Toward a Grainless Semantics for Shared-Variable Concurrency [⋆]

John C. Reynolds

Carnegie Mellon University
and
Edinburgh University
john.reynolds@cs.cmu.edu

Abstract. Conventional semantics for shared-variable concurrency suffers from the "grain of time" problem, i.e., the necessity of specifying a default level of atomicity. We propose a semantics that avoids any such choice by regarding all interference that is not controlled by explicit critical regions as catastrophic. It is based on three principles:

- Operations have duration and can overlap one another during execution.
- If two overlapping operations touch the same location, the meaning of the program execution is "wrong".
- If, from a given starting state, execution of a program can give "wrong", then no other possibilities need be considered.

1 Introduction

Ever since the early 1970's, when researchers began to propose programming languages in which concurrent processes interact via shared variables, the problem of default atomicity, which Dijkstra called the "grain of time" phenomenon, has plagued the design and definition of such languages. Basically, if two concurrent processes access the same variable, without any explicit description of atomicity or mutual exclusion, the variety of outcomes will depend on a default choice of the level of atomicity, increasing as the atomicity becomes more fine-grained.

For example, consider the concurrent execution of two assignments to the same variable:

$$\mathsf{x} := \mathsf{x} \times \mathsf{x} \parallel \mathsf{x} := \mathsf{x} + 1 \ .$$

1. If each of these assignment commands is an atomic action, then there are two possible interleavings of the actions, which lead to two distinct possible outcomes.

[⋆] Research was partially supported by National Science Foundation Grant CCR-0204242, by an EPSRC Visiting Fellowship at Edinburgh University, and by the Basic Research in Computer Science (http://www.brics.dk/) Centre of the Danish National Research Foundation. A more preliminary version of this material was presented at POPL 2004 [1].

2. If the evaluation of expressions and the storing of a value in a variable are atomic, then there are more interleavings and more possible outcomes.
3. If each load and store of a variable is atomic, there are still more interleavings and outcomes.
4. In the extreme case, say if x is a double-precision floating-point number, the atomic actions might be loads and stores of parts of the number representation, so that the possible outcomes would depend upon details of the machine representation of numbers.

In practice, at most the first of these interpretations would be useful to a programmer, while efficient implementation would be possible only at levels 3 or 4, where hardware-implemented mutual exclusion of memory references would suffice to guarantee atomicity.

The fact that there is no default level of atomicity that is natural for both user and implementor led researchers such as Hoare [2] and Brinch-Hansen [3] to propose that interfering concurrent commands such as $\mathsf{x} := \mathsf{x} \times \mathsf{x} \parallel \mathsf{x} := \mathsf{x} + 1$ should be prohibited syntactically, so that, whenever interference is possible, the programmer must indicate atomicity explicitly by means of critical regions, e.g.,

$$\textbf{with}\ \mathsf{lock}\ \textbf{do}\ \mathsf{x} := \mathsf{x} \times \mathsf{x} \parallel \textbf{with}\ \mathsf{lock}\ \textbf{do}\ \mathsf{x} := \mathsf{x} + 1\,.$$

Unfortunately, this proposal floundered when applied to languages that permit a flexible usage of pointers (which is why, for instance, the approach was not followed in ADA, even though it was mandated in the early requirements specifications of that language). For example, consider the concurrent composition of two indirect assignments:

$$[\mathsf{x}] := [\mathsf{x}] \times [\mathsf{x}] \parallel [\mathsf{y}] := [\mathsf{y}] + 1$$

(where $[\mathsf{x}]$ denotes the contents of the pointer that is the value of x). This command should be prohibited just when $\mathsf{x} = \mathsf{y}$ — but in general this kind of condition cannot be determined by a compiler.

Our answer to this dilemma is that, when $\mathsf{x} = \mathsf{y}$, the semantics of the above program is simply "**wrong**". To provide any further information would make no sense at any level of abstraction above the machine-language implementation, and would unnecessarily restrict the ways in which the program could be implemented.

More precisely, we propose a compositional semantics of shared-variable concurrency that avoids the "grain of time" phenomenon by employing three principles:

– All operations, except locking and unlocking, have duration, and can overlap one another during execution.
– If two overlapping operations lookup or set the same location, which may be either a variable or a pointer, then the meaning of program execution is **wrong**.
– If, from a given starting state, execution of a program can give **wrong**, then no other possibilities need be considered.

It must be emphasized that there is no intention of implementing **wrong** as a run-time error stop, which would be extremely inefficient and, in view of the nondeterminacy of concurrent computation, of little use. Instead, when the semantics of a program execution is **wrong**, there are no constraints on how it may be implemented.

Thus it is the programmer's obligation to make a convincing argument that his program will not go wrong. We can define what that means, and we should be able to provide a logic in which such arguments can be made rigorous. But the development of programming languages over the past thirty years makes it clear that this concept of wrongness cannot be decided automatically without restricting the programming language in ways that are unacceptable for many applications.

Our hope of providing an appropriate logic lies in the development of separation logic [4], and, more particularly, in its extension to shared-variable concurrency [5]. The soundness of this extension is very delicate, however, and thus must be demonstrated rigorously with respect to a compelling semantics of the concurrent programming language. Although we will not discuss separation logic further in this paper, we believe that the work described herein can provide such a semantics.

Recently, Steve Brookes proposed a novel semantics for shared-variable concurrency and used it to establish the soundness of separation logic [6]. There are considerable similarities between this work and ours: The starting point of both is a form of transition-trace semantics developed earlier by Brookes [7] (and based on still earlier ideas of Park [8]), in which traces are sequences of start-finish pairs of states. In Brookes's current semantics, these pairs are replaced by "actions", the actions of concurrent processes are interleaved, and uncontrolled interference is mirrored by interference between adjacent actions.

In contrast, our semantics captures duration directly by regarding start and finish as separate actions, between which the actions of other processes may intervene. (Although these two approaches are conceptually quite distinct, each has had significant influence on the other.)

2 Some Examples

The meaning of a command is a set of traces, each of which is a finite or infinite sequence of actions. Except for critical regions, the relevant actions are states labelled "**start**" or "**fin**".

For example, the meaning of the operation $\mathsf{x}{:=}\mathsf{x}{\times}\mathsf{x}$, which we write $[\![\mathsf{x}{:=}\mathsf{x}{\times}\mathsf{x}]\!]$ is the set of traces of the form

$$\mathbf{start}[\,\mathsf{x}\colon n\,]\,\mathbf{fin}[\,\mathsf{x}\colon n \times n\,]\,,$$

for all integers n. Notice that the states in the start and finish actions have the same domain, which is the *footprint* of the operation, i.e., the exact set of locations that are examined or set by the operation.

If $\mathsf{x} := \mathsf{x} \times \mathsf{x}$ runs by itself, without other processes running concurrently, its behavior is determined by *executing* the traces in its meaning. For example, starting in the state $[\,\mathsf{x}\colon 3 \mid \mathsf{t}\colon 22\,]$, the trace $\textbf{start}[\,\mathsf{x}\colon 3\,]\,\textbf{fin}[\,\mathsf{x}\colon 9\,]$ has the execution

$$\begin{array}{ll}
[\,\mathsf{x}\colon 3 \mid \mathsf{t}\colon 22\,] & \\
\quad\downarrow & \textbf{start}[\,\mathsf{x}\colon 3\,] \\
[\,\mathsf{t}\colon 22\,] & \\
\quad\downarrow & \textbf{fin}[\,\mathsf{x}\colon 9\,] \\
[\,\mathsf{x}\colon 9 \mid \mathsf{t}\colon 22\,]\,. &
\end{array}$$

In essence, the effect of the start action is to check that $\mathsf{x} = 3$ and then mark x "busy" by removing $[\,\mathsf{x}\colon 3\,]$ from the current state. Then the effect of the finish action is to return x to the current state with a new value.

On the other hand, when started in a state with a different value of x, the trace $\textbf{start}[\,\mathsf{x}\colon 3\,]\,\textbf{fin}[\,\mathsf{x}\colon 9\,]$ is irrelevant and has no execution. But an execution will be provided by another trace, such as $\textbf{start}[\,\mathsf{x}\colon 4\,]\,\textbf{fin}[\,\mathsf{x}\colon 16\,]$, in $[\![\mathsf{x} := \mathsf{x} \times \mathsf{x}]\!]$.

A third possibility arises when x does not occur in the domain of the starting state. In this case the execution goes wrong:

$$\begin{array}{ll}
[\,\mathsf{t}\colon 22\,] & \\
\quad\downarrow & \textbf{start}[\,\mathsf{x}\colon 3\,] \\
\textbf{wrong}\,. &
\end{array}$$

When a command assigns indirectly to a pointer, its footprint is more complex. For example, the meaning $[\![[\mathsf{x}] := [\mathsf{x}] \times [\mathsf{x}]]\!]$ is the set of traces

$$\textbf{start}[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2\,]\,\textbf{fin}[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2 \times n_2\,]\,,$$

for all integers n_1 and n_2. Here the footprint contains both x and its value n_1, which is a pointer. (We regard pointers as integers, in order to permit unrestricted pointer arithmetic.) Similarly, the meaning $[\![[\mathsf{y}] := [\mathsf{y}] + 1]\!]$ is the set of traces

$$\textbf{start}[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4\,]\,\textbf{fin}[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4 + 1\,]\,,$$

for all integers n_3 and n_4.

Now consider the concurrent execution of these two indirect assignments. The meaning

$$[\![[\mathsf{x}] := [\mathsf{x}] \times [\mathsf{x}] \parallel [\mathsf{y}] := [\mathsf{y}] + 1]\!]$$

is the set of interleavings of the traces of each subcommand. This set includes traces of the form

$$\begin{array}{l}
\textbf{start}[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2\,]\,\textbf{start}[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4\,] \\
\qquad \textbf{fin}[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4 + 1\,]\,\textbf{fin}[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2 \times n_2\,]\,,
\end{array}$$

in which the two assignment operations overlap.

When $n_1 \neq n_3$, such a trace executes without interference:

$$\begin{array}{ll}
[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2 \mid \mathsf{y}\colon n_3 \mid n_3\colon n_4\,] & \\
\quad\downarrow & \mathbf{start}[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2\,] \\
[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4\,] & \\
\quad\downarrow & \mathbf{start}[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4\,] \\
[\,] & \\
\quad\downarrow & \mathbf{fin}[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4 + 1\,] \\
[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4 + 1\,] & \\
\quad\downarrow & \mathbf{fin}[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2 \times n_2\,] \\
[\,\mathsf{y}\colon n_3 \mid n_3\colon n_4 + 1 \mid \mathsf{x}\colon n_1 \mid n_1\colon n_2 \times n_2\,]\,. &
\end{array}$$

On the other hand, when $n_1 = n_3$ and $n_2 = n_4$, the two assignments interfere:

$$\begin{array}{ll}
[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2 \mid \mathsf{y}\colon n_1\,] & \\
\quad\downarrow & \mathbf{start}[\,\mathsf{x}\colon n_1 \mid n_1\colon n_2\,] \\
[\,\mathsf{y}\colon n_1\,] & \\
\quad\downarrow & \mathbf{start}[\,\mathsf{y}\colon n_1 \mid n_1\colon n_2\,] \\
\mathbf{wrong}\,. &
\end{array}$$

Our treatment of critical regions follows the recent work of Brookes [6]. Three actions are involved, each of which names a *lock* or semaphore:

$\mathbf{try}(k)$:	Try to acquire k, but find it is already locked.
$\mathbf{acq}(k)$:	Succeed in acquiring k, and lock it.
$\mathbf{rel}(k)$:	Unlock k (or signal "impossible" if it is already unlocked).

For example, the meaning of the critical region **with** k **do** $\mathsf{x} := \mathsf{x} \times \mathsf{x}$ is the set of traces (for all integers n):

$$\begin{array}{l}
\mathbf{acq}(\mathsf{k})\ \mathbf{start}[\,\mathsf{x}\colon n\,]\ \mathbf{fin}[\,\mathsf{x}\colon n \times n\,]\ \mathbf{rel}(\mathsf{k}), \\
\mathbf{try}(\mathsf{k})\ \mathbf{acq}(\mathsf{k})\ \mathbf{start}[\,\mathsf{x}\colon n\,]\ \mathbf{fin}[\,\mathsf{x}\colon n \times n\,]\ \mathbf{rel}(\mathsf{k}), \\
\mathbf{try}(\mathsf{k})\ \mathbf{try}(\mathsf{k})\ \mathbf{acq}(\mathsf{k})\ \mathbf{start}[\,\mathsf{x}\colon n\,]\ \mathbf{fin}[\,\mathsf{x}\colon n \times n\,]\ \mathbf{rel}(\mathsf{k}), \\
\qquad\vdots \\
\mathbf{try}(\mathsf{k})\ \mathbf{try}(\mathsf{k})\ \mathbf{try}(\mathsf{k})\ \cdots\,.
\end{array}$$

To execute these new actions, we augment the current state of the computation with a set κ of "closed" locks. When $\mathsf{k} \notin \kappa$ (and n is 3), a trace of the first form has an execution:

$$\begin{array}{ll}
\kappa, [\,\mathsf{x}\colon 3 \mid \mathsf{t}\colon 22\,] & \\
\quad\downarrow & \mathbf{acq}(\mathsf{k}) \\
\kappa \cup \{\mathsf{k}\}, [\,\mathsf{x}\colon 3 \mid \mathsf{t}\colon 22\,] & \\
\quad\downarrow & \mathbf{start}[\,\mathsf{x}\colon 3\,] \\
\kappa \cup \{\mathsf{k}\}, [\,\mathsf{t}\colon 22\,] & \\
\quad\downarrow & \mathbf{fin}[\,\mathsf{x}\colon 9\,] \\
\kappa \cup \{\mathsf{k}\}, [\,\mathsf{x}\colon 9 \mid \mathsf{t}\colon 22\,] & \\
\quad\downarrow & \mathbf{rel}(\mathsf{k}) \\
\kappa, [\,\mathsf{x}\colon 9 \mid \mathsf{t}\colon 22\,]\,. &
\end{array}$$

On the other hand, when $\mathsf{k} \in \kappa$, the last trace has a nonterminating execution that represents deadlock:

$$\begin{array}{ll}
\kappa, [\,\mathsf{x}\colon 3 \mid \mathsf{t}\colon 22\,] & \\
\qquad\downarrow & \mathbf{try}(\mathsf{k}) \\
\kappa, [\,\mathsf{x}\colon 3 \mid \mathsf{t}\colon 22\,] & \\
\qquad\downarrow & \mathbf{try}(\mathsf{k}) \\
\qquad\vdots &
\end{array}$$

The remaining traces in $[\![\mathbf{with}\ \mathsf{k}\ \mathbf{do}\ \mathsf{x} := \mathsf{x} \times \mathsf{x}]\!]$ can only execute successfully after being interleaved with other traces that affect the same lock. For example, one possible interleaving of

$$\mathbf{try}(\mathsf{k})\,\mathbf{try}(\mathsf{k})\,\mathbf{acq}(\mathsf{k})\,\mathbf{start}[\,\mathsf{x}\colon 3\,]\,\mathbf{fin}[\,\mathsf{x}\colon 9\,]\,\mathbf{rel}(\mathsf{k})$$

with

$$\mathbf{acq}(\mathsf{k})\,\mathbf{start}[\,\mathsf{x}\colon 2\,]\,\mathbf{fin}[\,\mathsf{x}\colon 3\,]\,\mathbf{rel}(\mathsf{k})$$

(which is a trace in the meaning of $\mathbf{with}\ \mathsf{k}\ \mathbf{do}\ \mathsf{x} := \mathsf{x} + 1$) is

$$\begin{gathered}
\mathbf{acq}(\mathsf{k})\,\mathbf{start}[\,\mathsf{x}\colon 2\,]\,\mathbf{try}(\mathsf{k})\,\mathbf{fin}[\,\mathsf{x}\colon 3\,]\,\mathbf{try}(\mathsf{k}) \\
\mathbf{rel}(\mathsf{k})\,\mathbf{acq}(\mathsf{k})\,\mathbf{start}[\,\mathsf{x}\colon 3\,]\,\mathbf{fin}[\,\mathsf{x}\colon 9\,]\,\mathbf{rel}(\mathsf{k})\ ,
\end{gathered}$$

which executes as follows when $\mathsf{k} \notin \kappa$:

$$\begin{array}{llcll}
\kappa, [\,\mathsf{x}\colon 2\,] & & \qquad & \kappa \cup \{\mathsf{k}\}, [\,\mathsf{x}\colon 3\,] & \\
\qquad\downarrow & \mathbf{acq}(\mathsf{k}) & & \qquad\downarrow & \mathbf{rel}(\mathsf{k}) \\
\kappa \cup \{\mathsf{k}\}, [\,\mathsf{x}\colon 2\,] & & & \kappa, [\,\mathsf{x}\colon 3\,] & \\
\qquad\downarrow & \mathbf{start}[\,\mathsf{x}\colon 2\,] & & \qquad\downarrow & \mathbf{acq}(\mathsf{k}) \\
\kappa \cup \{\mathsf{k}\}, [\,] & & & \kappa \cup \{\mathsf{k}\}, [\,\mathsf{x}\colon 3\,] & \\
\qquad\downarrow & \mathbf{try}(\mathsf{k}) & & \qquad\downarrow & \mathbf{start}[\,\mathsf{x}\colon 3\,] \\
\kappa \cup \{\mathsf{k}\}, [\,] & & & \kappa \cup \{\mathsf{k}\}, [\,] & \\
\qquad\downarrow & \mathbf{fin}[\,\mathsf{x}\colon 3\,] & & \qquad\downarrow & \mathbf{fin}[\,\mathsf{x}\colon 9\,] \\
\kappa \cup \{\mathsf{k}\}, [\,\mathsf{x}\colon 3\,] & & & \kappa \cup \{\mathsf{k}\}, [\,\mathsf{x}\colon 9\,] & \\
\qquad\downarrow & \mathbf{try}(\mathsf{k}) & & \qquad\downarrow & \mathbf{rel}(\mathsf{k}) \\
\qquad\vdots & & & \kappa, [\,\mathsf{x}\colon 9\,]\ . &
\end{array}$$

3 Syntax, States, and the Semantics of Expressions

The programing language we will use throughout this paper is an extension of the simple imperative language:

$$\langle\text{exp}\rangle ::= \langle\text{var}\rangle \mid \langle\text{constant}\rangle \mid \langle\text{exp}\rangle + \langle\text{exp}\rangle \mid \cdots$$

$$\langle\text{boolexp}\rangle ::= \langle\text{exp}\rangle = \langle\text{exp}\rangle \mid \cdots \mid \langle\text{boolexp}\rangle \wedge \langle\text{boolexp}\rangle \mid \cdots$$

$$\begin{aligned}
\langle\text{comm}\rangle ::= {} & \langle\text{var}\rangle := \langle\text{exp}\rangle \mid \mathbf{skip} \mid \langle\text{comm}\rangle\ ;\ \langle\text{comm}\rangle \\
& \mid \mathbf{if}\ \langle\text{boolexp}\rangle\ \mathbf{then}\ \langle\text{comm}\rangle\ \mathbf{else}\ \langle\text{comm}\rangle \\
& \mid \mathbf{while}\ \langle\text{boolexp}\rangle\ \mathbf{do}\ \langle\text{comm}\rangle
\end{aligned}$$

with operations for looking up and mutating the contents of addresses:

$$\langle\text{exp}\rangle ::= [\langle\text{exp}\rangle]$$

$$\langle\text{comm}\rangle ::= [\langle\text{exp}\rangle] := \langle\text{exp}\rangle$$

concurrent composition:

$$\langle\text{comm}\rangle ::= \langle\text{comm}\rangle \parallel \langle\text{comm}\rangle$$

and critical regions:

$$\langle\text{comm}\rangle ::= \textbf{with}\ \langle\text{lock}\rangle\ \textbf{do}\ \langle\text{comm}\rangle \mid \textbf{with}\ \langle\text{lock}\rangle\ \textbf{when}\ \langle\text{boolexp}\rangle\ \textbf{do}\ \langle\text{comm}\rangle$$

(In fact, the unconditional critical region **with** k **do** c can be regarded as an abbreviation for the conditional critical region **with** k **when true do** c. We treat it as a separate form for expository reasons.)

We assume that constants are integers, and that variables and locks are unstructured syntactic names (which are not integers). We also identify addresses with integers. Then we define a location to be either a variable or an address, and a state to be a mapping from a finite set of locations to integers:

$$\begin{aligned}
\text{Addresses} &= \text{Integers} \\
\text{Locations} &= \langle\text{var}\rangle \uplus \text{Addresses} \\
\text{States} &= \bigcup\{\, \delta \to \text{Integers} \mid \delta \overset{\text{fin}}{\subseteq} \text{Locations}\,\}\ .
\end{aligned}$$

We will use the following metavariables (with occasional decorations) to range over specific sets:

v : ⟨var⟩ (Variables)	δ : Finite Sets of Locations
e : ⟨exp⟩ (Expressions)	σ : States
b : ⟨boolexp⟩ (Boolean Expressions)	k : ⟨lock⟩ (Locks)
c : ⟨comm⟩ (Commands)	κ : Finite Sets of Locks
n : Integers	τ : Traces
t : Truth Values	T : Sets of Traces
ℓ : Locations	Φ : Configurations

(Traces and configurations will be defined later.)

We will also need some concepts and notations for states. We say that σ and σ' are *compatible*, written $\sigma \smile \sigma'$, iff $\sigma \cup \sigma'$ is a function, or equivalently, σ and σ' agree on the intersection of their domains. We also write $\delta \perp \delta'$ when the sets δ and δ' are disjoint, and $\sigma \perp \sigma'$ when $\operatorname{dom}\sigma \perp \operatorname{dom}\sigma'$.

When $\ell_1, \ldots, \ell_m$ are distinct, we write $[\,\ell_1\colon n_1 \mid \ldots \mid \ell_m\colon n_m\,]$ for the state with domain $\{\ell_1, \ldots, \ell_m\}$ that maps each ℓ_i into n_i. We also write $[\,\sigma \mid \ell\colon n\,]$ for the state such that

$$\begin{aligned}
\operatorname{dom}[\,\sigma \mid \ell\colon n\,] &= \operatorname{dom}\sigma \cup \{\ell\} \\
[\,\sigma \mid \ell\colon n\,](\ell) &= n \\
[\,\sigma \mid \ell\colon n\,](\ell') &= \sigma(\ell') \text{ when } \ell \neq \ell'\ .
\end{aligned}$$

(Note that $[\,\sigma \mid \ell\colon n\,]$ may either be an extension of σ or a possibly altered function with the same domain as σ.)

The meaning $[\![e]\!]$ of an expression (or boolean expression) e is a set of pairs $\langle \sigma, n \rangle$ in which n is the value obtained by evaluating e in any state that is an extension of σ, and in which the domain of σ is the *footprint* of the evaluation, i.e., the set of locations that are actually examined during the evaluation.

For example,

$$
\begin{aligned}
[\![\mathsf{x} - \mathsf{x}]\!] &= \{\, \langle [\,\mathsf{x}\colon m\,], 0 \rangle \mid m \in \text{Integers} \,\} \\
[\![\mathsf{x} + [\mathsf{y}]]\!] &= \{\, \langle [\,\mathsf{x}\colon m \mid \mathsf{y}\colon n \mid n\colon n'\,], m + n' \rangle \mid m, n, m' \in \text{Integers} \,\} \,.
\end{aligned}
$$

The relevant semantics equations are

$$
[\![\langle \text{exp} \rangle]\!] \subseteq \text{States} \times \text{Integers}
$$

$$
\begin{aligned}
[\![n]\!] &= \{ \langle [\,], n \rangle \} \\
[\![v]\!] &= \{\, \langle [\,v\colon n\,], n \rangle \mid n \in \text{Integers} \,\} \\
[\![e + e']\!] &= \{\, \langle \sigma \cup \sigma', n + n' \rangle \mid \langle \sigma, n \rangle \in [\![e]\!], \langle \sigma', n' \rangle \in [\![e']\!], \text{ and } \sigma \smile \sigma' \,\} \\
[\![[e]]\!] &= \{\, \langle \sigma \cup [\,n\colon n'\,], n' \rangle \mid \langle \sigma, n \rangle \in [\![e]\!], n' \in \text{Integers}, \text{ and } \sigma \smile [\,n\colon n'\,] \,\}
\end{aligned}
$$

$$
[\![\langle \text{boolexp} \rangle]\!] \subseteq \text{States} \times \text{Bool}
$$

$$
\begin{aligned}
[\![e = e']\!] &= \{\, \langle \sigma \cup \sigma', n = n' \rangle \mid \langle \sigma, n \rangle \in [\![e]\!], \langle \sigma', n' \rangle \in [\![e']\!], \text{ and } \sigma \smile \sigma' \,\} \\
[\![b \wedge b']\!] &= \{\, \langle \sigma, \mathbf{false} \rangle \mid \langle \sigma, \mathbf{false} \rangle \in [\![b]\!] \,\} \\
&\quad \cup \{\, \langle \sigma \cup \sigma', t' \rangle \mid \langle \sigma, \mathbf{true} \rangle \in [\![b]\!], \langle \sigma', t' \rangle \in [\![b']\!], \text{ and } \sigma \smile \sigma' \,\} \,.
\end{aligned}
$$

(Note that the final equation describes short-circuit evaluation of conjunction.)

Since expression evaluation in our programming language happens to be deterministic (though this is not required by the nature of our semantics), the meaning of an expression will be a function, but because its domain contains only states whose domains are footprints, this function will be a restriction of the meaning in a conventional denotational semantics.

Nevertheless, one can show an appropriate property of *totality*: For all e and σ, there are σ' and n such that $\sigma \smile \sigma'$ and $\langle \sigma', n \rangle \in [\![e]\!]$. A similar property holds for boolean expressions.

4 Traces and the Semantics of Commands

Actions can be defined grammatically:

$$
\langle \text{action} \rangle ::= \mathbf{start}(\langle \text{state} \rangle) \mid \mathbf{fin}(\langle \text{state} \rangle) \mid \mathbf{try}(\langle \text{lock} \rangle) \mid \mathbf{acq}(\langle \text{lock} \rangle) \mid \mathbf{rel}(\langle \text{lock} \rangle)
$$

Then a *trace* is a finite or infinite sequence of actions, or a finite sequence of actions followed by either **wrong** or $\perp$.

We use ";" to denote the following concatenation of traces:

$$\tau_1 \,;\, \tau_2 = \begin{cases} \tau_1 & \text{if } \tau_1 \text{ is infinite, or ends in } \mathbf{wrong} \text{ or } \bot, \\ \tau_1\, \tau_2 & \text{otherwise.} \end{cases}$$

Then we can define the concatenation and exponentiation of trace sets in a standard way:

$$\begin{aligned} T\,;T' &= \{\, \tau\,;\tau' \mid \tau \in T, \tau' \in T' \,\} \\ T^0 &= \{\epsilon\} \\ T^{n+1} &= T\,;T^n \\ T^* &= \textstyle\bigcup_{n=0}^{\infty} T^n \\ T^\omega &= \{\, \tau_0\,;\tau_1\,;\cdots \mid \forall i \geq 0.\ \tau_i \in T \,\}\,. \end{aligned}$$

We will also need the concept of a *filter* to describe the use of boolean expressions to determine control flow:

$$\begin{aligned} &\text{filter}(\langle\text{boolexp}\rangle) \subseteq \text{Traces} \\ &\qquad \text{filter}(b) = \{\, \mathbf{start}(\sigma)\, \mathbf{fin}(\sigma) \mid \langle \sigma, \mathbf{true} \rangle \in [\![b]\!] \,\}\,. \end{aligned}$$

From the totality property of boolean expressions, one can obtain a totality condition for filters: For all b and σ, there is a σ' such that $\sigma \smile \sigma'$ and

$$\mathbf{start}(\sigma')\, \mathbf{fin}(\sigma') \in \text{filter}(b) \cup \text{filter}(\neg\, b)\,.$$

With these preliminaries, we can give semantic equations that determine the sets of traces that are meanings of sequential commands:

$$\begin{aligned} &[\![\langle\text{comm}\rangle]\!] \subseteq \text{Traces} \\ [\![v := e]\!] &= \{\, \mathbf{start}(\sigma \cup [\, v\colon n_{\text{old}}\,])\, \mathbf{fin}([\, \sigma \mid v\colon n\,]) \mid \\ &\qquad\qquad \langle \sigma, n \rangle \in [\![e]\!], \sigma \smile [\, v\colon n_{\text{old}}\,] \,\} \\ [\![[e] := e']\!] &= \{\, \mathbf{start}(\sigma \cup \sigma' \cup [\, n\colon n_{\text{old}}\,])\, \mathbf{fin}([\, \sigma \cup \sigma' \mid n\colon n'\,]) \mid \\ &\qquad \langle \sigma, n \rangle \in [\![e]\!], \langle \sigma', n' \rangle \in [\![e']\!], \sigma \smile \sigma', (\sigma \cup \sigma') \smile [\, n\colon n_{\text{old}}\,] \,\} \\ [\![\mathbf{skip}]\!] &= \{\mathbf{start}[\,]\, \mathbf{fin}[\,]\} \\ [\![c_1\,;c_2]\!] &= [\![c_1]\!]\,;[\![c_2]\!] \\ [\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] &= (\text{filter}(b)\,;[\![c_1]\!]) \cup (\text{filter}(\neg\, b)\,;[\![c_2]\!]) \\ [\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] &= \big((\text{filter}(b)\,;[\![c]\!])^*\,;\text{filter}(\neg\, b)\big) \cup (\text{filter}(b)\,;[\![c]\!])^\omega\,. \end{aligned}$$

For example

$$
\begin{aligned}
&[\![[\mathsf{x}] := [\mathsf{y}] + 1]\!] = \\
&\qquad \{\, \mathbf{start}[\,\mathsf{x}\colon m \mid m\colon m' \mid \mathsf{y}\colon n \mid n\colon n'\,]\; \mathbf{fin}[\,\mathsf{x}\colon m \mid m\colon n'+1 \mid \mathsf{y}\colon n \mid n\colon n'\,] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid\; m, m', n, n' \in \text{Integers and } m \neq n \,\} \\
&\quad \cup \{\, \mathbf{start}[\,\mathsf{x}\colon n \mid \mathsf{y}\colon n \mid n\colon n'\,]\; \mathbf{fin}[\,\mathsf{x}\colon n \mid \mathsf{y}\colon n \mid n\colon n'+1\,] \;\mid\; n, n' \in \text{Integers} \,\}\,.
\end{aligned}
$$

To define concurrent composition, we must first introduce the concept of a *fair merge* of the traces τ_1 and τ_2, which is an merge (or interleaving) that contains every occurrence of actions in τ_1 and τ_2 (even when τ_1 or τ_2 is infinite).

To make this concept precise, we regard a trace as a function whose domain is the finite or infinite set of nonnegative numbers less than the length of the trace. Then τ is a *fair merge* of τ_1 and τ_2 iff there are functions ϕ_1 and ϕ_2 such that

$\mathrm{dom}\,\phi_1 = \mathrm{dom}\,\tau_1$ and $\mathrm{dom}\,\phi_2 = \mathrm{dom}\,\tau_2$.

ϕ_1 and ϕ_2 are strictly monotone.

The ranges of ϕ_1 and ϕ_2 are a partition of $\mathrm{dom}\,\tau$.

For all $i \in \mathrm{dom}\,\tau_1$, $\tau_1(i) = \tau(\phi_1(i))$.

For all $i \in \mathrm{dom}\,\tau_2$, $\tau_2(i) = \tau(\phi_2(i))$.

Next, we define

$$\tau_1 \parallel \tau_2 = \{\, \mathrm{truncate}(\tau) \mid \tau \text{ is a fair merge of } \tau_1 \text{ and } \tau_2 \,\}\,,$$

where truncate is a function that captures the fact that **wrong** and $\perp$ always terminate traces: When τ does not contain **wrong** or $\perp$,

$$
\begin{aligned}
\mathrm{truncate}(\tau) &= \tau \\
\mathrm{truncate}(\tau\ \mathbf{wrong}\ \tau') &= \tau\ \mathbf{wrong} \\
\mathrm{truncate}(\tau \perp \tau') &= \tau \perp .
\end{aligned}
$$

Finally, the meaning of the concurrent composition $c_1 \parallel c_2$ is the set of truncated fair merges of traces in the meaning of c_1 with traces in the meaning of c_2:

$$[\![c_1 \parallel c_2]\!] = \bigcup \{\, \tau_1 \parallel \tau_2 \mid \tau_1 \in [\![c_1]\!], \tau_2 \in [\![c_2]\!] \,\}\,.$$

Our semantics of critical regions follows closely that of Brookes [6]:

$$[\![\mathbf{with}\ k\ \mathbf{do}\ c]\!] = \big(\{\mathbf{try}(k)\}^* ; \{\mathbf{acq}(k)\} ; [\![c]\!] ; \{\mathbf{rel}(k)\}\big) \cup \{\mathbf{try}(k)\}^\omega$$

$$
\begin{aligned}
&[\![\mathbf{with}\ k\ \mathbf{when}\ b\ \mathbf{do}\ c]\!] = \\
&\qquad \big(\mathrm{wait}^* ; \{\mathbf{try}(k)\}^* ; \{\mathbf{acq}(k)\} ; \mathrm{filter}(b) ; [\![c]\!] ; \{\mathbf{rel}(k)\}\big) \cup \mathrm{wait}^\omega\,,
\end{aligned}
$$

where

$$\mathrm{wait} = \big(\{\mathbf{try}(k)\}^* ; \{\mathbf{acq}(k)\} ; \mathrm{filter}(\neg\, b) ; \{\mathbf{rel}(k)\}\big) \cup \{\mathbf{try}(k)\}^\omega\,.$$

Notice that **with** k **when** b **do** c will fail to terminate if either it fails to ever acquire the lock k, or if it acquires k, but never when b is true.

There is a concept of totality that is appropriate to sets of traces: A set T of traces is said to be *total* whenever, if

$$\tau\,\mathbf{start}(\sigma_0)\,\tau' \in T\ ,$$

holds for some state σ_0, then for every state σ there is a trace

$$\tau\,\mathbf{start}(\sigma')\,\tau'' \in T\ ,$$

such that $\sigma \smile \sigma'$.

There is also a somewhat analogous concept related to locks: A set T of traces is *lock-total* whenever, if

$$\tau\,\mathbf{acq}(k)\,\tau' \in T\ ,$$

then there is a trace

$$\tau\,\mathbf{try}(k)\,\tau'' \in T\ .$$

It can be shown that, for all commands c, $[\![c]\!]$ is total and lock-total.

5 Executing Traces

The execution of traces is described by a small-step operational semantics. A configuration Φ consists of a trace to be executed, coupled with the currently available state and a finite set of closed locks, or it is one of three special configurations that indicate abnormal termination or explicit nontermination:

$$\begin{aligned}&\text{Configurations} =\\ &\qquad \text{Finite Sets of Locks} \times \text{States} \times \text{Traces} \cup \{\text{wrong}, \text{impossible}, \bot\}\ .\end{aligned}$$

A configuration is *nonterminal* if it contains a nonempty trace; otherwise it is *terminal.* (For readability, we omit empty traces from terminal configuations.)

Informally, the special terminal configurations have the following meanings:

- "wrong" indicates that a **start** operation has tried to access a location that is not in the domain of the currently available state.
- "impossible" indicates that a **fin** operation has tried to extend the current state at a location that is already in its domain, or that a **rel** has tried to open a lock that is not closed.
- $\bot$ indicates that the trace will execute forever without performing further actions.

Transitions go from nonterminal configurations to configurations that are either nonterminal or terminal. They are described by the relation

$$\rightarrow\ \subseteq \text{Nonterminal Configurations} \times \text{Configurations}$$

that is the least relation satisfying:

$$\kappa, \sigma, \mathbf{start}(\sigma')\, \tau \to \begin{cases} \kappa, \sigma - \sigma', \tau & \text{if } \sigma' \subseteq \sigma \\ \text{wrong} & \text{if } \sigma \smile \sigma' \text{ and } \sigma' \not\subseteq \sigma \\ \textbf{no transition} & \text{if } \sigma \not\smile \sigma' \end{cases}$$

$$\kappa, \sigma, \mathbf{fin}(\sigma')\, \tau \to \begin{cases} \kappa, \sigma \cup \sigma', \tau & \text{if } \sigma' \perp \sigma \\ \text{impossible} & \text{otherwise} \end{cases}$$

$$\kappa, \sigma, \mathbf{try}(k)\, \tau \to \begin{cases} \kappa, \sigma, \tau & \text{if } k \in \kappa \\ \textbf{no transition} & \text{if } k \notin \kappa \end{cases}$$

$$\kappa, \sigma, \mathbf{acq}(k)\, \tau \to \begin{cases} \kappa \cup \{k\}, \sigma, \tau & \text{if } k \notin \kappa \\ \textbf{no transition} & \text{if } k \in \kappa \end{cases}$$

$$\kappa, \sigma, \mathbf{rel}(k)\, \tau \to \begin{cases} \kappa - \{k\}, \sigma, \tau & \text{if } k \in \kappa \\ \text{impossible} & \text{if } k \notin \kappa \end{cases}$$

$$\begin{aligned} \kappa, \sigma, \mathbf{wrong} \to &\quad \text{wrong} \\ \kappa, \sigma, \perp \to &\quad \perp . \end{aligned}$$

It is easy to see that $\to$ is a partial, but not total function.

A sequence of configurations that begins with the nonterminal $\Phi = \kappa, \sigma, \tau$, and in which each configuration is related to the next, is called an *execution of τ in κ, σ*. If there is such an execution that is finite and ends with Φ', we write $\Phi \to^* \Phi'$; if there is such an execution that is infinite, we say that Φ diverges.

Since $\to$ is a partial function, if Φ diverges, then there is no terminal Φ' such that $\Phi \to^* \Phi'$, while if Φ does not diverge, then there is at most one Φ' such that $\Phi \to^* \Phi'$. In other words, the execution of a trace is always determinate.

On the other hand, since $\to$ is not a total function, there are nonterminal $\Phi = \kappa, \sigma, \tau$ such that Φ does not diverge and there is no terminal Φ' such that $\Phi \to^* \Phi'$. In this case, there is no execution of τ in κ, σ, and we say that τ is *irrelevant* to κ, σ.

Another property of trace execution stems from from the fact that in any trace of any command in our programming language, the actions **acq** and **rel**, and also **start** and **fin**, are balanced, in much the same sense as with parentheses. Specifically, in any prefix of any trace of any command, the number of occurrences of $\mathbf{rel}(k)$ for a particular lock k will never exceed the number of occurrences of $\mathbf{acq}(k)$ for the same lock, and the number of occurrences of $\mathbf{fin}(\sigma)$ for which a particular variable occurs in $\operatorname{dom} \sigma$ will never exceed the number of occurrences of $\mathbf{start}(\sigma')$ for which the same variable occurs in $\operatorname{dom} \sigma'$.

Because of this property (which might not hold for a lower-level programming language), one can show that the abnormal termination "impossible" never arises: For all commands c, traces $\tau \in [\![c]\!]$, lock sets κ, and states σ:

$$\kappa, \sigma, \tau \not\to^* \text{impossible} .$$

Finally, we define the execution of a set of traces T, in κ, σ:

$$\begin{aligned}\text{Exec}(\kappa, \sigma, T) = \{\, \langle \kappa', \sigma' \rangle \mid \exists \tau \in T.\ \kappa, \sigma, \tau \to^* \kappa', \sigma' \,\} \\ \cup\ \textbf{if } \exists \tau \in T.\ \kappa, \sigma, \tau \to^* \text{wrong } \textbf{then } \text{Wrong } \textbf{else } \{\} \\ \cup\ \textbf{if } \exists \tau \in T.\ \kappa, \sigma, \tau \text{ diverges } \textbf{then } \{\bot\} \textbf{ else } \{\} ,\end{aligned}$$

where Wrong is the set of all terminal configurations, including “wrong”. Since

$$\text{Wrong} \cup S = \text{Wrong} ,$$

for all sets S of terminal configurations, this captures the principle that Wrong overrides other possible outcomes. It also captures the notion that Wrong can be implemented arbitrarily.

In contrast to the execution of a particular trace, the execution of a trace set that is the meaning of a command can be nondeterminate. On the other hand, since command meanings are total and lock-total, $\text{Exec}(\kappa, \sigma, [\![c]\!])$ contains at least one terminal configuration for every command c, state σ, and finite lock set κ.

6 Future Directions

Our trace semantics leads to a broader notion of observational equivalence than the conventional semantics of shared-variable concurrency. For example, in a conventional semantics the commands

$$\mathsf{x} := \mathsf{x} + 1 \,; \mathsf{x} := \mathsf{x} + 2 \qquad \text{and} \qquad \mathsf{x} := \mathsf{x} + 3$$

are distinguishable when run concurrently with, say, $\mathsf{x}{:=}1$. But in our semantics, these commands are observationally equivalent: They would both go wrong if run concurrently with any command that assigns to or evaluates x; otherwise they would both increase x by three.

As suggested by this example, the broader notion of observational equivalence should provide greater scope for the development of code optimization. Unfortunately, however, our trace semantics is far from fully abstract; for example, the above commands denote distinct trace sets.

We intend to study equivalences on trace sets that at least approach observational equivalence. For example, we conjecture that every command without critical regions has a meaning that is equivalent to a set of traces whose members each have one of the forms:

$$\begin{aligned}&\textbf{start}(\sigma)\ \textbf{fin}(\sigma') \ \text{ where } \operatorname{dom}\sigma = \operatorname{dom}\sigma' \\ &\textbf{start}(\sigma)\ \bot \\ &\textbf{start}(\sigma)\ \textbf{wrong} .\end{aligned}$$

I hope to report more on this topic in my talk.

We also hope to extend the programming language described by our semantics to include declarations of variables and locks. This should be a straightforward adaptation of the approach used by Brookes for transition traces [7]. It would also be useful to introduce operations for storage allocation and deallocation, perhaps similar to the **cons** and **dispose** operations of separation logic.

Another important direction would be permit passivity, i.e., to relax the assumption that overlapping operations on the same location go wrong, in order to allow the overlapping of read-only operations.

Finally, we hope to use our grainless semantics to model concurrent separation logic [5], and to relate the semantics to Brookes's model [6].

Acknowledgement

The author wishes to thank Stephen D. Brookes for numerous helpful discussions.

References

1. Reynolds, J.C.: Towards a grainless semantics for shared variable concurrency (abstract only). In: Conference Record of POPL 2004: The 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, ACM Press (2004)
2. Hoare, C.A.R.: Towards a theory of parallel programming. In Hoare, C.A.R., Perrott, R.H., eds.: Operating Systems Techniques. Volume 9 of A.P.I.C. Studies in Data Processing, London, Academic Press (1972) 61–71
3. Brinch Hansen, P.: Structured multiprogramming. Communications of the ACM **15** (1972) 574–578
4. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science, Los Alamitos, California, IEEE Computer Society (2002) 55–74
5. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: CONCUR 2004 — Concurrency Theory, Proceedings of the 15th International Conference. Volume 3170 of Lecture Notes in Computer Science, Berlin, Springer-Verlag (2004) 49–67
6. Brookes, S.D.: A semantics for concurrent separation logic. In: CONCUR 2004 — Concurrency Theory, Proceedings of the 15th International Conference. Volume 3170 of Lecture Notes in Computer Science, Berlin, Springer-Verlag (2004) 16–34
7. Brookes, S.D.: Full abstraction for a shared-variable parallel language. Information and Computation **127** (1996) 145–163
8. Park, D.M.R.: On the semantics of fair parallelism. In Bjørner, D., ed.: Abstract Software Specifications. Volume 86 of Lecture Notes in Computer Science, Berlin, Springer-Verlag (1980) 504–526