-FOK-W] allows interference between threads by using $\nrightarrow_\top$ (weak noninterference). [C-DET-W] is special, as it allows a deterministic concurrent effect $\{\mathit{det}\ \{\mathit{fork}\ \varphi_1\}\}$ to be composable with its following effect φ_2 only if φ_2 does not interfere with φ_1; the last pair of examples in Section 2 illustrate the use of this rule.

Table 3 defines strong and weak *noninterference* of concurrent effects; noninterference is asymmetric. We consider strong noninterference $\varphi_1 \nrightarrow_\bot \varphi_2$ to mean that the effect φ_1 *does not affect* φ_2. So reading shared memory does not affect any writes to that memory, whereas a write can affect a read. [N-R/R] states that a read effect interferes with nothing; [N-W/R] and [N-W/W] state that a write effect does not interfere with other concurrent side-effect (read or write) only if they access distinct regions. Weak noninterference rules, on the other hand, are treated in a way that resembles weak sequencing; they are designed to allow arbitrary interference amongst expressions, except for deterministic expressions. [N-EFF] asserts that side effects do not weakly interfere with one another, implying they may occur in parallel. The remaining rules are generic for either strong or weak noninterference. The difference between [N-DEL] and [N-DER] is important. [N-DEL] asserts that a qualified *det* effect does not interfere with another

$$\text{[R-APP]}\quad \langle \varsigma,\ (\textbf{fn}\ x => e)\ v\rangle \xrightarrow{\emptyset} \langle \varsigma,\ [v/x]e\rangle$$

$$\text{[R-REF]}\quad \frac{\iota \notin \mathit{dom}(\varsigma) \qquad \varsigma' = \varsigma, \iota^{\pi} \mapsto v}{\langle \varsigma,\ \textbf{ref}_{\pi}\ v\rangle \xrightarrow{\emptyset} \langle \varsigma',\ \iota\rangle}$$

$$\text{[R-DRF]}\quad \langle \varsigma,\ !\iota\rangle \xrightarrow{\{!\{\iota\}\}} \langle \varsigma,\ \varsigma(\iota)\rangle$$

$$\text{[R-UPD]}\quad \langle \varsigma,\ \iota := v\rangle \xrightarrow{\{\{\iota\}:=\}} \langle \varsigma[\iota \mapsto v],\ v\rangle$$

$$\text{[R-FOK]}\quad \langle \varsigma,\ \textbf{fork}\ e\rangle \xrightarrow{\emptyset} \langle \varsigma,\ \widehat{e}\ 0\rangle$$

$$\text{[R-SEQ]}\quad \langle \varsigma,\ \widehat{v}\ e\rangle \xrightarrow{\emptyset} \langle \varsigma,\ e\rangle$$

$$\text{[R-CON]}\quad \langle \varsigma,\ E[\widehat{e}\ v]\rangle \xrightarrow{\emptyset} \langle \varsigma,\ \widehat{e}\ E[v]\rangle$$

$$\text{[R-DET]}\quad \frac{\langle \varsigma,\ e\rangle \xrightarrow{\varphi} \langle \varsigma,\ e'\rangle}{\langle \varsigma,\ \textbf{det}\ e\rangle \xrightarrow{\{\mathit{det}\ \varphi\}} \langle \varsigma,\ \textbf{det}\ e'\rangle}$$

$$\text{[R-DER]}\quad \langle \varsigma,\ \textbf{det}\ v\rangle \xrightarrow{\emptyset} \langle \varsigma,\ v\rangle$$

$$\text{[R-CTX]}\quad \frac{\langle \varsigma,\ e\rangle \xrightarrow{\varphi} \langle \varsigma',\ e'\rangle}{\langle \varsigma,\ E[e]\rangle \xrightarrow{\varphi} \langle \varsigma',\ E[e']\rangle}$$

Table 5. Operational Semantics

effect, providing the underlying effect does not. However, [N-DER] asserts that an effect does not (either strongly or weakly) interfere with a *det* effect only if it does not *strongly* interfere with it; this is essentially how strong determinism is enforced by using the *det* qualification. [N-FOL] and [N-FOR] state that a fork effect does not interfere with another effect provided their components do not and vice versa. [N-UNL] and [N-UNR] state that a union of effects does not interfere with another effect if its component do not and vice versa; [N-EMP] simply states that the empty effect interferes with no effect.

We complete this section with subtyping and subeffecting rules in Table 4. Subtyping is reflexive. As usual, function types are contravariant in arguments and covariant in results. Both function types and reference types allow broadening of effects or regions in moving to a supertype. [E-SUB] defines subeffecting for deterministic effects Δ. A deterministic expression can be considered as a nondeterministic expression. The [E-DET] rule is a special case, which states φ_1 cannot be a subeffect of $\{\mathit{det}\ \varphi_2\}$ unless it is deterministic (as suggested by $\perp$). The [F-] rules define subeffecting for base effects φ; those defining subeffecting for side effects are standard. [F-FOK] states that a fork effect is a supereffect of its component; it loses any information about effects being on the same thread. [F-FOR] states that a fork effect is monotonic on subeffecting, as does [F-DET] for *det*. For our purposes, the key property for subeffecting is that an effect inherits any noninterference enjoyed by a supereffect.

3.2 Dynamic Semantics and Properties

We define the dynamic behaviour of the core calculus, and demonstrate that the actual effects arising from the evaluation of a well-typed expression are consistent with its statically inferred effects. The syntax is extended to include features required for the dynamic semantics, as follows:

$$e \ ::= \ ... \mid \widehat{e}\ e \qquad\qquad v \ ::= \ ... \mid \iota^{\pi} \qquad\qquad \rho \ ::= \ ... \mid \{\iota\}$$

Unlike the more standard "list of threads" technique that does not preserve the relative nesting of thread creations [9], our extended syntax for runtime

[STORE] $$\frac{\mathit{dom}(\mathit{\Sigma}) = \mathit{dom}(\varsigma) \qquad \forall \iota \in \mathit{dom}(\varsigma) \;\cdot\; \mathit{\Gamma}; \mathit{\Sigma} \vdash \varsigma(\iota) : \mathit{\Sigma}(\iota) \;\&\; \mathit{\Delta}}{\mathit{\Gamma}; \mathit{\Sigma} \vdash \varsigma}$$

[PARALLEL] $$\frac{\mathit{\Gamma}; \mathit{\Sigma} \vdash e_1 : \tau_1 \;\&\; \varphi_1.\theta_1 \quad \mathit{\Gamma}; \mathit{\Sigma} \vdash e_2 : \tau_2 \;\&\; \mathit{\Delta}_2 \quad \{\mathit{fork}\ \varphi_1\}.\theta_1 \;; \mathit{\Delta}_2 = \mathit{\Delta}}{\mathit{\Gamma}; \mathit{\Sigma} \vdash \widehat{e_1}\ e_2 : \tau_2 \;\&\; \mathit{\Delta}}$$

[REG-IN] $\iota^{\pi} \in \{\pi\}$ [LOCATION] $$\frac{\mathit{\Sigma}(\iota) = \tau}{\mathit{\Gamma}; \mathit{\Sigma} \vdash \iota : \mathit{ref}_{\{\iota\}}\ \tau \;\&\; \emptyset.\bot}$$

[REG-OUT] $$\frac{\pi \neq \varpi}{\iota^{\pi} \notin \{\varpi\}}$$ [EQ-IDE] $$\frac{\mathit{for\ all}\ \iota\ \mathit{appears\ in}\ e \;\cdot\; \varsigma(\iota) = \varsigma'(\iota)}{\langle \varsigma, e\rangle \cong \langle \varsigma', e\rangle}$$

[EQ-LOC] $$\frac{\begin{array}{c}\langle \varsigma, \varsigma(\iota)\rangle \cong \langle \varsigma', \varsigma'(\iota')\rangle \\ [\iota''/\iota]\langle \varsigma, e\rangle \cong [\iota''/\iota']\langle \varsigma', e'\rangle \qquad \iota''\ \mathit{does\ not\ appear\ in}\ \varsigma,\ \varsigma',\ e\ \mathit{or}\ e'\end{array}}{\langle \varsigma, e\rangle \cong \langle \varsigma', e'\rangle}$$

[EQ-PAR] $$\frac{\begin{array}{c}\forall i \in 1..n \;\cdot\; \langle \varsigma, e_i\rangle \cong \langle \varsigma', e_{\sigma(i)}\rangle \qquad \mathit{where}\ \sigma\ \mathit{is\ a\ permutation\ of}\ 1..n \\ \langle \varsigma, e\rangle \cong \langle \varsigma', e'\rangle\end{array}}{\langle \varsigma, \widehat{e_1}\ ..\ \widehat{e_n}\ e\rangle \cong \langle \varsigma', \widehat{e_{\sigma(1)}}\ ..\ \widehat{e_{\sigma(n)}}\ e'\rangle}$$

Table 6. Auxiliary Definitions

expressions introduces a novel parallel construct $\widehat{e}\ e$ to allow us to represent an expression together with its threading context, that is, the threads it has forked. This expression is right-associative, the sequential continuation e, after forking n threads $e_1 \dots e_n$, is simply written as $\widehat{e_1}\ (\widehat{e_2} \dots (\widehat{e_n}\ e))$ or just $\widehat{e_1}\ \widehat{e_2} \dots \widehat{e_n}\ e$. This expression records the tree structure of forked threads. For example $\widehat{e_1}\ e_2$ may evaluate to $\widehat{\widehat{e_3}\ e_4}\ e_2$ if e_1 forks a thread e_3 and continues with e_4; again it may further evaluate to $\widehat{\widehat{e_3}\ e_4}\ \widehat{e_5}\ e_6$ if e_2 forks a thread e_5 and continues with e_6.

Values now include runtime store locations ι; these are annotated with the label π corresponding to the $\mathbf{ref}_{\pi}$ expression from where the location was allocated. For succinctness we omit the labels wherever they are not explicitly required. Runtime regions are sets of locations. Runtime effects correspond to accessing the store using locations; a runtime effect is a set of side effects on locations. The conventional form of evaluation contexts is used to define the order of evaluation of subexpressions in compound terms. Except for parallel expressions $\widehat{e}\ e$, the evaluation context is deterministic in its selection of subexpression.

$$E \;::=\; [\,] \mid E\ e \mid (\mathbf{fn}\ x \Rightarrow e)\ E \mid \mathbf{ref}_{\pi}\ E \mid\ !E \mid E := e \mid \iota := E \mid \widehat{E}\ e \;\mid\; \widehat{e}\ E$$

The small step operational semantics in Table 5 uses this form of transition:

$$\langle \varsigma,\ e\rangle \xrightarrow{\varphi} \langle \varsigma',\ e'\rangle$$

where ς is the global store, e is the expression to be evaluated. A store ς maps locations to the values stored in them, and is initially empty at the beginning of evaluation. The store may be extended by [R-REF]; store extension is written as $\varsigma, \iota \mapsto v$. The effect of the evaluation φ identifies the effect of the small step transition, which is either empty, $\emptyset$, or a singleton side effect, $\{\varepsilon\}$.

All possible single step transitions between configurations are given, where the initial state for a top-level expression e has an empty store. Like conventional operational semantics for concurrent programs [9, 1], each single step transition is atomic and thread interleaving is modelled by choice of step. The only non-determinism (modulo choice of new locations and structure on sequential terms; such equivalence is formally defined by the [EQ-] rules in Table 6) offered by these evaluation contexts arises from the two choices corresponding to the parallel construct $\widehat{e}\ e$, as either the concurrent part or the sequential part may undergo transitions, which is implicit in the rule [R-CTX], thus capturing thread interleaving.

The rules [R-APP] through [R-UPD] build up base cases on evaluation in a single thread. The [R-REF] transition captures the introduction of a new location into the store, with label annotations corresponding to the **ref** construct. The only transitions which directly have an effect are those for reads and writes of the store, namely [R-DRF] and [R-UPD]. New threads are introduced via [R-FOK]; it causes no direct effect but introduces a new thread ($\widehat{e}$) whose effect corresponds to the fork effect tracked in the static type system. The value that results from the **fork** step is arbitrarily chosen to be 0. If a concurrent thread $\widehat{e}$ reduces to a value $\widehat{v}$, there are no further reductions available for it. Such threads are effectively garbage and are easily eliminated with [R-SEQ]. Similarly, [R-DER] simply removes the keyword **det** from an expression when it finishes its evaluation. [R-DET] preserves **det** on expressions so that subsequent evaluation must also be deterministic; this is necessary in proving our determinism result, though it does not affect the evaluation. In [R-CON] a forked thread $\widehat{e}$ is relocated outside of its parent expression; this allows the return value to be used by the parent expression (e.g. a fork in an assignment needs to return a value for the assignment to progress). It preserves the nesting of thread creations in our syntax.

To help formalise and prove the safety properties, we use standard store typing for reference types [20] by extending the type judgements with an additional store typing Σ, which maps locations to their types; store typing extension is written as $\Sigma, \iota \mapsto \tau$. All expression typing judgements will have the form:

$$\Gamma; \Sigma \vdash e : \tau \ \&\ \varphi$$

The only rule that needs to use Σ is [LOCATION]. We do not rewrite other expression typing rules in Table 1, because they do not need to do anything interesting with the store typing—just pass it from premise to conclusion.

Table 6 provides auxiliary definitions used by the operational semantics and the theorems. The type rule for our intermediate form for concurrent expressions $\widehat{e_1}\ \ e_2$, is given by [PARALLEL]. This records the fact that concurrent threads have fork effects, and that their combined effect is given by the union of their effects. The standard definition [STORE] asserts that a store is well-typed if the value stored in every location has the type predicted by the store typing. In [STORE], we also capture the property that locations created at different program points must be different. We connect labels used in static semantics with the locations in dynamic semantics in [REG-IN] and [REG-OUT]. The [EQ-] rules define an equivalence on state: [EQ-IDE] is the induction base which allows us to restrict attention to just that part of the store which affects the value of the expression; [EQ-LOC] does location substitution which allows us to treat two locations holding the same value as being identical; [EQ-PAR] does shuffling which allows reordering of forked threads. For example, ι is regarded the same as ι' if they contain the same value, and $\widehat{e_1}\ \widehat{e_2}\ e_3$ is regarded the same as $\widehat{e_2}\ \widehat{e_1}\ e_3$, because the order of forked threads is not important.

We extend the transitions to multi-step, which have the form

$$\langle \varsigma,\ e\rangle \overset{\varphi}{\Longrightarrow} \langle \varsigma',\ e'\rangle$$

where φ is the union of effects of finitely many steps. We prove that for a terminating program, well-typedness implies determinism in its final value. The main result is a strong determinism theorem which states:

1. the evaluation of any deterministic sub-term (**det** e) cannot be affected by reductions in the external context.
2. without interleaving with its context, the result of the evaluation of **det** e is unique, independently of how the threads inside e interleave.

Theorem 1 (Strong determinism)
Given all reachable states $\langle \varsigma, E[\mathbf{det}\ e]\rangle$ *such that* $\Gamma;\Sigma \vdash E[\mathbf{det}\ e] : \tau\ \&\ \Delta$ *and* $\Gamma;\Sigma \vdash \varsigma$*, if* $\langle \varsigma,\ \mathbf{det}\ e\rangle \overset{\varphi_1}{\Longrightarrow} \langle \varsigma_1,\ v\rangle$ *and* $\langle \varsigma,\ E[\mathbf{det}\ e]\rangle \overset{\varphi_2}{\Longrightarrow} \langle \varsigma_2,\ E'[\mathbf{det}\ e]\rangle$*, then*

1. *there exists* $\langle \varsigma_2, E'[\mathbf{det}\ e]\rangle \overset{\varphi_3}{\Longrightarrow} \langle \varsigma_3, E'[v']\rangle$ *and* $\langle \varsigma_1, v\rangle \cong \langle \varsigma_3, v'\rangle$.
2. *for all* $\langle \varsigma,\ \mathbf{det}\ e\rangle \overset{\varphi_4}{\Longrightarrow} \langle \varsigma_4,\ v''\rangle$*, we have* $\langle \varsigma_1, v\rangle \cong \langle \varsigma_4, v''\rangle$.

4 Discussion

4.1 Extensions

This paper has presented a core calculus and effect system, based on the approach of Nielson et al., to allow a focus on the novel features of our approach and formal results. In this section, we review a number of existing techniques for improving precision and expressiveness of effect systems, and how they can be extended to our deterministic effects.

Thread-locality Thread-local references cannot be aliased by other threads, thus effectively reducing interference between threads. Type systems can be used to restrict thread-local references. For example, the lexically scoped reference construct $\mathbf{new}_\pi\ x := e_1\ \mathbf{in}\ e_2$ in [18, 19] creates a new reference variable x for use in e_2 and initialises it to the value of e_1. With such a construct we can confine the reference within its scope e_2 (incidentally, [18, 19] do not explicitly impose such restrictions). Consider the following example adopted from [9]:

```
let  f = (fn x => new_π y := 0 in y := x)  in
let  rec = ref_ϖ (fn x => x)  in
   rec := fn x => (fork f 0; !rec 0);    !rec 0
```

This example captures the essence of a server, which creates a thread to handle each incoming request. The core of this example is a recursive function that creates a new thread to allocate a reference cell y and use y in handling an incoming request (we simply use an assignment to represent the handling, and incoming requests are not modelled), and finally calls itself recursively to handle the next incoming request. We confine the references created at π within its lexical scope so that request handling threads have their own memory space isolated from one another. The effect of $!rec\ 0$ is $\{fork\ \{\{\pi\} :=\}, !\{\varpi\}\}$. Since references created at π are always thread-local, the visible effect of $!rec\ 0$ is $\{!\{\varpi\}\}$. There is no interference present in this example.

More expressive type systems such as uniqueness [10, 7] or ownership [6, 3, 14] may be extended with our effect system to confine thread-local references in a more flexible way. Besides type systems, escape analysis [23] and thread sharing analysis [17] identify thread-local locations based on program analysis, and are often used in data race detection and other static analysis tools.

Structured Parallelism Previous type systems for deterministic parallelism [1, 2] support structured parallel programs, which may be considered as a special case where lifetimes of threads are restricted to lexical scopes. It is easy to support such a fork/join construct, **forkjoin** $e_1\ e_2$, in our deterministic effect system, by a typing rule which simply checks if two tasks may interfere:

$$\text{[FORKJOIN]}\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1\ \&\ \varphi_1.\theta_1 \qquad \Gamma \vdash e_2 : \tau_2\ \&\ \varphi_2.\theta_2 \\ \varphi_1 \not\rightharpoonup_{\perp} \varphi_2 \qquad \varphi_2 \not\rightharpoonup_{\perp} \varphi_1 \quad \theta_1 \leq \theta_3 \qquad \theta_2 \leq \theta_3\end{array}}{\Gamma \vdash \textbf{forkjoin}\ e_1\ e_2 : \tau_2\ \&\ (\varphi_1 \cup \varphi_2).\theta_3}$$

For example, here is a Fibonacci function encoded in our calculus using the **forkjoin** expression (with conditionals and basic arithmetic):

```
let  fib = ref (fn n => n)  in
  fib := fn n => ( if (n < 2) n
                   if (n >= 2) ( new_π x := 0  in  new_ϖ y := 0  in
                     forkjoin (x := fib (n − 1)) (y := fib (n − 2));  x + y )
  );   !fib 10
```

Because task lifetime is constrained to the **forkjoin** construct, it does not introduce fork effects, unlike **fork**. The rule simply requires that the two parallel expressions do not interfere with one another (in this example, it is easy to see their effects are disjoint). Unlike [1, 2], we can mix structured parallelism with threads; for example, the **forkjoin** expression may fork threads too. This generality does not compromise safety, because thread effects will be captured in the resulting effect of the **forkjoin** expression.

Polymorphism Hindley/Milner polymorphism, as found in Standard ML and other functional languages, is a classical technique for increasing the precision of types and effects. It allows us to distinguish the effects of two applications of the same function. Let us consider the following example:

```
let  count1 = ref_π 0  in  let  count2 = ref_ϖ 0  in
let  inc = (fn x => x := !x + 1)  in
  fork (inc count1);   fork (inc count2)
```

According to the type rules presented in Section 3, the type of the function variable x of *inc* is $\mathit{ref}_{\{\pi,\varpi\}}$ *int*, so that the effect of *inc* is $\{!\{\pi,\varpi\},\{\pi,\varpi\} :=\}$. This example cannot type check because the threads have the same effect $\{\mathit{fork}\ \{!\{\pi,\varpi\},\{\pi,\varpi\} :=\}\}$, which includes a write side-effect, so the threads are judged to interfere with one another. Of course we know that these threads access different reference cells and hence do not actually interfere at runtime.

The treatment of polymorphism for our type and effect is standard and has been given in [18, 19], which allows us to achieve the extra precision required to type check this example. Using polymorphism in the previous example, the type of the variable x is $\forall\zeta.\mathit{ref}_{\zeta}$ *int* (where ζ is a region variable) and the effect of *inc* becomes $\forall\zeta.\{!\zeta,\zeta :=\}$. After instantiation of the region variable, the effects of the two threads can be distinguished as $\{\mathit{fork}\ \{!\{\pi\},\{\pi\} :=\}\}$ and $\{\mathit{fork}\ \{!\{\varpi\},\{\varpi\} :=\}\}$ respectively, which clearly do not interfere. Polymorphism also makes type checking modular, which is useful for checking incomplete programs. For example, without polymorphism, the type of *inc* may depend on how it is used in the last two expressions. With polymorphism, the type of *inc* is independent of its use so the function may be defined in a different module.

Effect Abstraction Nielson et al.'s effect systems use simple abstraction to model shared locations—effectively the label of the program point at which the reference is created. Although our effect system extends their framework, the general approach presented in this paper is largely independent of the specific abstraction chosen. However in practice, stronger effect abstractions and specifications are needed for precision and modularity.

The precision of effect reasoning relies on aliasing reasoning, which is one of the main sources of imprecision in type and effect systems (in fact, any static analysis). Because any sound type system must make conservative choices about aliasing (for example, when two references may be aliases, we must conservatively consider them as aliases), some good programs may not type check or may signal false warnings. It is particularly difficult to distinguish references created from the same program point. Ownership [6, 3, 15] and region-based [16, 1] effect systems parameterise object types with owners or regions to enrich the type structure. For instance, elements in a data structure may be distinguished if they have different types (parameterised with different owners/regions). DPJ [1] also suggests a special treatment for arrays, which relies on distinguishing element types by their array indices. Moreover, these effect systems support explicit effect specifications, which enhance program reasoning, enable separate compilation and facilitate modular software development. Our approach in this paper naturally supports modularity, as the effect of an expression can be inferred from only itself (i.e. independent to the context/environment, see more discussion in Section 4.2). Ownership or region-based effects may be extended in our framework to replace the simple label-based effects; we only need to adapt some noninterference rules for supporting these new forms of effects.

4.2 Related Work

Much work has addressed the challenges of shared-memory concurrent programming. In this section we restrict attention to directly related work, including determinism and effects systems. Traditionally, determinism can be guaranteed for some restricted styles of parallel programming, such as data parallel or pure functional. Recent times have seen increased support for deterministic parallelism in imperative programs. Examples include the use of dynamic analysis [22, 5], type systems [25, 1, 2], static analysis [26] or separation logic [8].

Dynamic analysis allows programmers to assert desired deterministic sections and enforce determinism by runtime checks. Dynamic approaches are generally more flexible (e.g. [5] supports *semantic determinism* to tolerate some controlled nondeterminism inside a deterministic section) and precise (e.g. less or no false positives), but they often impose considerable runtime overhead and have limited test coverage. Type systems, on the other hand, enforce determinism statically at compile time; they typically capture errors earlier with no runtime cost, but may report more false warnings. Type systems require annotations, which may increase programming effort but are useful for program specification and documentation especially for modular development of large software.

Our type system is modelled on those of Nielson et al. [18, 19]. In their systems they have provided separate effects for spawned expressions (like fork for us); their systems also track sequencing in effects. The key difference for our effects are that we explicitly prohibit unwanted concurrent behaviours; we only allow an effect sequence $\Delta_1; \Delta_2$ to be formed when the effects are sequentially composable. Our use of deterministic effects appears to be novel. Unlike related work on determinism for other concurrency models [1, 2], as befits a threading model, our approach focuses on the effect of the current thread separately from the effect of the threads that it has forked. These approaches treat the components of a parallel expression symmetrically, and are flow-insensitive.

Type and effect systems for tracking noninterference in programs are useful for facilitating program reasoning and verification in sequential programs [6], for analysing behavior of concurrent processes in process algebras [11], and for enforcing determinism in structured parallel programs [1]. Fractional capabilities [4, 25] provide similar support by treating a read capability as a fraction of the write capability and distributing capabilities on memory locations at synchronisation points to ensure that each thread must have sufficient capability to read/write these locations. They can support determinism and synchronisation but inherently lack modularity, because in order to type a function, for instance, the capability of the calling context (thread) must be known. On the other hand, deterministic effects may be described independently of the context in which they are used (flow-sensitivity is captured locally by sequential composability), thus making modular reasoning about incomplete programs (e.g. a function or a class) feasible. Furthermore, our approach can support a variety of computational effects (not just reads/writes) and check for different interference properties. For example, it may reason about deadlocks by capturing (ordered) lock sets as effects and the inconsistency in lock ordering as interference.

4.3 Future Work

With this work, we provide a sound and general framework which can be used as a basis for studying more kinds of computational effects and interference in multithreaded programs. Two interesting directions would be: higher level mechanisms for expressing effects, and incorporating synchronisation.

Understanding and writing multithreaded code is difficult, partly because of the lack of specification for concurrent behaviours. Programmers typically work with large libraries of code whose concurrent behaviours (e.g. threading, synchronisation, locks, etc.) are not precisely specified. Our framework may be extended with existing object-oriented effect systems (e.g. ownership or regions) to allow programmers to express their high-level design intentions via effect contracts on methods. Moreover, the ability to statically determine if two parallel computations may interfere is critical in the design of concurrent software, for instance, the degree of concurrency can be increased by reducing thread interference and removing redundant synchronisation.

In this paper, we have not been concerned with synchronisation/locking which adds little to the novelty of our model (deterministic and nondeterministic effects) and does not affect the results (strong and weak determinism). Instead we have aimed to present a foundation of a simple and general formalism for reasoning about multithreaded programming upon which we can build more elaborate models, including for synchronisation. For example, by capturing lock sets [9, 21] as effects, our framework can be used to reason about data races [27] and deadlocks, which may be characterised as two kinds of thread interference. Type-based techniques for imposing locking disciplines can detect data races [9, 3] or deadlocks [3] by requiring a shared location to be consistently guarded by a common lock, or locks to be acquired in a fixed order; typically they, pessimistically, assume maximal concurrency. With our thread-sensitive approach, it is possible to improve the precision of reasoning about concurrency vulnerabilities.

5 Conclusion

Threads are the dominant model in use today for general concurrent programming, but they are wildly nondeterministic and notoriously difficult to understand and predict, even for expert programmers. This paper proposes a novel approach for analysing thread interference and determinism in multithreaded programs, and presents a simple type and effect system to demonstrate our approach which can guarantee the preservation of desired deterministic behaviour. Deterministic effects may be used by tools or as interface specifications to assist with modular development of multithreaded software. We believe that deterministic effects are simple and easy enough to understand for average programmers, thus assisting them with difficult parts of multithreaded programming.

Acknowledgements

This research is supported by an Australian Research Council Grant DP0987236.

References

1. R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.
2. R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.
3. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.
4. J. Boyland. Checking interference with fractional permissions. In *SAS*, 2003.
5. J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *FSE*, 2009.
6. D. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *OOPSLA*, 2002.
7. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.
8. M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *POPL*, 2011.
9. C. Flanagan and M. Abadi. Types for safe locking. In *ESOP*, 1999.
10. J. Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA*, 1991.
11. N. Kobayashi. Type systems for concurrent programs. In *10th Anniversary Colloquium of UNU/IIST*, 2002.
12. D. Lea. A Java fork/join framework. In *Java Grande*, 2000.
13. E. A. Lee. The problem with threads. *IEEE Computer*, 39(5), 2006.
14. Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP*, 2006.
15. Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *POPL*, 2006.
16. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988.
17. I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, 2008.
18. F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*, 1999.
19. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
20. B. Pierce. *Types and Programming Languages.* The MIT Press, 2002.
21. P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*, 2006.
22. C. Sadowski, S. N. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *ESOP*, 2009.
23. A. Salcianu and M. C. Rinard. Pointer and escape analysis for multithreaded programs. In *PPOPP*, 2001.
24. H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7), 2005.
25. T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *TOPLAS*, 30(5), 2008.
26. M. T. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic verification of determinism for structured parallel programs. In *SAS*, 2010.
27. X. Xie and J. Xue. Acculock: Accurate and efficient detection of data races. In *CGO*, 2011.