Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 308 (2014) 65–86

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

On Grainless Footprint Semantics for Shared-memory Programs

Stephen Brookes

Department of Computer Science
Carnegie Mellon University
Pittsburgh, USA

Abstract

We develop an improved grainless denotational semantics for shared-memory parallel programs, building on ideas from earlier trace-based models with local states and footprints [4]. The key new idea is a more refined approach to race detection, leading to a model with better abstraction properties. Rather than treat a race condition as a "global" catastrophe [3,4], we track information about variables whose value may be tainted by a race, and retain accurate information about unaffected variables. As in the prior work, we abstract away from state changes that occur in between synchronization points, in a manner consistent with Dijkstra's Principle [5]. We obtain a model in which only synchronization operations (for locking and unlocking binary semaphores) are deemed to be atomic, which matches the usual implementation of these constructs in an abstract manner. Apart from this, no other atomicity assumptions are made, so our model is truly grainless. Our semantics supports compositional program analysis based on "sequential" reasoning for sequential code fragments, even when this code occurs in parallel contexts, and yields a simple semantic characterization of race-free code. The semantics validates the static constraints on "critical variables" imposed in concurrent programming methodology [6,9] and serves as a foundation for reasoning about safe partial correctness, as in concurrent separation logic [8]. The new treatment of race detection allows for more refined analysis of racy programs. By framing our ideas and concepts in a general manner we hope that our results may be applied in a wider setting.

Keywords: concurrency, shared memory, atomicity, granularity, partial correctness, race condition, denotational semantics

Introduction

Shared-memory programs are difficult to reason about, because of the potential for interference between concurrent processes when updating the same piece of shared state. Similar difficulties arise in constructing semantic models of shared-memory programs. Most traditional models have assumed a fixed granularity of execution, for example atomic assignments or atomic reads and writes [2,10]. Such assumptions

[1] This research was sponsored by the National Science Foundation under grant no. CCF-1017011. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

http://dx.doi.org/10.1016/j.entcs.2014.10.005
1571-0661/© 2014 Elsevier B.V. All rights reserved.

may not hold in practice, and do not hold universally, so program analysis based on these models is only valid for a limited range of implementations. In contrast, in the sequential setting it is safe to ignore granularity, so traditional semantic models for sequential programs focus simply on the initial and final states of program execution, interpreting a program as a state transformation. As is well known, this simple approach is inadequate for parallel programs [10]: the state transformation denoted by $c_1 \| c_2$ cannot be determined from the state transformations denoted by c_1 and c_2. Ideally we need to design a semantic model for parallel programs that enjoys the simplicity of state transformation semantics while retaining enough information about intermediate states to facilitate reasoning about concurrent interference, yet without making overly specific granularity assumptions.

Such concerns have stimulated an effort to design "grainless" semantic models for shared-memory concurrency, notably by John Reynolds [11] and this author [4]. Reynolds sought to avoid granularity by breaking atomic actions into instantaneous fragments, an approach that leads to a semantics based on very small steps, and therefore likely to suffer from combinatorial problems. This author developed a "footstep trace" model, a pre-cursor to the approach offered in this paper, but in retrospect we see now that this model is overly complex and fails full abstraction. Here we offer a more streamlined version of footstep trace semantics, with better abstraction properties. The main new idea involves a more refined account of race conditions, an apparently simple idea with deep ramifications in the construction of the semantics. We classify our semantic model as "grainless" because the model construction abstracts away from irrelevant information about what constitutes an atomic action, other than the usual (and reasonable) assumption that the primitive operations for synchronization (locking and unlocking resources) are atomic, and that resources behave like binary semaphores: at all stages in program execution each resource is held (having been locked and not yet unlocked) by at most one process. Our ideas should generalize to work with other atomic synchronization constructs (for instance Dijkstra-style `P` and `V` operations on semaphores) but we do not consider the details here.

We deal in this paper with a simple shared-memory language, omitting pointers. Our development builds on our prior work on trace models [2] and on concurrent separation logic [3,4], in which we combined mutable state with concurrency, so we expect to be able to adapt the new ideas presented here accordingly by working with states as stores paired with heaps. The extension to pointers and mutable state does introduce new features such as storage allocation and de-allocation, so we would need to adjust the technical details carefully. It is straightforward to incorporate locally scoped declarations, as in the block construct **resource** r **in** c which declares a local resource named r for use solely by c. For space reasons we defer these extensions and some semantic details and proofs to a fuller version of the paper.

1 Syntax and Static Semantics

We consider a simple shared-memory parallel language, while-programs extended with parallel composition and conditional critical regions. Identifiers (or program variables) i are assignable integer-valued variables, and region names (or resources) r take values 0 and 1, representing "available" and "unavailable". The sets **Ide** of identifiers and **Res** of resource names are disjoint. It makes sense to keep resource names and identifiers apart, because of their different syntactic rôles (resources do not appear in assignment commands or in expressions) and the assumptions that we will make about their implementation (resources are only updated atomically, on entry and on exit from critical regions, whereas we make no assumptions about whether or not assignments are atomic).

The syntax of integer expressions e and boolean expressions b is standard, with the usual arithmetic and boolean constructs. For example, the abstract grammar for expressions includes:

$$\begin{aligned} e &::= n \mid i \mid e_1 + e_2 \mid \dots \\ b &::= \mathbf{true} \mid \mathbf{false} \mid e_1 = e_2 \mid \dots \end{aligned}$$

where n ranges over integer numerals.

The static semantics of expressions is given as follows.

Definition 1
We define the sets $\mathtt{free}(e)$ *and* $\mathtt{free}(b)$ *of identifiers with a free occurrence in* e *and* b*, respectively, by structural induction:*

$$\begin{aligned} &\mathtt{free}(n) = \{\} \\ &\mathtt{free}(i) = \{i\} \\ &\mathtt{free}(e_1 + e_2) = \mathtt{free}(e_1) \cup \mathtt{free}(e_2) \\ &\mathtt{free}(\mathbf{true}) = \{\} \\ &\mathtt{free}(\mathbf{false}) = \{\} \\ &\mathtt{free}(e_1 = e_2) = \mathtt{free}(e_1) \cup \mathtt{free}(e_2) \end{aligned}$$

The syntax of commands (or processes) c is given by:

$$\begin{aligned} c \;::=\; &\mathbf{skip} \mid i{:=}e \mid c_1; c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \\ &c_1 \| c_2 \mid \mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c \end{aligned}$$

A *conditional critical region* has the form **with** r **when** b **do** c. A process executing a critical region must lock the resource named r, then evaluate b; if the value of b is **true** the process can then "enter" the critical region and execute c, then finish by unlocking the resource. If b is **false** the process unlockes the resource and tries again.

Resources behave like binary semaphores with atomic operations for locking and unlocking, so conditional critical regions can be used to ensure mutually exclusive access to shared variables. We abbreviate **with** r **when true do** c as **with** r **do** c.

The static semantics of commands is summarized in the following definitions.

Definition 2
We define the set $\mathtt{res}(c)$ *of resource names with a free occurrence in c by structural induction:*

$$\mathtt{res}(\mathbf{skip}) = \{\}$$

$$\mathtt{res}(i{:=}e) = \{\}$$

$$\mathtt{res}(c_1; c_2) = \mathtt{res}(c_1 \| c_2) = \mathtt{res}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) = \mathtt{res}(c_1) \cup \mathtt{res}(c_2)$$

$$\mathtt{res}(\mathbf{while}\ b\ \mathbf{do}\ c) = \mathtt{res}(c)$$

$$\mathtt{res}(\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c) = \{r\} \cup \mathtt{res}(c)$$

Definition 3 *We define the set* $\mathtt{free}(c)$ *of identifiers with a free occurrence in c by structural induction:*

$$\mathtt{free}(\mathbf{skip}) = \{\}$$

$$\mathtt{free}(i{:=}e) = \{i\} \cup \mathtt{free}(e)$$

$$\mathtt{free}(c_1; c_2) = \mathtt{free}(c_1 \| c_2) = \mathtt{free}(c_1) \cup \mathtt{free}(c_2)$$

$$\mathtt{free}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) = \mathtt{free}(b) \cup \mathtt{free}(c_1) \cup \mathtt{free}(c_2)$$

$$\mathtt{free}(\mathbf{while}\ b\ \mathbf{do}\ c) = \mathtt{free}(b) \cup \mathtt{free}(c)$$

$$\mathtt{free}(\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c) = \mathtt{free}(b) \cup \mathtt{free}(c)$$

2 Dynamic Semantics

A program denotes a set of traces representing interactive computations in which the program and its environment make changes to the shared state. Each step in a trace represents the effect of a finite sequence of actions performed by the program, and records just the overall footprint. We detect the potential for race conditions, involving a write to a shared variable whose value is used in a concurrent update. A race condition can lead to unpredictable behavior, so we use $\top$ to represent the value of a variable whose value is race-dependent, a special value which taints all future computations involving that variable. In this sense we treat races as "locally" catastrophic, and we track accurately the values of variables unaffected by races.

Since we focus on footprints rather than global states we will give details of the basic definitions to be used in our development. Although these may seem unfamiliar at first, later we will establish links with more traditional semantic notions such as (global) state transformations.

States

States are finite partial functions from identifiers and resource names to values. For simplicity we let V be the set of integers and use values 0 ("available") and 1 ("unavailable") for resources. Let $V^\top = V \cup \{\top\}$. Let $\mathbf{Var} = \mathbf{Ide} \cup \mathbf{Res}$. We use ι to range over $\mathbf{Var}$, i over $\mathbf{Ide}$ and r over $\mathbf{Res}$. We use a list-like notation $[\iota_1 : v_1, \ldots, \iota_k : v_k]$ for states, and we may also use set-theoretic notation such as $\{(\iota_1, v_1), \ldots, (\iota_k, v_k)\}$ for the same purpose.

Definition 4 *The set Σ of states is given by:*

$$\Sigma = \{\sigma : \mathbf{Var} \rightharpoonup_{fin} V^\top \mid \forall r \in dom(\sigma) \cap \mathbf{Res}.\ \sigma(r) \in \{0, 1\}\}.$$

We use σ and τ to range over states, and let $res(\sigma) = dom(\sigma) \cap \mathbf{Res}$ be the set of resources used in σ. For a set $X \subseteq \mathbf{Var}$ we let $\sigma\backslash X = \{(\iota, v) \in \sigma \mid \iota \notin X\}$ and $\sigma \restriction X = \{(\iota, v) \in \sigma \mid \iota \in X\}$. When X is a singleton we write $\sigma\backslash\iota$ for $\sigma\backslash\{\iota\}$.

Definition 5 *A state σ is race-free if $\top \notin rge(\sigma)$.*

Definition 6 *Two states σ and τ are consistent, $\sigma \Uparrow \tau$, iff they agree on the values of all relevant variables, i.e. $\forall \iota \in dom(\sigma) \cap dom(\tau).\ \sigma(\iota) = \tau(\iota)$.*

Consistency of σ and τ is the same as requiring that $\sigma \restriction dom(\tau) = \tau \restriction dom(\sigma)$, or that $\sigma \cup \tau$ is also a well-defined state. States with disjoint domains are always consistent.

Definition 7 *We let $[\sigma \mid \iota : v] = (\sigma\backslash\iota) \cup \{(\iota, v^\top)\}$, where $v^\top = \top$ if $(\iota, \top) \in \sigma$, and $v^\top = v$ otherwise. This is the state obtained by updating σ with $\iota : v$, unless $\sigma(\iota) = \top$, in which case the update has no effect.*

We generalize this update operation to multiple updates, writing $[\sigma \mid \tau]$ for the state obtained by updating σ with the updates in the state τ, i.e.

$$[\sigma \mid \tau] = (\sigma\backslash dom(\tau)) \cup \{(\iota, v^\top) \mid (\iota, v) \in \tau\}.$$

When τ is a singleton state $[\iota : v]$ we have $[\sigma \mid \iota : v] = [\sigma \mid [\iota : v]]$, so the notations agree.

Updating is associative: when σ, τ and ρ are states we have

$$[[\sigma \mid \tau] \mid \rho] = [\sigma \mid [\tau \mid \rho]],$$

so we can write $[\sigma \mid \tau \mid \rho]$ without ambiguity.

When X is a set of identifiers we write $[\sigma \mid X \mapsto \top]$ for the state obtained by updating σ with $\{(x, \top) \mid x \in X\}$, i.e. $(\sigma\backslash X) \cup \{(\iota, \top) \mid \iota \in X\}$.

Steps

Steps represent the effect of state changes, and to model footprints of programs we record just the portion of state relevant to a step, rather than the entire global state. So a step will involve a pair of states (σ, σ'), where σ is the piece of state

read and σ' is the piece of state *written*, and we decorate this pair with a *flow relation* $R \subseteq dom(\sigma) \times dom(\sigma')$ indicating in particular which reads influence the value of each write. We require that if $(\iota, \top) \in \sigma$ & $(\iota, \iota') \in R$ then $(\iota', \top) \in \sigma'$, since an update based on a tainted value is also deemed to be tainted. Further, if $\iota \in dom(\sigma')$ we insist that $(\iota, \iota) \in R$, since a write can only be performed if its target is present in the initial state. Coupled with the previous requirement this means that once a variable has been involved in a race its value never "recovers". Note that $dom(\sigma') \subseteq dom(R^{-1})$: for every variable ι written in the step the set $R^{-1}(\iota) \subseteq dom(\sigma)$ indicates the variables whose values in σ influence the update.

Definition 8 *The set Λ of steps consists of all triples (σ, R, σ') with $\sigma, \sigma' \in \Sigma$, such that:*

- $R \subseteq dom(\sigma) \times dom(\sigma')$.
- *For all* $\iota \in dom(\sigma')$, $(\iota, \iota) \in R$.
- *If* $(\iota, \top) \in \sigma$ *and* $(\iota, \iota') \in R$ *then* $(\iota', \top) \in \sigma'$.

We use λ and μ to range over the set of steps.

Definition 9 *For a step $\lambda = (\sigma, R, \sigma')$, let $reads(\lambda) = dom(\sigma)$, $writes(\lambda) = dom(\sigma')$, and $res(\lambda) = dom(\sigma) \cap \mathbf{Res}$. By assumption, $writes(\lambda) \subseteq reads(\lambda)$.*

For a step (σ, R, σ') and $\iota \in dom(\sigma')$, $R^{-1}(\iota)$ is the set of variables used to compute the update for ι. Since R is surjective, we can specify the dependency relation by listing $R^{-1}(\iota)$ for each $\iota \in dom(\sigma')$. We omit the flow relation when we intend the smallest relation satisfying the requirements, often the identity relation on $dom(\sigma')$ or the empty relation when $dom(\sigma') = \{\}$. Where relevant we use U for the universal relation on $dom(\sigma) \times dom(\sigma')$. It is helpful to introduce names for some simple steps. In each case the intended flow relation is obvious:

$$\begin{aligned} lock(r) &= ([r:0], [r:1]) \\ unlock(r) &= ([r:1], [r:0]) \\ read(i, v) &= ([i:v], [\,]) \\ write(i, v, v') &= ([i:v], [i:v']) \\ \delta &= ([\,], [\,]) \end{aligned}$$

δ is an "idle" step, in which nothing is read and nothing is written. The other examples listed here each involve a single variable.

For steps in which multiple reads and writes occur, there may be several possible choices of flow relation, expressing different dependencies. For example, in the step $([x:0, y:0], \{(y, y)\}, [y:1])$ the update to y does not depend on the read of x, whereas in $([x:0, y:0], \{(x, y), (y, y)\}, [y:1])$ the update to y depends on both x and y. These two steps have the same overall effect, but different flow relations.

There are also racy steps such as $([x:0], [x:\top])$ in which the presence of $\top$ indicates tainting of the value of x. States (and steps) can contain both tainted

identifiers and untainted identifiers. For example $([x : \top, y : 0], [x : \top, y : 1])$ is also a valid step; however $([x : \top, y : 0], [x : 0, y : 1])$ is not a valid step, because it violates the rule that a tainted variable stays tainted.

In our development, steps involving resources play a special rôle because we interpret operations to lock and unlock resources as atomic actions. Steps with $res(\lambda) = res(\mu) = \{\}$ are *resource-free*, and (under certain conditions) may be composable, consecutively or concurrently, to produce a single step representing the composite effect. While it is possible to introduce more general forms of concurrent and sequential composition for actions, allowing resourced steps to be composed under reasonable compatibility conditions, we do not do so here. By limiting such composition to resource-free steps we obtain a model in which (only) locking and unlocking behave like atomic actions.

Executing steps

Although steps describe footprints, programs operate on a shared global state, and we need to characterize the effect on the global state of executing a step. Global states are also finite partial functions from variables to values, so Σ as defined before also represents the set of global states.

Given a state σ we can characterize the steps that are *executable* from σ, and their *effect*, with an enabling relation $\Rightarrow \subseteq \Sigma \times \Lambda \times \Sigma$. We write this as an infix relation, writing

$$\sigma \stackrel{\lambda}{\Longrightarrow} \sigma'$$

when step λ is enabled from σ, and its execution causes the state to change to σ'. The definition is intuitive: a step can only be executed from a state containing (as a subset) its start state, and its effect is to perform the updates specified in its write state, leaving all other variables unchanged. We say that the state σ *enables* step $\lambda = (\tau, R, \tau')$ if σ satisfies the read properties expressed in λ, i.e. if $\tau \subseteq \sigma$. When this holds, it is possible to *execute* this step from state σ, which produces the new state $[\sigma \mid \tau]$.

Definition 10 *The enabling relation $\Rightarrow \subseteq \Sigma \times \Lambda \times \Sigma$ is given by:*

$$\sigma \stackrel{(\tau,R,\tau')}{\Longrightarrow} \sigma' \text{ iff } \tau \subseteq \sigma \ \&\ \sigma' = [\sigma \mid \tau'].$$

Referring again to the examples introduced above, note the key facts that

$$\begin{array}{ll}
\sigma \stackrel{read(i,v)}{\Longrightarrow} \sigma & \text{iff } i : v \in \sigma \\
\sigma \stackrel{write(i,v,v')}{\Longrightarrow} [\sigma \mid i : v'] & \text{iff } i : v \in \sigma \\
\sigma \stackrel{lock(r)}{\Longrightarrow} [\sigma \mid r : 1] & \text{iff } r : 0 \in \sigma \\
\sigma \stackrel{unlock(r)}{\Longrightarrow} [\sigma \mid r : 0] & \text{iff } r : 1 \in \sigma
\end{array}$$

Also note that $lock(r)$ is not enabled in σ if $\sigma(r) = 1$, and $unlock(r)$ is not enabled in σ if $\sigma(r) = 0$.

It is obvious that $\sigma \stackrel{\delta}{\Longrightarrow} \sigma$ holds, as to be expected: one can always do nothing, and doing nothing does not change the state.

In general when $\sigma \stackrel{\lambda}{\Longrightarrow} \sigma'$ we have $dom(\sigma') = dom(\sigma)$. Intuitively this reflects the fact that programs in our simple language do not allocate fresh storage as they execute.

Consecutive steps

Two resource-free steps λ and μ are *consecutive* (or sequentially executable) if their effects are composable, λ then μ. This is the case when μ *follows* λ, i.e. μ can be enabled after λ, as characterized below.

Definition 11 *For steps $\lambda = (\sigma, R, \sigma')$ and $\mu = (\tau, S, \tau')$ we say that μ follows λ, written $\lambda \smile \mu$, iff $res(\lambda) = res(\mu) = \{\}$ and $\tau \Uparrow [\sigma \mid \sigma']$.*

The requirement that $\tau \Uparrow [\sigma \mid \sigma']$ says that the start state of μ is consistent with the effect of λ. This is equivalent to requiring that $\sigma \Uparrow (\tau \backslash dom(\sigma'))$ and $\tau \Uparrow \sigma'$. Note that the sequential composition operation on steps is *partial*, only defined on steps that satisfy the imposed constraints. When $\lambda \smile \mu$ we define a single step $\lambda; \mu$ to represent the sequential composition (λ then μ).

Definition 12 *When $\lambda = (\sigma, R, \sigma')$ and $\mu = (\tau, S, \tau')$ and $\lambda \smile \mu$, we define*

$$\lambda; \mu \;=\; (\sigma \cup (\tau \backslash dom(\sigma')),\;\; R; S,\;\; [\sigma' \mid \tau' \mid \rho]),$$

where $\rho = \{(\iota', \top) \mid \exists (\iota, \top) \in \sigma'.\ (\iota, \iota') \in S\}$, and where $R; S$ is the relation

$$\begin{aligned} &\{(\iota, \iota'') \mid \exists \iota'.\ (\iota, \iota') \in R \;\&\; (\iota', \iota'') \in S\} \\ \cup\; &\{(\iota, \iota') \in R \mid \iota' \notin dom(S)\} \\ \cup\; &\{(\iota', \iota'') \in S \mid \iota' \notin rge(R)\} \end{aligned}$$

Each step specifies a (piece of) state deemed to be read and a (piece of) state to be written. The composite step $\lambda; \mu$ essentially reads the piece of state read by λ as well as those reads made by μ that were not affected by λ. Similarly the composite step accumulates the writes of λ and the writes of μ in the correct sequential order, adjusted as needed to preserve tainting. The term ρ here propagates the effect of racy updates from the first step. If λ is race-free ρ degenerates to the empty state, and the write effect of the cumulative step is $[\sigma' \mid \tau']$ as expected. The flow relation of the composite step is $R; S$, obtained from the relational composition of R and S, bearing in mind that the first step may write to a variable not used in the second step, and the second step may read a variable not influenced by the first step.

Sequential composition of steps is associative, and δ is a unit.

Lemma 13 *$\lambda_1 \smile \lambda_2$ & $(\lambda_1; \lambda_2) \smile \lambda_3$ iff $\lambda_2 \smile \lambda_3$ & $\lambda_1 \smile (\lambda_2; \lambda_3)$. When these equations hold $\lambda_1; (\lambda_2; \lambda_3) = (\lambda_1; \lambda_2); \lambda_3$. Further, for all steps λ we have $\lambda \smile \delta$, $\delta \smile \lambda$ and $\lambda; \delta = \delta; \lambda = \lambda$.*

Proof: An easy calculation using the definitions. For all relations R, S, T we have $(R; S); T = R; (S; T)$, and $R; \{\} = \{\}; R = R$.

Examples

In each of the following cases, the two steps are sequentially executable and we show the resulting step. As usual the intended flow relations are obvious:

$$read(x, v); read(x, v) = read(x, v)$$

$$read(x, v_1); read(y, v_2) = ([x : v_1, y : v_2], [\,])$$

$$read(x, v); write(x, v, v') = write(x, v, v')$$

$$write(x, v, v'); read(x, v') = write(x, v, v')$$

$$write(x, v_1, v_2); write(x, v_2, v_3) = write(x, v_1, v_3)$$

$$write(x, v_1, v_1'); write(y, v_2, v_2') = ([x : v_1, y : v_2], [x : v_1', y : v_2'])$$

$$write(x, v_1, v_1'); read(y, v) = read(y, v); write(x, v_1, v_1') = ([x : v_1, y : v], [x : v_1'])$$

In particular, reads and writes to distinct identifiers have the same effect in either sequential order. Writes to the same identifier are only sequentially composable if the value read by the second step agrees with the value written in the first step.

As expected, there is a simple relationship between sequential composition of steps and execution, the former dealing with footprint states and the latter with the global state.

Lemma 14 *If $\lambda \smile \mu$, then $\sigma \xrightarrow{\lambda;\mu} \sigma''$ iff $\exists \sigma'.\ (\sigma \xrightarrow{\lambda} \sigma'\ \&\ \sigma' \xrightarrow{\mu} \sigma'')$.*

Concurrent steps

Resource-free steps λ and μ with compatible start states are *concurrently executable*, or *concurrent* for short. The steps *conflict* if one writes to an identifier on which the other one depends, in which case we record the value of the race-sensitive identifier as $\top$. We obtain a composite step $\lambda \otimes \mu$ describing the concurrent combination of λ and μ. Again this operation on steps is *partial*, only defined for concurrent steps.

Definition 15 *Two steps $\lambda = (\sigma, R, \sigma')$ and $\mu = (\tau, S, \tau')$ are concurrent, written λ **co** μ, iff $\sigma \Uparrow \tau$ & $res(\lambda) = res(\mu) = \{\}$. When this holds we define*

$$\lambda \otimes \mu = (\sigma \cup \tau,\ R \cup S,\ [(\sigma' \cup \tau') \mid X \mapsto \top]),$$

where X is the set of identifiers whose value is susceptible to a race condition, i.e. those identifiers written by one of the steps with a value dependent on an identifier affected by the other step:

$$X = \{\iota \in dom(\sigma') \mid R^{-1}(\iota) \cap dom(\tau') \neq \{\}\}$$
$$\cup\ \{\iota \in dom(\tau') \mid S^{-1}(\iota) \cap dom(\sigma') \neq \{\}\}.$$

Since σ and τ are assumed to be compatible ($\sigma \Uparrow \tau$) the union $\sigma \cup \tau$ is a partial function, hence a valid state. The flow relation $R \cup S$ combines the flow information from the two steps in the obvious way. Even though the union $(\sigma' \cup \tau')$ may not be a partial function when $dom(\sigma') \cap dom(\tau') \neq \{\}$, performing the update $X \mapsto \top$ does produce a partial function, hence a valid state, since $X \supseteq dom(\sigma') \cap dom(\tau')$, $dom(\sigma') \subseteq dom(R^{-1})$, and $dom(\tau') \subseteq dom(S^{-1})$.

We can paraphrase the above definition, perhaps in a more readable manner, as saying that $\lambda \otimes \mu = (\sigma \cup \tau, R \cup S, \theta)$, where θ is the state with $dom(\theta) = dom(\sigma') \cup dom(\tau')$ and the following properties:

- $\theta(\iota) = \sigma'(\iota)$ if $\iota \in dom(\sigma')$ and $R^{-1}(\iota) \cap dom(\tau') = \{\}$;
- $\theta(\iota) = \tau'(\iota)$ if $\iota \in dom(\tau')$ and $S^{-1}(\iota) \cap dom(\sigma') = \{\}$;
- $\theta(\iota) = \top$ otherwise.

Concurrent combination of steps is an associative operation, and δ is again a unit.

Lemma 16 *λ_1 **co** λ_2 iff λ_2 **co** λ_1, and when these hold $\lambda_1 \otimes \lambda_2 = \lambda_2 \otimes \lambda_1$. Further, ($\lambda_1$ **co** λ_2 and ($\lambda_1 \otimes \lambda_2$) **co** λ_3) iff (λ_2 **co** λ_3 and λ_1 **co** ($\lambda_2 \otimes \lambda_3$)), and when these hold $(\lambda_1 \otimes \lambda_2) \otimes \lambda_3 = \lambda_1 \otimes (\lambda_2 \otimes \lambda_3)$. For all steps λ we have λ **co** δ and $\delta \otimes \lambda = \lambda \otimes \delta = \lambda$.*

It is also obvious that λ **co** μ iff μ **co** λ, and $\lambda \otimes \mu = \mu \otimes \lambda$ whenever both defined.

Examples

- Reads and writes with compatible start states can be executed in parallel, and reads and writes to distinct variables x and y never conflict:

$$write(x, v_1, v_1') \otimes write(y, v_2, v_2') = ([x : v_1, y : v_2], [x : v_1', y : v_2'])$$
$$read(x, v_1) \otimes read(y, v_2) = ([x : v_1, y : v_2], [\,])$$
$$read(x, v_1) \otimes write(y, v_2, v_2') = ([x : v_1, y : v_2], [y : v_2'])$$
$$read(x, v_1) \otimes read(y, v_2) = ([x : v_1, y : v_2], [\,])$$

- For steps affecting the same single variable, concurrent reads are benign but concurrent writes constitute a race:

$$read(x, v) \otimes read(x, v) = read(x, v) = ([x : v], [\,])$$
$$write(x, v, v') \otimes read(x, v) = write(x, v, v') = ([x : v], [x : v'])$$
$$write(x, v, v_1') \otimes write(x, v, v_2') = ([x : v], [x : \top])$$

- A write to x concurrent with a read of x only constitutes a significant race if the read influences the value of some identifier. The following examples show how races get handled, specifically that the correct identifiers get tainted, when the flow relations are non-trivial. To facilitate comparison with the above definition we enumerate the flow relations explicitly.

(a) Consider a step that updates y and reads x but does not use the value of x in the update. If we concurrently write to x, the combined effect is to update both x and y as intended. For example,

$$([x:1,y:0],\{(y,y)\},[y:1]) \otimes ([x:1],\{(x,x)\},[x:2])$$
$$= ([x:1,y:0],\{(x,x),(y,y)\},[x:2,y:1])$$

(b) Now consider a step that updates y using a value obtained by reading x. If we concurrently write to x the value of y is tainted. For example,

$$([x:1,y:0],\{(x,y),(y,y)\},[y:1]) \otimes ([x:1],\{(x,x)\},[x:2])$$
$$= ([x:1,y:0],\{(x,x),(x,y),(y,y)\},[x:2,y:\top])$$

In each case the concurrent composite step reflects the overall effect, and indicates tainting when relevant.

There is an obvious connection between concurrent composition and enabling. Again the former notion concerns footprint states and the latter involves the global state.

Theorem 17
*When λ and μ have identity flow relations, λ **co** μ, and $\sigma \stackrel{\lambda\otimes\mu}{\Longrightarrow} \sigma'$ there are states σ_1, σ_2 such that $\sigma \stackrel{\lambda}{\Longrightarrow} \sigma_1, \sigma \stackrel{\mu}{\Longrightarrow} \sigma_2$, and σ' is uniquely determined by the following properties:*

- $\sigma'(\iota) = \sigma_1(\iota)$ *for* $\iota \in writes(\lambda) - reads(\mu)$
- $\sigma'(\iota) = \sigma_2(\iota)$ *for* $\iota \in writes(\mu) - reads(\lambda)$
- $\sigma'(\iota) = \top$ *for* $\iota \in writes(\lambda) \cap reads(\mu)$ *or* $writes(\mu) \cap reads(\lambda)$
- $\sigma'(\iota) = \sigma(\iota)$ *for* $\iota \in dom(\sigma) - (writes(\lambda) \cup writes(\mu))$.

A more general relationship is derivable, involving R and S in a natural manner.

Traces

Traces are finite sequences of steps. We want to abstract away from intermediate states occurring between resource actions, in the spirit of Dijkstra's Principle: *processes should be regarded as independent, except when they synchronize* [5]. Hence each step should be either an (atomic) resource action or a resource-free step representing the cumulative effect of a finite computation, and we will work with traces in which adjacent resource-free steps are sequentially executable and get "mumbled" together using sequential composition. This will indeed abstract away from the order of intermediate reads and writes occur, since a mumbled step only reports the cumulative reads and writes. We lose no generality when considering the effect on the shared state as viewed by other process running concurrently. The only way another process could be influenced by or affect an intermediate stage would be by reading or writing to a variable written by this step, causing a race condition. The

mumbled step also writes to this variable, so we would also get a race condition under these circumstances and nothing is lost by keeping only the mumbled step.

To build traces with this reduced structure we introduce a modified form of concatenation, a partial operation $\cdot$ that combines traces whose concatenation can be properly reduced by such mumbling, and implements this reduction.

Definition 18 *λ precedes μ, written $\lambda \triangleleft \mu$, if $\lambda \smile \mu$ or $res(\lambda) \cup res(\mu) \neq \{\}$. When this holds we define $\lambda \cdot \mu$ to be the step given by:*

$$\begin{aligned} \lambda \cdot \mu &= \lambda\mu \quad \textit{if } res(\lambda) \cup res(\mu) \neq \{\} \\ &= \lambda;\mu \textit{ if } res(\lambda) = res(\mu) = \{\} \textit{ and } \lambda \smile \mu \end{aligned}$$

We refer to this operation from now on simply as concatenation. We extend to finite traces in the obvious inductive manner. Let ϵ be the empty trace.

Definition 19 *For all $\alpha, \beta, \lambda, \mu$ we have $\epsilon \triangleleft \beta$, $\alpha \triangleleft \epsilon$, and $(\alpha\lambda) \triangleleft (\mu\beta)$ iff $\lambda \triangleleft \mu$. Further, $\epsilon \cdot \beta = \beta, \alpha \cdot \epsilon = \alpha$, and when $\lambda \triangleleft \mu$ we let $(\alpha\lambda) \cdot (\mu\beta) = \alpha(\lambda \cdot \mu)\beta$.*

It is easy to verify that $\delta \cdot \delta = \delta$ and that $\cdot$ is associative.

Theorem 20 *For all traces α, β, γ, $\alpha \triangleleft \beta$ & $(\alpha \cdot \beta) \triangleleft \gamma$ iff $\beta \triangleleft \gamma$ & $\alpha \triangleleft (\beta \cdot \gamma)$ and when these hold, $(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$.*

Examples
Clearly $write(x,0,1) \cdot write(x,1,2) = write(x,0,2)$, and

$$lock(r) \cdot write(x,0,1) = lock(r)\, write(x,0,1)$$

$$lock(r) \cdot write(x,0,1) \cdot write(x,1,2) \cdot unlock(r) = lock(r)\, write(x,0,2)\, unlock(r)$$

Reducing traces
We say that a trace α is *reduced* (or "mumbled") iff for all pairs of successive steps $\lambda\mu$ in α either $res(\lambda) \neq \{\}$ or $res(\mu) \neq \{\}$. It is easy to see that any trace built using $\cdot$ is reduced, because when $\alpha\lambda$ and $\mu\beta$ are reduced traces and $\lambda \triangleleft \mu$, $\alpha(\lambda \cdot \mu)\beta$ is also a reduced trace. We say that $\alpha = \lambda_1 \ldots \lambda_n$ is *feasible* iff its steps can be combined using $\cdot$, and when this happens we obtain the reduced trace $red(\alpha) = \lambda_1 \cdot \ldots \cdot \lambda_n$.

For example, the trace

$$\alpha = lock(r)\, write(x,0,1)\, write(x,1,2)\, unlock(r)\, lock(r)\, write(x,0,1)\, unlock(r)$$

is feasible, and $red(\alpha)$ is the trace

$$lock(r)\, write(x,0,2)\, unlock(r)\, lock(r)\, write(x,0,1)\, unlock(r).$$

However the trace $write(x,0,2)\, write(x,0,1)$ is not feasible.

Every reduced trace α is also feasible and satisfies the equation $red(\alpha) = \alpha$. Whenever α is feasible, $red(\alpha)$ is a reduced trace.

Executing traces

We extend the effect relation to traces in the obvious way:

$$\sigma \xrightarrow{\lambda_1 \ldots \lambda_n} \sigma' \text{ iff } \exists \sigma_0, \ldots, \sigma_n.\ \sigma = \sigma_0 \ \&\ \sigma_n = \sigma' \\ \&\ \sigma_0 \xrightarrow{\lambda_1} \sigma_1 \xrightarrow{\lambda_2} \sigma_2 \cdots\ \sigma_{n-1} \xrightarrow{\lambda_n} \sigma_n$$

When $n = 0$ we get $\sigma \stackrel{\epsilon}{\Longrightarrow} \sigma$. When $\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$ we say that α is enabled in σ, and σ' is the result of executing α from σ'. A trace is *executable* if it is enabled by some state.

We are mainly interested in executable traces, yet our semantic construction deals with *feasible* traces, including both executable and non-executable traces. This is unavoidable, because a parallel program may have an executable trace that arises by interleaving of non-executable traces. For example, consider the traces $\alpha = \mathit{lock}(r)\, \mathit{write}(x, 0, 1)\, \mathit{unlock}(r)$ and $\beta = \mathit{lock}(r)\, \mathit{write}(x, 1, 0)\, \mathit{unlock}(r)$. The trace $\alpha\alpha$ is non-executable, but $(\alpha\alpha)\|\beta$ contains an executable interleaving $\alpha\beta\alpha$. As this example shows, the executable traces of a parallel program $c_1\|c_2$ cannot always be determined from the executable traces of its component processes c_1 and c_2, i.e. executable traces are not compositional. However the executable traces of $c_1\|c_2$ *can* be determined from the *feasible* traces of c_1 and c_2, and feasible traces can indeed be defined in compositionally. Note that in the above example $\alpha\alpha$ is a feasible trace.

The claim that we lose no generality by dropping infeasible traces is validated by the facts that: when α is infeasible there are no states σ such that $\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$, and there is no way to fill the gaps in α without concurrently writing to a variable read by α.

Parallel composition

We adapt the definition of resource-sensitive fair merge from our prior work, adjusted to generate mumbled traces. For traces α and β, and disjoint finite sets of resources A and B, we define the set of feasible merges

$$\alpha\ {}_A\|_B\ \beta \subseteq \Lambda^*$$

by induction on trace length. This set consists of all traces in which a process holding resources A runs concurrently with a process holding resources B. We first define the relation $A \xrightarrow[B]{\lambda} A'$ that characterizes when the process holding A can do step λ in an environment holding B:

$$\begin{array}{ll} A \xrightarrow[B]{\mathit{lock}(r)} A \cup \{r\} & \text{if } r \notin A \cup B \\ A \xrightarrow[B]{\mathit{unlock}(r)} A - \{r\} & \text{if } r \in A \\ A \xrightarrow[B]{\lambda} A & \text{if } \mathit{res}(\lambda) = \{\} \end{array}$$

When one (or both) of the traces is empty we define:

$$\alpha \,{}_A\|_B\, \epsilon = \{\alpha \mid A \xrightarrow[B]{\alpha} A'\}$$
$$\epsilon \,{}_A\|_B\, \beta = \{\beta \mid B \xrightarrow[A]{\alpha} B'\}$$

For the inductive case we define:

$$\begin{aligned}(\lambda\alpha)\,{}_A\|_B\,(\mu\beta) = {} & \{\lambda\cdot\gamma \mid A \xrightarrow[B]{\lambda} A' \;\&\; \gamma \in \alpha_{A'}\|_B\,(\mu\beta) \;\&\; \lambda \lhd \gamma\} \\ \cup {} & \{\mu\cdot\gamma \mid B \xrightarrow[A]{\mu} B' \;\&\; \gamma \in (\lambda\alpha)\,{}_A\|_{B'}\,\beta \;\&\; \mu \lhd \gamma\} \\ \cup {} & \{(\lambda\otimes\mu)\cdot\gamma \mid \lambda \;\mathbf{co}\; \mu, \gamma \in \alpha\,{}_A\|_B\,\beta \;\&\; (\lambda\otimes\mu) \lhd \gamma\}\end{aligned}$$

When $A = B = \{\}$ we omit the subscripts and write $\alpha\|\beta$.

The first two terms produce interleavings in which one process does a step first; the third term allows concurrent combinations when enabled, and takes account of the potential for race conditions. Use of $\cdot$ ensures that we only include feasible traces. The reader can check that when α and β are feasible traces, every trace belonging to $\alpha\|\beta$ is also feasible.

When λ and μ are concurrently enabled (so resource-free) steps, $\lambda\|\mu = \{\lambda\otimes\mu\}$. If in addition the steps are conflict-free, we have $\lambda;\mu = \mu;\lambda = \lambda\otimes\mu$.

When λ and μ are resource-free steps that are not consecutively executable in either order, and not concurrently executable, $\lambda\|\mu = \{\}$.

When at least one of λ and μ is a resource step, $\lambda\|\mu = \{\lambda\mu, \mu\lambda\}$.

Examples

In each of these examples the reader should check that we do obtain the set of all (reduced) feasible combinations. It is also worth noticing that several distinct interleavings may lead to the same reduced trace, showing the succinctness of our construction.

(i) Concurrent writes to distinct variables: $write(x,0,1)\|write(y,0,1)$ yields three feasible interleavings:

$$write(x,0,1)\cdot write(y,0,1)$$
$$write(y,0,1)\cdot write(x,0,1)$$
$$write(x,0,1)\otimes write(y,0,1)$$

and each of these reduces to the same trace $([x:0, y:0], [x:1, y:1])$. So $write(x,0,1)\|write(y,0,1) = \{([x:0, y:0], [x:1, y:1])\}$.

(ii) Concurrently executable writes to the same variable:

$$\begin{aligned}&write(x,0,1)\|write(x,0,1)\\ &\quad = \{write(x,0,1)\otimes write(x,0,1)\}\\ &\quad = \{([x:0], [x:\top])\}.\end{aligned}$$

In this case the updates are not sequentially composable.

(iii) Non-concurrently executable writes to the same variable:

$$\begin{aligned} &write(x,0,1) \| write(x,1,2) \\ &\quad = \{write(x,0,1) \cdot write(x,1,2)\} \\ &\quad = \{write(x,0,2)\} \end{aligned}$$

because there is only one feasible interleaving. On the other hand,

$$write(x,0,1) \| write(x,2,1) = \{\},$$

because the two steps cannot be composed sequentially or concurrently.

Semantics of expressions

Since our semantics is designed to detect race conditions involving concurrent access to shared variables, we can work with a very simple model for expressions. There is no need to keep track of the order in which reads occur during expression evaluation, provided we record the set of reads on which the expression value depends, so we can use a set of state-value pairs. For an integer expression e we will let $[\![e]\!] \subseteq \Sigma \times V$, and for a boolean expression b we let $[\![b]\!] \subseteq \Sigma \times \{true, false\}$. An entry $(\sigma, v) \in [\![e]\!]$ represents the fact that evaluation of e in any state τ such that $\tau \supseteq \sigma$ only reads the portion σ of the state and produces the integer value v, and similarly for boolean expressions. So each entry is a minimal piece of computational information about expression evaluation, in line with our desire to deal with footprints.

Definition 21 *We define* $[\![e]\!]$ *and* $[\![b]\!]$ *by structural induction:*

$$\begin{aligned} &[\![n]\!] = \{([\,], n)\} \\ &[\![i]\!] = \{([i:v], v) \mid v \in V\} \\ &[\![e_1 + e_2]\!] = \{(\sigma_1 \cup \sigma_2, v_1 + v_2) \mid (\sigma_1, v_1) \in [\![e_1]\!] \,\&\, (\sigma_2, v_2) \in [\![e_2]\!] \,\&\, \sigma_1 \Uparrow \sigma_2\} \\ &[\![\mathbf{true}]\!] = \{([\,], true)\} \\ &[\![e_1 = e_2]\!] = \{(\sigma_1 \cup \sigma_2, v_1 = v_2) \mid (\sigma_1, v_1) \in [\![e_1]\!] \,\&\, (\sigma_2, v_2) \in [\![e_2]\!] \,\&\, \sigma_1 \Uparrow \sigma_2\} \end{aligned}$$

We remark that when $(\sigma, v) \in [\![e]\!]$ the piece of state σ contains values for the identifiers occurring free in e whose values are used to compute v.

Theorem 22 *For all expressions* e*, if* $(\sigma, v) \in [\![e]\!]$ *then* $dom(\sigma) \subseteq \mathtt{free}(e)$. [2]

Since we deal here with footprints these semantic clauses are a little more involved than in traditional presentations, which usually assume that the entire global state is at hand.

[2] For the clauses listed above one can prove the stronger property that if $(\sigma, v) \in [\![e]\!]$ then $dom(\sigma) = \mathtt{free}(e)$. This fails if we extend the expression language to include conditionals, whereas the weaker property continues to hold, and our development only relies on the weaker property as stated here.

Semantics of commands

Commands denote sets of feasible traces. To simplify the presentation we introduce the abbreviations $[\![b]\!]_{true}$ for $\{(\sigma, [\,]) \mid (\sigma, true) \in [\![b]\!]\}$ and similarly for $[\![b]\!]_{false}$. Intuitively $[\![b]\!]_{true}$ and $[\![b]\!]_{false}$ are trace sets representing evaluation steps in which the truth value of b is calculated.

Definition 23 *We define the trace set of a command, $[\![c]\!] \subseteq \mathcal{P}(\Lambda^*)$, by structural induction:*

$$[\![\mathbf{skip}]\!] = \{\delta\}$$

$$[\![i{:=}e]\!] = \{(\sigma \cup [i : v], U, [i : v']) \mid (\sigma, v') \in [\![e]\!] \;\&\; \sigma \Uparrow [i : v]\}$$

$$\quad \textit{where } U^{-1}(i) = dom(\sigma) \cup \{i\}$$

$$[\![c_1; c_2]\!] = [\![c_1]\!] \cdot [\![c_2]\!] = \{\alpha_1 \cdot \alpha_2 \mid \alpha_1 \in [\![c_1]\!],\ \alpha_2 \in [\![c_2]\!] \;\&\; \alpha_1 \lhd \alpha_2\}$$

$$[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] = [\![b]\!]_{true} \cdot [\![c_1]\!] \ \cup\ [\![b]\!]_{false} \cdot [\![c_2]\!]$$

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = ([\![b]\!]_{true} \cdot [\![c]\!])^* \cdot [\![b]\!]_{false}$$

$$[\![c_1 \| c_2]\!] = [\![c_1]\!] \| [\![c_2]\!] = \bigcup\{\alpha_1 \| \alpha_2 \mid \alpha_1 \in [\![c_1]\!] \;\&\; \alpha_2 \in [\![c_2]\!]\}$$

$$[\![\mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c]\!] = \{lock(r)\, \beta \cdot \gamma\, unlock(r) \mid \beta \in [\![b]\!]_{true},\ \gamma \in [\![c]\!],\ \beta \lhd \gamma\}$$

Note the careful use of $\lhd$ and $\cdot$ to ensure that only feasible traces are included. It follows by an easy structural induction that for all commands c the set $[\![c]\!]$ defined by these clauses is indeed a set of feasible traces. The implicit use of the sequencing and concurrent composition operations on steps in these clauses is designed to omit irrelevant (infeasible) traces, and to reduce (feasible) traces, and thus abstract away from the order of action occurrences in between resource steps. Hence our semantics truly embodies Dijkstra's Principle. Note that the *executable* traces of c can be extracted from $[\![c]\!]$ as a subset.

The clause for critical regions shows how entry and exit are modeled as $lock(r)$ and $unlock(r)$, and the resource-sensitive nature of interleaving ensures that $c_1 \| c_2$ correctly models concurrent execution while obeying the atomicity and mutual exclusion constraints on resources. Since we only care here about partial correctness behavior we do not include infinite traces to represent the busy-waiting caused by perpetually unavailable resources and/or the falsity of the entry condition b. Similarly the clause for while-loops does not include infinite iteration.

The semantic clause for assignment conceals a slight subtlety: commands do not allocate storage, so an assignment can only be executed from a state in which it target variable already has a value. So the footprint of $i{:=}e$ needs a read state in which i has a value and from which e can be evaluated. To allow for cases where i is not relevant to the value of e as well as when e needs the value of i, we use $\sigma \cup [i : v]$ (with $\sigma \Uparrow [i : v]$) as the read state in the footprint of $i{:=}e$, where $(\sigma, v') \in [\![e]\!]$). When $i \notin dom(\sigma)$, so that i is not needed by e, the consistency constraint holds for all v; when $i \in dom(\sigma)$, so the value of e may depend on i, the consistency

constraint only holds for $v = \sigma(i)$ and the state $\sigma \cup [i : v]$ is the same as σ.

Examples

(i) Our semantics deals properly with multiple assignments in sequence and in parallel, taking account of race conditions:

$$[\![x{:=}1]\!] = \{([x : v], [x : 1]) \mid v \in V\}$$
$$[\![x{:=}1; x{:=}2]\!] = [\![x{:=}2]\!] = \{([x : v], [x : 2]) \mid v \in V\}$$
$$[\![x{:=}1; y{:=}1]\!] = [\![x{:=}1 \| y{:=}1]\!] = \{([x : v_1, y : v_2], [x : 1, y : 1]) \mid v_1, v_2 \in V\}$$
$$[\![x{:=}1 \| x{:=}1]\!] = \{([x : v], [x : \top]) \mid v \in V\}$$

In each of these cases the flow relation is the obvious identity relation.

When one assignment affects a variable used later, our semantics again makes the right distinctions:

$$[\![x{:=}1; y{:=}x]\!] = \{([x : v_1, y : v_2], R, [x : 1, y : 1]) \mid v_1, v_2 \in V\}$$
$$[\![y{:=}x; x{:=}1]\!] = \{([x : v_1, y : v_2], R, [x : 1, y : v_1]) \mid v_1, v_2 \in V\},$$

where $R = \{(x, x), (y, y), (x, y)\}$. Here the flow relation is the same but the write effect differs.

(ii) When we run $x{:=}1; y{:=}x$ in parallel with a command that writes to x there is a race condition involving x and y. For example:

$$[\![(x{:=}1; y{:=}x) \| x{:=}2]\!] = \{([x : v_1, y : v_2], [x : \top, y : \top]) \mid v_1, v_2 \in V\}.$$

From above we see that our semantics distinguishes between $x{:=}1; y{:=}x$ and $x{:=}1; y{:=}1$. This is necessary because in the latter command the value of y is not influenced by the value of x, leading to different behavior in contexts that write to x. Indeed, we have

$$[\![(x{:=}1; y{:=}1) \| x{:=}2]\!] = \{([x : v_1, y : v_2], [x : \top, y : 1]) \mid v_1, v_2 \in V\}.$$

(iii) A slightly more complex example shows how expression evaluation and control flow fit in: $[\![\textbf{if } x > 0 \textbf{ then } y{:=}1 \textbf{ else } z{:=}1]\!]$ is the set

$$\{([x : v_1, y : v_2], U_y, [y : 1]) \mid v_1 > 0, v_2 \in V\}$$
$$\cup \{([x : v_1, z : v_2], U_z, [z : 1]) \mid v_1 \leq 0, v_2 \in V\}$$

where $U_y^{-1}(y) = \{x, y\}$ and $U_z^{-1}(z) = \{x, z\}$.

(iv) For a while-loop with no critical regions, our semantics abstracts away from the intermediate states generated by successive iterations:

$$[\![\textbf{while } y > 0 \textbf{ do } (x{:=}x+1; y{:=}y-1)]\!]$$
$$= \{([x : v_1, y : v_2], U, [x : v_1 + v_2, y : 0]) \mid v_2 > 0\} \cup \{([y : v], [\,]) \mid v \leq 0\}$$

where $U^{-1}(x) = \{x, y\}$.

(v) The only traces of the program (**with** r **do** $x{:=}1$)$\|$(**with** r **do** $x{:=}2$) that are executable from a state in which the values of r and x are 0 are

$$\mathit{lock}(r)\ \mathit{write}(x, 0, 1)\ \mathit{unlock}(r)\ \mathit{lock}(r)\ \mathit{write}(x, 1, 2)\ \mathit{unlock}(r)$$
$$\text{and}\quad \mathit{lock}(r)\ \mathit{write}(x, 0, 2)\ \mathit{unlock}(r)\ \mathit{lock}(r)\ \mathit{write}(x, 2, 1)\ \mathit{unlock}(r).$$

(vi) Consider the program $(x{:=}1; x{:=}2)\|$**with** r **do** $y{:=}x$. Each of its traces arises by interleaving a trace $([x : v], [x : 2])$ of $x{:=}1; x{:=}2$ with a trace of form $\mathit{lock}(r)\,([x : v_1, y : v_2], U_y, [y : v_1])\,\mathit{unlock}(r)$ for some v, v_1, v_2, where $U_y^{-1}(y) = \{x, y\}$. The feasible interleavings are all traces of the following forms:
- $([x : v], [x : 2])\ \mathit{lock}(r)\,([x : v_1, y : v_2], U_y, [y : v_1])\ \mathit{unlock}(r)$
- $\mathit{lock}(r)\,([x : v_1, y : v_2], U_y, [x : 2, y : \top])\ \mathit{unlock}(r)$
- $\mathit{lock}(r)\,([x : v_1, y : v_2], U_y, [x : 2, y : v_1])\ \mathit{unlock}(r)$
- $\mathit{lock}(r)\,([x : v_1, y : v_2], U_y, [x : 2, y : 2])\ \mathit{unlock}(r)$
- $\mathit{lock}(r)\,([x : v_1, y : v_2], U_y, [y : v_1])\ \mathit{unlock}(r)\,([x : v], [x : 2])$

There is no trace in which y gets set to 1; instead there is a racy trace in which y gets set to $\top$. The remaining traces represent computations in which steps are taken sequentially.

Note also that the traces of $(x{:=}1; x{:=}2)\|$**with** r **do** $y{:=}x$ are identical to the traces of $x{:=}2\|$**with** r **do** $y{:=}x$. This can be verified by construction, but it also follows by compositionality of the semantics, since $[\![x{:=}1; x{:=}2]\!]$ is the same as $[\![x{:=}2]\!]$.

(vii) Consider the program **with** r **do** $(x{:=}x+1; x{:=}x+1)$. Our semantics does not distinguish this from **with** r **do** $x{:=}x+2$. There is no need to distinguish them, because no other process can tell them apart without causing a race.

Semantic properties

We begin with an obvious fact connecting static semantics and dynamic semantics.

Theorem 24 *If* $\alpha(\sigma, R, \sigma')\beta \in [\![c]\!]$ *then* $\mathit{dom}(\sigma) \subseteq \mathtt{free}(c)$, $\mathit{dom}(\sigma') \subseteq \mathtt{writes}(c)$, $R \subseteq \mathit{dom}(\sigma) \times \mathit{dom}(\sigma')$, *and* $\mathit{res}(\sigma) \subseteq \mathtt{res}(c)$.

Together with the fact that trace sets are designed so that resource-free steps get composed together, this obvious fact implies the following noteworthy result.

Corollary 25 *If c is resource-free, i.e.* $\mathtt{res}(c) = \{\}$*, each trace of c consists of single step. Such a command determines a state transformation, expressible as*

$$|c| = \{(\sigma, \sigma') \mid \exists(\tau, R, \tau') \in [\![c]\!].\ \sigma \subseteq \tau\ \&\ \sigma' = [\sigma \mid \tau']\}.$$

If in addition to this c has no free variables, the only possible non-empty trace for c is $([\,], [\,])$, and $|c|$ is either the identity function on states or the empty function.

For example, $|\mathbf{skip}; \mathbf{skip}| = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$ and $|\mathbf{while\ true\ do\ skip}| = \{\}$.

The next result validates a simple static law on commutativity of assignments.

Theorem 26 *Suppose $x \notin \mathtt{free}(e_2)$ and $y \notin \mathtt{free}(e_1)$. Then*

$$[\![x{:=}e_1; y{:=}e_2]\!] =$$
$$\{(\sigma_1 \cup \sigma_2, R, [x : v_1, y : v_2]) \mid (\sigma_1, v_1) \in [\![e_1]\!],\ (\sigma_2, v_2) \in [\![e_2]\!],\ \sigma_1 \Uparrow \sigma_2,$$
$$R^{-1}(x) = dom(\sigma_1)\ \&\ R^{-1}(y) = dom(\sigma_2)\}$$

and $[\![x{:=}e_1; y{:=}e_2]\!] = [\![y{:=}e_2; x{:=}e_1]\!] = [\![x{:=}e_1 \| y{:=}e_2]\!]$.

Theorem 27
Sequential composition is associative, with **skip** *as a unit.*

Theorem 28
Parallel composition is symmetric and associative, with **skip** *as a unit.*

We define what it means for a command to be race-free from a given state.

Definition 29 *c is race-free from σ iff* $\forall \alpha \in [\![c]\!].\forall \sigma'.\ (\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$ *implies* $\top \notin rge(\sigma'))$. *When this holds for all states σ we say that c is race-free.*

Theorem 30

- **skip** *and* $i{:=}e$ *are race-free.*
- *If c_1 and c_2 are race-free, so are $c_1; c_2$ and* **if** b **then** c_1 **else** c_2.
- *If c is race-free, so are* **while** b **do** c *and* **with** r **when** b **do** c.
- *If* $\mathtt{writes}(c_1) \cap \mathtt{free}(c_2) = \mathtt{writes}(c_2) \cap \mathtt{free}(c_1) = \{\}$, *and c_1 and c_2 are race-free, then $c_1 \| c_2$ is race-free.*

Corollary 31 *Every sequential program c is race-free.*

A resource context Γ is a set of entries of form $r(X)$, where r is a resource name and X is a finite set of identifiers, called a protection list. We say that c respects Γ iff every free occurrence in c of an identifier protected by r in Γ is inside a conditional critical region naming r. The next result shows that our semantics validates the Owicki-Gries static constraints on critical variables [9]. We let $\mathtt{owned}(\Gamma)$ be the set of identifiers occurring in the protection lists of Γ.

Theorem 32
Let c_1 and c_2 be race-free commands. If c_1 and c_2 respect Γ and

$$\texttt{writes}(c_1) \cap \texttt{free}(c_2) \subseteq \texttt{owned}(\Gamma) \;\&\; \texttt{writes}(c_2) \cap \texttt{free}(c_1) \subseteq \texttt{owned}(\Gamma)$$

then $c_1 \| c_2$ is race-free.

The static constraints built into this theorem statement require that each "critical variable" of $c_1 \| c_2$ must be "protected" by resources and must only appear inside critical regions naming the relevant resources. In Owicki-Gries logic a variable is deemed "critical" if it has a write occurrence in c_1 or c_2 and is read or written by the other. Our result confirms that these constraints ensure race-freedom. Again, although this is a fairly obvious result, it is comforting to see that our semantics furnishes a straightforward proof.

Safe partial correctness

Let p and q be boolean-valued expressions in which identifiers may occur free. (We do not allow resource names to appear free in such expressions.) Let $|p|$ be the set of states satisfying p. The *safe partial correctness* formula $\{p\}c\{q\}$ is *valid* iff all finite executions of c from a state satisfying p are race-free and end in a state satisfying q. We formalize this notion of validity as follows, using the trace semantics.

Definition 33 *$\{p\}c\{q\}$ is valid for all $\alpha \in [\![c]\!]$, and all states σ such that $dom(\sigma) \supseteq \texttt{free}(p, c, q)$, if $\sigma \in |p|$ and $\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$, then σ' is race-free and $\sigma' \in |q|$.*

Since validity is defined in terms of trace sets it is immediate that programs with the same trace sets satisfy the same assertions.

Theorem 34 *If $[\![c_1]\!] = [\![c_2]\!]$ then c_1 and c_2 satisfy the same assertions in all program contexts, i.e. for all p and q, and all contexts $C[-]$, $\{p\}C[c_1]\{q\}$ is valid iff $\{p\}C[c_2]\{q\}$ is valid.*

Corollary 35 *If $[\![c_1]\!] = [\![c_2]\!]$ then c_1 and c_2 satisfy the same assertions, i.e. for all p and q, $\{p\}c_1\{q\}$ is valid iff $\{p\}c_2\{q\}$ is valid.*

Observing the state

For a trace α, let $|\alpha|$ be the set of state sequences obtainable by executing α. When $\alpha = \lambda_1 \dots \lambda_n$ we have $|\alpha| = \{\sigma_1 \dots \sigma_n \mid \sigma_1 \stackrel{\lambda_1}{\Longrightarrow} \sigma_2 \cdots \sigma_{n-1} \stackrel{\lambda_n}{\Longrightarrow} \sigma_n\}$. When α is non-executable this set is obviously empty. We define the set $\mathcal{S}(c)$ of state sequences observable during executions of c, to be $\mathcal{S}(c) = \bigcup\{|\alpha| \mid \alpha \in [\![c]\!]\}$. This is observing the state at start and finish and at all synchronization points.

Theorem 36 *If $[\![c_1]\!] = [\![c_2]\!]$ then for all contexts $C[-]$, $\mathcal{S}(C[c_1]) = \mathcal{S}(C[c_2])$.*

This demonstrates that our semantics supports compositional program analysis.

3 Conclusions

Our more refined treatment of races yields a footprint trace semantics that supports compositional reasoning about safe partial correctness, tracking the variables whose values are immune to race conditions. The new model enjoys better abstraction properties than our earlier model [4]. We can pinpoint the differences by looking at the notions of semantic equivalence induced by the old semantics and the new. In the earlier model a race condition was a global catastrophe, leading to a special **abort** state. This earlier model gave the same meaning, namely

$$\{([x : v_1, y : v_2], [x : 1, y : 1]) \mid v_1, v_2 \in V\},$$

to (i) $x{:=}1; y{:=}x$ and (ii) $x{:=}1; y{:=}1$, and hence the same meaning, namely

$$\{([x : v_1, y : v_2], \mathbf{abort}) \mid v_1, v_2 \in V\},$$

to programs (i)$\|x{:=}2$ and (ii)$\|x{:=}2$. The new semantics distinguishes (i) from (ii), because they induce different flow relations: only in (i) is y dependent on x. (We made a similar comment in an example, earlier.) Moreover, this distinction is worth making, because when run in parallel with a program that (only) writes to x, for (i) the value of y gets tainted but for (ii) it does not. Thus there is a program context in which (i) and (ii) have different observable behavior. In particular, the new semantics distinguishes (correctly) between the programs (i)$\|x{:=}2$ and (ii)$\|x{:=}2$.

We refer to our semantics as "grainless". This appellation is accurate, as the steps occurring in the traces of programs are obtained by "mumbling" that abstracts away from irrelevant details. We do still assume that resource actions $lock(r)$ and $unlock(r)$ are atomic. Further, we only include feasible traces, so we obtain a more succinct trace set that subscribes roughly speaking to the slogan: fewer traces, shorter traces, bigger steps. Thus our semantics strives to avoid the combinatorial explosion inherent in interleaving traces, by working with smaller trace sets and shorter traces. Our more localized account of races allows a more liberal view of safe partial correctness. We could say that $\{p\}c\{q\}$ is valid iff for all states σ satisfying p, and all $\alpha \in [\![c]\!]$, if $\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$ then σ' satisfies q. Assuming that we define satisfaction so that $\sigma \in |p|$ implies $\forall i \in \mathtt{free}(p).\ \sigma(i) \neq \top$, this notion of validity does not imply absolute race-freedom of c from states satisfying the pre-condition, just that the state relevant to the post-condition is race-free. We plan to explore this idea further, perhaps leading to a new variant of concurrent separation logic that deals more flexibly with race conditions.

We dealt here with a simple shared-memory parallel language: no pointers or mutable state, no dynamic allocation of storage. We believe that the foundations laid in our prior development of *action trace* semantics for concurrent separation logic [3] and our earlier grainless model [4] can be adapted to combine the new approach to race-detection and tainting with mutable state, and we plan to investigate further. We plan to explore work using flow analysis in other settings, such as secure information flow and program analysis. Insights from related fields may help assess the scope and limitations of our work, and may suggest some improvements.

We have assumed that programs are executed on an architecture that provides sequential memory consistency [7]:

> . . . the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

This assumption leads naturally to the use of interleaving to represent parallel composition, as is common in denotational models of shared-memory dating back to Park [10]. We have not tried to deal with *weak memory* assumptions, supported by some modern concurrency architectures [1]. We argue that even in (perhaps especially in) dealing with more relaxed implementations it would be useful to have a semantics like ours, that yields a (machine-independent) characterization of race-free programs and predicts (semantically) to what extent a program's behavior is susceptible to uncertainty about the values of variables.

Acknowledgement

Stephan van Staden made a number of helpful suggestions for clarification and improvement. We thank the referees for pointers to interesting related work.

References

[1] S. V. Adve and K. Gharachorloo. *Shared Memory Consistency Models: A Tutorial*. IEEE Computer 29 (12): 66–76 (December 1996).

[2] S. Brookes. *Full abstraction for a shared-variable parallel language*. Proc. 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1993), 98–109. Journal version in: *Inf. Comp.*, vol 127(2):145-163, Academic Press, June 1996.

[3] S. Brookes. *A Semantics for Concurrent Separation Logic*. Invited paper, CONCUR 2004, Philippa Gardner and Nobuko Yoshida (Eds.), Springer LNCS 3170, London, August/September 2004, pp. 16-34.

[4] S. Brookes. *A Grainless Semantics for Parallel Programs with Shared Mutable State*. Proc. 21^{st} Conference on Mathematical Foundations of Programming Semantics (MFPS XXI). Birmingham, 17–21 May, 2005.

[5] E. W. Dijkstra. *Cooperating sequential processes*. In: **Programming Languages**, F. Genuys (editor), pp. 43-112. Academic Press, 1968.

[6] C.A.R. Hoare. *Towards a Theory of Parallel Programming*. In **Operating Systems Techniques**, C. A. R. Hoare and R. H. Perrott, editors, pp. 61-71, Academic Press, 1972.

[7] L. Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Trans. Comput. C-28,9 (Sept. 1979), pp. 690-691.

[8] P.W. O'Hearn. *Resources, Concurrency, and Local Reasoning*. Invited paper, CONCUR 2004, Philippa Gardner and Nobuko Yoshida (Eds.), Springer LNCS 3170, London, August/September 2004, pp. 49-67.

[9] S. Owicki and D. Gries. *Verifying properties of parallel programs: An axiomatic approach*, Comm. ACM. 19(5):279-285, May 1976.

[10] D. Park. *On the semantics of fair parallelism*. In: **Abstract Software Specifications**, Springer-Verlag LNCS vol. 86, 504–526, 1979.

[11] J. C. Reynolds. *Towards a Grainless Semantics for Shared-Variable Concurrency*. Proc. 24^{th} Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004), Chennai, India. Springer-Verlag, December 2004.