Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 155 (2006) 277–307

www.elsevier.com/locate/entcs

A Grainless Semantics for Parallel Programs with Shared Mutable Data

Stephen Brookes

Department of Computer Science
Carnegie Mellon University
Pittsburgh, USA

Abstract

We provide a new denotational semantic model, based on "footstep traces", for parallel programs which share mutable state. The structure of this model embodies a classic principle proposed by Dijkstra: processes should be treated independently, with interference occurring only at synchronization points. As a consequence the model makes fewer distinctions between programs than traditional trace models, which may help to mitigate the combinatorial explosion triggered by interleaving. For a sequential or synchronization-free program the footstep trace semantics is equivalent to a non-deterministic state transformation, so the new model supports "sequential" reasoning about synchronization-free code fragments. We show that footstep trace semantics is strictly more abstract than action trace semantics and suitable for compositional reasoning about race-freedom and partial correctness. The new model can be used to establish the soundness of concurrent separation logic. We include some example programs to facilitate comparison with earlier models, and we discuss briefly the relationship with a recent model by John Reynolds in which actions have discernible starts and finishes.

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

1 Outline

It is notoriously difficult to reason about parallel programs, because of the potential for interference or race conditions between concurrent threads. In traditional semantic models a process or thread denotes a set of traces, a trace is a sequence of atomic actions, and parallel composition is interpreted as fair interleaving. As is well known, this can lead to a combinatorial explosion

[1] Email: brookes@cs.cmu.edu

1571-0661
doi:10.1016/j.entcs.2005.11.060

when attempting to prove correctness of a parallel program, because of the number of possible interleavings to be considered.

If concurrent threads write to the same variable, without any guarantee of atomicity or exclusivity, the result may be unpredictable or dependent on implementation details beyond the control of the programmer. Traditional semantic models usually make a granularity assumption about the atomicity of certain primitives at the hardware level. The assumption that reads and writes to integer-valued variables are atomic leads to a trace semantics with larger atomic steps and fewer interleavings, but this assumption is not realistic in practice because an integer value may not fit inside a single machine word. On the other hand, the slightly more realistic assumption that word-level reads and writes are atomic will yield more interleavings, tending to exacerbate the combinatorial problem, and the results may depend on word size. Instead of relying on possibly unfounded expectations about hardware implementation we would prefer to be able safely to abstract away from granularity and word size while simultaneously reducing or avoiding the interleaving problem.

To an extent this approach is possible for simple shared-memory programs (i.e. shared-memory programs without pointers) designed in a sufficiently disciplined manner. For such programs the "critical" variables can be detected by a static, syntax-directed analysis; as usual, a variable is "critical" if it has a free write occurrence in one parallel component of the program and a free read or write occurrence in another. In the resource-based approach proposed by Hoare [13], Brinch Hansen [2,5] and others, the programmer must partition the critical variables among named resources, and access to these variables is only allowed inside a critical region for the relevant resource. Since resources are assumed to be implemented as semaphores, these design rules ensure mutually exclusive access to critical variables and disallow programs with "racy" behavior whose execution results might depend on granularity. Owicki and Gries introduced a Hoare-style logic that reflects this discipline, with rules for parallel composition, regions and resources [16]. This provides a methodology for safe reasoning about partial correctness without dependence on granularity.

Unfortunately this methodology is unsuitable for parallel programs that use pointers, because the possibility of aliasing renders purely static detection of race conditions impossible, and the Owicki-Gries proof system is unsound in the presence of aliasing. Recently, O'Hearn [14] has proposed an adaptation of the Owicki-Gries logic to handle pointers, incorporating ideas from separation logic [18]. The author has developed a semantic model (using *action traces*) and used it to prove soundness of this *concurrent separation logic* [7]. The action traces semantic model interprets parallel composition as a resource-

sensitive form of fair interleaving, and treats a potential race condition as a catastrophe, as suggested by John Reynolds. The soundness proof shows that a provable program is race-free.

The above logics [16,14] and semantics [7] were designed with an eye to a classic principle of concurrent program design, articulated by Dijkstra [11]:

> ... processes should be loosely connected; ... apart from the (rare) moments of explicit intercommunication, ... processes are to be regarded as completely independent of each other.

The action traces model supports the definition of a "local enabling" relation that embodies this principle, allowing a characterization of process behavior in an environment that interferes only via synchronization. However, this model is unnecessarily fine-grained, and contains traces that correspond to process executions in less well-behaved environments, allowing arbitrary interference. As a result the model distinguishes between some pairs of commands which satisfy exactly the same concurrent separation logic formulas. Indeed the action traces model supports compositional analysis of safety and liveness properties, in addition to partial and total correctness, because it retains information about the sequences of atomic state changes that occur in a computation. This level of detail is irrelevant for partial and total correctness.

In this paper we develop a more abstract semantics, tailored specifically for reasoning about race-free partial and total correctness. Instead of action traces built from resource actions and atomic state actions such as reads and writes to individual variables, we work with *footstep traces* built from resource actions and (compressed sequences of) state actions, representing the cumulative effect of a sequence of state changes executed without interference. We restrict the structure of traces to reflect the assumption that interference can only occur within a critical region. The typical trace will therefore consist of an alternating sequence of state actions and resource actions, with "external" state changes only occurring on synchronization, when a resource is acquired, so that the new semantic model truly embodies Dijkstra's principle.

This new semantics can be used instead of the action traces semantics of [7] to establish the soundness of concurrent separation logic, providing a more streamlined soundness proof based on a more succinct semantics. Moreover, the footstep trace set of a command is independent of any assumption about the granularity of atomic actions. For a well designed concurrent program whose synchronization points are few and far between, our model requires fewer interleavings, helping to mitigate the combinatorial explosion.

To facilitate comparison with action trace semantics [7], and to retain as much common ground with that model as is still relevant, our technical development of the footstep trace semantics will echo the main concepts and

definitions from the action trace setting, reformulating them as needed in terms of footsteps. (We also re-examine some of the program examples from [7] to highlight the advantages of the new semantics.)

Our footstep trace semantics has been influenced by ideas of John Reynolds, who recently developed a model with similar aims [19]. Reynolds' model also incorporates some ideas from action trace semantics, but there are significant differences in structure and design. In Reynolds' approach state actions such as reads and writes have a separate start and finish, leading to a trace model which makes more distinctions between processes than the footstep traces model. Reynolds then introduces an abstraction function on trace sets that ignores unnecessary details, such as the relative order of independent steps. In contrast, duration plays no rôle in our approach, and we coalesce adjacent footsteps to avoid order-based distinctions, so that we obtain a more abstract semantics by construction. For a command with no free resource names, such as a sequential program, or a concurrent program with no critical variables and therefore only "disjoint" parallelism and no need for critical regions, the footstep trace semantics degenerates into a form equivalent to a (non-deterministic) state transformation. Thus our semantics achieves one of the desirable properties mentioned by Reynolds [19].

2 Syntax

Our programming language combines shared-memory parallelism with pointer operations.

Definition 2.1 The syntax for *commands* (ranged over by c) is given by the following abstract grammar, in which r ranges over *resource names*, i over *identifiers*, e over *integer expressions*, b over *boolean expressions*, and E over *list expressions*:

$$\begin{aligned} c ::= {} & \mathbf{skip} \mid i{:=}e \mid i{:=}[e] \mid [e]{:=}e' \mid i{:=}\mathbf{cons}\,(E) \mid \mathbf{dispose}\,e \mid \\ & c_1; c_2 \mid c_1 \| c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \\ & \mathbf{local}\ i = e\ \mathbf{in}\ c \mid \mathbf{resource}\ r\ \mathbf{in}\ c \mid \mathbf{with}\ r\ \mathbf{when}\ b\ \mathbf{do}\ c \end{aligned}$$

We omit the syntax of expressions, which includes the usual arithmetic operations and boolean connectives. We assume that expressions are "pure", i.e. the value of an expression does not depend on the heap. A list expression E is a list $e_0, \ldots, e_n$ of integer expressions.

An *allocation* $i{:=}\mathbf{cons}(E)$ allocates a series of fresh heap cells; a *lookup* $i{:=}[e]$ reads a heap cell and assigns its contents to i; an *update* $[e]{:=}e'$ changes the contents of a heap cell; and a *disposal* $\mathbf{dispose}(e)$ de-allocates a heap cell.

A command of form **local** $i = e$ **in** c introduces a local variable named i, initialized to the value of e, with scope c.

A *resource block* **resource** r **in** c introduces a local resource name r, with scope c. A command of form **with** r **when** b **do** c is a *conditional critical region* for resource r. A process attempting to enter a region must wait until the resource is available, acquire the resource and evaluate b: if b is **true** the process executes c then releases the resource; if b is **false** the process releases the resource and waits to try again. A resource can only be held by one process at a time. We use the abbreviation **with** r **do** c when b is **true**.

Let $\mathtt{free}(c)$ be the set of identifiers occurring free in c, with similar notation for expressions. Let $\mathtt{reads}(c)$ be the set of identifiers having a free read occurrence in c, $\mathtt{writes}(c)$ be the set of identifiers having a free write occurrence in c. These sets may be defined by structural induction as usual.

Definition 2.2 Let $\mathtt{res}(c)$ be the set of resource names occurring free in c, defined by the obvious structural induction. In particular,

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

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

$$\mathtt{res}(\textbf{resource } r \textbf{ in } c) = \mathtt{res}(c) - \{r\}.$$

A command c is *resource-free* if $\mathtt{res}(c) = \{\}$.

3 States

A state is a partial function $\sigma : \mathtt{Var} \rightharpoonup V_{int}$, where $\mathtt{Var}$ is the disjoint union of **Ide** and V_{int}. We use V_{int} for the set of locations (or heap cells), and we make no distinction between locations and integers. **Ide** is the set of identifiers.

Let **St** be the set of states. We use σ as a meta-variable ranging over **St**. The state σ is *finite* if $\mathtt{dom}(\sigma)$ is a finite set. Note that $\mathtt{dom}(\sigma)$ is a subset of **Var**, possibly including both identifiers and values. [2]

We use conventional notation for states, such as $[v_1 : v_1', \ldots, v_k : v_k']$, where each $v_i \in \mathtt{Var}$ and each $v_i' \in V_{int}$. In particular $[\,]$ denotes the empty state. We write $[\sigma \mid v : v']$ for the state that agrees with σ except at v, which it maps to v'. For $X \subseteq \mathtt{Var}$ let $\sigma\rceil X = \{(v, v') \in \sigma \mid v \in X\}$ and $\sigma \backslash X = \{(v, v') \in \sigma \mid v \notin X\}$. We write $\sigma \smile \sigma'$ when $\sigma\rceil\mathtt{dom}(\sigma') = \sigma'\rceil\mathtt{dom}(\sigma)$, i.e. when the union $\sigma \cup \sigma'$ is a valid state. Let $\sigma \perp \sigma'$ when $\mathtt{dom}(\sigma) \cap \mathtt{dom}(\sigma') = \{\}$. Obviously if $\sigma \perp \sigma'$ then also $\sigma \smile \sigma'$.

[2] It would be equally reasonable to present the state as a pair (s, h) where $s : \textbf{Ide} \rightharpoonup V_{int}$ is the "store" and $h : V_{int} \rightharpoonup V_{int}$ is the "heap".

4 Actions

Actions are the building blocks of our semantic model. An action is either a "footstep", representing a piece of state change, or a resource action representing an attempt to acquire a resource, or the releasing of a resource. We include both "normal" footsteps and "abnormal" footsteps representing a runtime error or a non-terminating computation. We use the term "footstep" by analogy with the usage of "footprint" in related work such as [15,14]. Footsteps may involve reading and writing to variables, and allocation and disposal of heap cells.

Definition 4.1 Let λ range over actions, as described by the following abstract grammar, in which σ, σ' range over states, X ranges over $\mathcal{P}(\mathtt{Var})$, and r ranges over resource names:

$$\lambda ::= (\sigma, \sigma')_X \mid (\sigma, \bot)_X \mid (\sigma, \mathit{abort}) \mid \mathit{try}(r) \mid \mathit{acq}(r) \mid \mathit{rel}(r)$$

The actions dealing with resources are interpreted as in the earlier semantic models: $\mathit{try}(r)$ represents an unsuccessful attempt to acquire r, $\mathit{acq}(r)$ represents a successful acquisition, and $\mathit{rel}(r)$ releases the resource.

A (normal) footstep has the form $(\sigma, \sigma')_X$ where σ and σ' are finite states and X is a subset of $\{v \in \mathtt{dom}(\sigma) \cap \mathtt{dom}(\sigma') \mid \sigma(v) = \sigma'(v)\}$. This side-condition on X can be rephrased equivalently as $\sigma' \lceil X = \sigma \lceil X$. The set X contains the variables deemed to be "read-only" in the step, and σ' is to be regarded as all that is left of σ after the step, together with any new state introduced by allocations performed in the step. (Note that $X \subseteq \mathtt{Var}$, so X may include heap cells as well as identifiers.) Such a step represents the footprint of a command that "reads" σ and "writes" $\sigma' \backslash X$. The state portion $\sigma \lceil X$ is read-only, and survives the step unchanged.

A footstep of the form (σ, abort), where σ is a (finite) state, represents a runtime error such as a race condition, or an attempt to dispose a location that is not in use. Such steps will be treated as catastrophic, so there is no need to record a read-only set here.

Finally a footstep of the form $(\sigma, \bot)_X$, where σ is a state, represents a non-terminating computation in which the variables in X are read-only and the variables in $\mathtt{dom}(\sigma)$ are read/written or written. We include a read-only set in order to be able to predict race conditions accurately.

Definition 4.2 For a footstep $\lambda = (\sigma, \sigma')_X$ let $\mathtt{dom}(\lambda) = \mathtt{dom}(\sigma) \cup \mathtt{dom}(\sigma')$, and $\mathtt{mod}(\lambda) = \mathtt{dom}(\lambda) - X$; this is the set of variables which are modified by λ. We also define $\mathtt{allocates}(\lambda) = \mathtt{dom}(\sigma') - \mathtt{dom}(\sigma)$ and $\mathtt{disposals}(\lambda) = \mathtt{dom}(\sigma) - \mathtt{dom}(\sigma')$.

For a non-terminating footstep $\lambda = (\sigma, \bot)_X$ we let $\mathtt{mod}(\lambda) = \mathtt{dom}(\sigma) - X$.

Again this represents the set of identifiers assigned to or heap cells updated by the step. We need this set in order to keep a proper account of race conditions. There is no need, however, to define $\mathtt{mod}(\lambda)$ when λ is an abort step.

Example 4.3 Let $v \in \mathbf{Var}$ and $v', v'' \in V_{int}$.

- The footstep $([v : v'], [v : v''])_{\{v\}}$ is only well-formed if $v' = v''$, in which case it represents a "read" of v with result v'.
- The footstep $([v : v'], [v : v''])_{\{\}}$ is always well-formed, and represents a "read-write" of v; even if $v' = v''$ we distinguish this from a pure read.
- A footstep of form $([v' : v''], [\,])_{\{\}}$ represents a disposal of v'.
- A footstep of form $([\,], [v' : v''])_{\{\}}$ represents an allocation of v'.

Example 4.4 The footstep $([x : 0, y : 1, z : 2, 1 : 42], [x : 1, y : 1, z : 99, 99 : 0])_{\{y\}}$ represents a state change that updates the value of x, disposes the heap cell denoted by y, and sets z to a fresh heap cell initialized to contain 0. Notice that the variable y is not itself updated, as indicated by the read-only set annotation.

5 Traces

We build traces by concatenating finite or infinite sequences of actions. We will use α, β to range over traces, and let **Tr** be the set of traces.

We need to take care when determining which sequences of actions make sense, bearing in mind our assumptions about the interactions between a command and its environment. Each resource is held by at most one process at all times, so the acquire and release actions for each resource name along a trace will alternate (as in the action traces model [7]). Storage management is assumed to be handled globally: each call to the storage allocator yields a heap cell not in use by any process. Finally, in line with the Dijkstra principle, we will build traces in which adjacent footsteps are always combined into a single step. This is a radical departure from the action traces model and we make this design choice so that we ignore intermediate states irrelevant for reasoning about race-freedom and partial correctness. [3]

Catenable actions

We characterize the conditions under which one footstep can be executed after another: λ_1 can follow λ_0 if the state needed for λ_1 is consistent with the effect

[3] This coalescing of adjacent steps is reminiscent of *mumbling* [6], but we work exclusively with "maximally mumbled" traces rather than including all possible mumblings of a given sequence, since this results in a more succinct model.

of λ_0, and λ_1 does not allocate any location that was allocated by λ_0. When this holds we also say that the sequence $\lambda_0\lambda_1$ is *catenable*. Accordingly, we will build traces by concatenating sequences of actions which obey this constraint.

Definition 5.1 Let $\lambda_0 = (\sigma_0, \sigma_0')_{X_0}$ and $\lambda_1 = (\sigma_1, \sigma_1')_{X_1}$ be footsteps. We say that λ_1 *follows* λ_0, written $\lambda_0 \asymp \lambda_1$, if

- $\sigma_0' \smile \sigma_1$, i.e. $\sigma_0' \rceil \mathtt{dom}(\sigma_1) = \sigma_1 \rceil \mathtt{dom}(\sigma_0')$
- $\sigma_0 \smile (\sigma_1 - \sigma_0')$
- $\mathtt{allocates}(\lambda_1) \cap \mathtt{dom}(\sigma_0') = \{\}$, i.e. $\mathtt{dom}(\sigma_0') \cap (\mathtt{dom}(\sigma_1') - \mathtt{dom}(\sigma_1)) = \{\}$.

The first two conditions guarantee that the state resulting from λ_0 is consistent with the state needed for λ_1. The third requirement reflects the assumption that a call to the storage allocator yields truly fresh storage.

Obviously this is not a symmetric relationship: $\lambda_0 \asymp \lambda_1$ does not generally imply that $\lambda_1 \asymp \lambda_0$.

The read-only sets play no rôle in determining the follows relation. They do contribute, however, in defining the combined effect of catenable actions. When λ_1 follows λ_0 we can construct a single footstep, denoted $\lambda_0; \lambda_1$, that represents the combined effects of the successive steps.

Definition 5.2 Let $\lambda_0 = (\sigma_0, \sigma_0')_{X_0}$ and $\lambda_1 = (\sigma_1, \sigma_1')_{X_1}$ be footsteps such that $\lambda_0 \asymp \lambda_1$. We define $\lambda_0; \lambda_1$ to be the following footstep:

- if $\mathtt{disposals}(\lambda_0) \cap \mathtt{dom}(\sigma_1) = \{\}$ we let

$$(\sigma_0, \sigma_0')_{X_0}; (\sigma_1, \sigma_1')_{X_1} = (\sigma_0 \cup (\sigma_1 - \sigma_0'),\ \sigma_1' \cup (\sigma_0' - \sigma_1))_X$$

  where $X = (X_0 - \mathtt{mod}(\lambda_1)) \cup (X_1 - \mathtt{mod}(\lambda_0))$.
- if $\mathtt{disposals}(\lambda_0) \cap \mathtt{dom}(\sigma_1) \neq \{\}$ then λ_0 disposes some cell needed by λ_1, so we let

$$(\sigma_0, \sigma_0')_{X_0}; (\sigma_1, \sigma_1')_{X_1} = (\sigma_0 \cup (\sigma_1 - \sigma_0'),\ \mathit{abort}).$$

We extend the catenability relation $\asymp$ and the sequencing operator $\lambda_0; \lambda_1$ to more general pairs of actions in the obvious way. For abort steps we make the following definitions:

- $(\sigma_0, \sigma_0')_{X_0} \asymp (\sigma_1, \mathit{abort})$ if $\sigma_0' \smile \sigma_1$ and $(\sigma_1 - \sigma_0') \smile \sigma_0$
  $(\sigma_0, \sigma_0')_{X_0}; (\sigma_1, \mathit{abort}) = (\sigma_0 \cup (\sigma_1 - \sigma_0'), \mathit{abort})$
- $(\sigma_0, \mathit{abort}) \asymp \lambda$ for all actions λ
  $(\sigma_0, \mathit{abort}); \lambda = (\sigma_0, \mathit{abort})$

The corresponding definitions involving a non-terminating step are similar:

- Let $\lambda_0 = (\sigma_0, \sigma_0')_{X_0}$ and $\lambda_1 = (\sigma_1, \perp)_{X_1}$.
  $(\sigma_0, \sigma_0')_{X_0} \asymp (\sigma_1, \perp)_{X_1}$ if $\sigma_0' \smile \sigma_1$ and $(\sigma_1 - \sigma_0') \smile \sigma_0$

$(\sigma_0, \sigma_0')_{X_0}; (\sigma_1, \bot)_{X_1} = (\sigma_0 \cup (\sigma_1 - \sigma_0'), \bot)_X$,
where $X = (X_0 - \mathtt{mod}(\lambda_1)) \cup (X_1 - \mathtt{mod}(\lambda_0))$.

- $(\sigma_0, \bot)_{X_0} \asymp \lambda$ for all actions λ
  $(\sigma_0, \bot)_{X_0}; \lambda = (\sigma_0, \bot)_{X_0}$.

We allow any action λ to be followed by (or to follow) any resource action, so that for all resource names r, $\lambda \asymp try(r)$, $\lambda \asymp acq(r)$, $\lambda \asymp rel(r)$, $try(r) \asymp \lambda$, and so on.

Remark 5.3 Let δ be the footstep $([\,], [\,])_{\{\}}$. It is easy to check that, for all footsteps λ_0 and λ_1, we have $\lambda_0 \asymp \delta$, $\delta \asymp \lambda_1$, and $\lambda_0; \delta = \lambda_0$, $\delta; \lambda_1 = \lambda_1$.

The following examples illustrate the above definitions, showing how the read-only information of successive steps gets combined, and motivating the carefully chosen side conditions in the previous development.

Example 5.4 Consecutive reads of the same variable are catenable provided they yield the same value; the result is again a read. More formally:
$([x : v], [x : v])_{\{x\}} \asymp ([x : v'], [x : v'])_{\{x\}}$ if and only if $v = v'$
$([x : v], [x : v])_{\{x\}}; ([x : v], [x : v])_{\{x\}} = ([x : v], [x : v])_{\{x\}}$

Example 5.5 A read followed by a write to the same variable is sensible when the read yields the same value as is used to start the write; the result is a write.
$([x : v], [x : v])_{\{x\}} \asymp ([x : v'], [x : v''])_{\{\}}$ if and only if $v = v'$
$([x : v], [x : v])_{\{x\}}; ([x : v], [x : v'])_{\{\}} = ([x : v], [x : v'])_{\{\}}$

Example 5.6 Reads of distinct variables can be concatenated; their effects commute, and the result is a single read-only step.
$([x : v], [x : v])_{\{x\}} \asymp ([y : v'], [y : v'])_{\{y\}}$ when $x \neq y$
$([x : v], [x : v])_{\{x\}}; ([y : v'], [y : v'])_{\{y\}} = ([x : v, y : v'], [x : v, y : v'])_{\{x,y\}}$

Example 5.7 Writes to distinct variables can be concatenated; their effects commute; the result is a single action representing the combined writes.
$([x : v_0], [x : v_0'])_{\{\}} \asymp ([y : v_1], [y : v_1'])_{\{\}}$ when $x \neq y$
$([x : v_0], [x : v_0'])_{\{\}}; ([y : v_1], [y : v_1'])_{\{\}} = ([x : v_0, y : v_1], [x : v_0', y : v_1'])_{\{\}}$
$([y : v_1], [y : v_1'])_{\{\}}; ([x : v_0], [x : v_0'])_{\{\}} = ([x : v_0, y : v_1], [x : v_0', y : v_1'])_{\{\}}$

Example 5.8 A disposal of one heap cell can always be followed by a disposal of another cell. An attempt to dispose the same cell twice in succession causes an error.
$([v_0 : v_0'], [\,])_{\{\}} \asymp ([v_1 : v_1'], [\,])_{\{\}}$ if $v_0 \neq v_1$ or $(v_0 = v_1$ & $v_0' = v_1')$.
$([v_0 : v_0'], [\,])_{\{\}}; ([v_1, v_1'])_{\{\}} = ([v_0 : v_0', v_1 : v_1'], [\,]))_{\{\}}$ if $v_0 \neq v_1$
$([v : v'], [\,])_{\{\}}; ([v : v'], [\,])_{\{\}} = ([v : v'], \mathit{abort})$

Note that in the erroneous case we have $([v : v'], [\,])_{\{\}} \asymp ([v : v'], [\,])_{\{\}}$, so that the actions are catenable but their concatenation represents a runtime error.

Example 5.9 Storage allocation is assumed to be managed globally, so successive calls to the allocator always return distinct heap cells; therefore we do not need to allow consecutive allocations of the same heap cell. Indeed, according to the above definitions we have $([\,], [v : v'])_{\{\}} \not\asymp ([\,], [v : v''])_{\{\}}$.

Example 5.10 A step that allocates a heap cell v can follow any step in which v was not read, written, or allocated, again because the storage manager always allocates "fresh" cells. We have $(\sigma, \sigma')_X \asymp ([\,], [v : v'])_{\{\}}$ if $v \notin \mathtt{dom}(\sigma')$, in which case $(\sigma, \sigma')_X ; ([\,], [v : v'])_{\{\}} = (\sigma, \sigma' \cup [v : v'])_{X \setminus v}$.

Catenable sequences

If $\lambda_0 \asymp \lambda_1$ and $\lambda_1 \asymp \lambda_2$, it does not necessarily follow that $(\lambda_0; \lambda_1) \asymp \lambda_2$. For example, consider the following footsteps:

$$\lambda_0 = ([v : 0], [\,])_{\{\}} \quad \lambda_1 = ([y : 0], [y : 0])_{\{\}} \quad \lambda_2 = ([v : 1], [\,])_{\{\}}.$$

We have $\lambda_0 \asymp \lambda_1$ and $\lambda_1 \asymp \lambda_2$, but $\lambda_0; \lambda_1 = ([v : 0, y : 0], [y : 0])_{\{\}}$, so that $(\lambda_0; \lambda_1) \not\asymp \lambda_2$. Indeed, the sequence of footsteps $\lambda_0\lambda_1\lambda_2$ clearly cannot be executed without the help of interference, because of the discrepancy in the value of v between the first and third steps. We should not, therefore, regard this sequence of actions as catenable.

In contrast, in all cases where a sequence of footsteps $\lambda_0\lambda_1\lambda_2$ is executable without interference we will have $\lambda_0 \asymp \lambda_1$ and $(\lambda_0; \lambda_1) \asymp \lambda_2$, $\lambda_1 \asymp \lambda_2$ and $\lambda_0 \asymp (\lambda_1; \lambda_2)$, and $(\lambda_0; \lambda_1); \lambda_2 = \lambda_0; (\lambda_1; \lambda_2)$.

We want to streamline our semantics so that we only include traces that reflect our underlying assumptions about interference, only allowing external interference through synchronization. We therefore extend the catenability relation to finite traces and actions as follows, using the notation $\beta \asymp \lambda$ when action λ can follow trace β. We extend the concatenation operation analogously, so that for a finite trace β and an action λ such that $\beta \asymp \lambda$ we obtain a definition of the trace $\beta; \lambda$. The definition uses case analysis on the final action of the trace. For sequences of footsteps the definitions reflect the above remarks concerning catenability.

Definition 5.11 For all finite traces α, resource names r, and footsteps λ, λ'

$$\begin{aligned}
\alpha\lambda &\asymp \lambda' && \text{iff } \lambda \asymp \lambda' \\
\alpha\, try(r) &\asymp \lambda' && \text{iff } \alpha \asymp \lambda' \\
\alpha\, acq(r) &\asymp \lambda' && \text{always} \\
\alpha\, rel(r) &\asymp \lambda' && \text{iff } \alpha \asymp \lambda'
\end{aligned}$$

Note that the step following a resource acquisition is not constrained here, allowing for the possibility of interference at synchronization.

Definition 5.12 For all finite traces α, resource names r, and footsteps λ, λ'

$$\begin{aligned}
(\alpha\lambda);\lambda' &= \alpha(\lambda;\lambda') && \text{when } \lambda \asymp \lambda' \\
(\alpha\, try(r));\lambda &= \alpha\, try(r)\,\lambda' && \text{when } \alpha \asymp \lambda' \\
(\alpha\, acq(r));\lambda' &= \alpha\, acq(r)\,\lambda' && \text{always} \\
(\alpha\, rel(r));\lambda' &= \alpha\, rel(r)\,\lambda' && \text{when } \alpha \asymp \lambda'
\end{aligned}$$

We can now give a formal characterization of the catenable finite sequences of actions, and for each such sequence α we can specify the trace $cat(\alpha)$ obtained by concatenating its steps.

Definition 5.13 A single footstep λ is catenable, and $cat(\lambda) = \lambda$. A sequence of form $\beta\lambda$ is catenable if and only if β is catenable and $cat(\beta) \asymp \lambda$, in which case we let $cat(\beta\lambda) = cat(\beta);\lambda$.

When α is the sequence $\lambda_0 \ldots \lambda_n$ and is catenable, we may use the notation $\lambda_0;\ldots;\lambda_n$ for $cat(\alpha)$. The notion of catenability extends to an infinite sequence of actions in the obvious way: an infinite sequence is catenable if and only if each of its finite prefixes is catenable. However, the result of concatenating an infinite sequence of footsteps $\lambda_n = (\sigma_n, \sigma'_n)_{X_n}$ needs to be defined carefully. There are two cases:

- if for each n we have $\lambda_0;\ldots;\lambda_n = (\tau_n, \tau'_n)_{Y_n}$ then the infinite sequence represents a non-terminating computation that will be enabled by the state $\bigcup_{n=0}^{\infty} \tau_n$. The read-only variables for this entire computation can be taken to be $Y = \{v \in \bigcup_{n=0}^{\infty} X_n \mid \forall n.\ (v \in X_n \vee v \notin \mathtt{dom}(\lambda_n))\}$. We therefore define $\lambda_0;\ldots;\lambda_n;\ldots = (\bigcup_{n=0}^{\infty} \tau_n, \bot)_Y$.
- if for some n we have $\lambda_0;\ldots;\lambda_n = (\tau_n, abort)$ then this will also be the case for all larger n, and we define $\lambda_0;\ldots;\lambda_n;\ldots$ to be $(\tau_n, abort)$ also.

The state appearing in a footstep constructed by concatenating an infinite sequence of steps may well be infinite, even if all of the individual steps λ_n

involve finite pieces of state. This has a natural computational interpretation: the process needs to access an infinite portion of the heap during its execution. Of course this can only happen in an infinite computation, since each finite prefix of the computation trace affects only a finite portion of the state. The following example exhibits this behavior.

Example 5.14 Consider the sequence of steps $\lambda_n = ([x : n, n : 0], [x : n + 1])_{\{\}}$, for $n \geq 0$; we have, using the above definition,

$$\lambda_0; \lambda_1; \ldots; \lambda_n; \ldots = ([x : 0, 0 : 0, 1 : 0, \ldots, n : 0, \ldots], \bot)_{\{\}}.$$

This footstep represents a possible behavior of the command

$$\textbf{while true do } (\textbf{dispose}(x); x{:=}x + 1).$$

It is easy to extend the notion of catenability, and the concatenation operator, to pairs of traces. When α and β are traces we write $\alpha \asymp \beta$ to mean that β follows α, and we write $\alpha; \beta$ for the trace obtained by concatenating them. [4] We then define concatenation and iteration for sets of traces in the usual way, with adjustments so as to include only catenable cases. For trace sets T_0, T_1 and T, we define:

- $T_0; T_1 = \{\alpha_0; \alpha_1 \mid \alpha_0 \in T_0 \ \& \ \alpha_1 \in T_1 \ \& \ \alpha_0 \asymp \alpha_1\}$.
- $T^* = \bigcup_{n=0}^{\infty} T^n$, where $T^0 = \{\delta\}$ and $T^{n+1} = T^n; T$ for $n \geq 0$.
- $T^\omega = \{\alpha_0; \alpha_1; \ldots \alpha_n; \ldots \mid \forall n \geq 0.\ \alpha_n \in T \ \& \ \alpha_0; \ldots; \alpha_n \asymp \alpha_{n+1}\}$.

Parallel composition

The behavior of a process running in parallel with other threads depends on the resources held by the process and is constrained by the resources being held by its environment. These sets of resources start empty and will always stay disjoint, given our assumptions about the implementation of resources via semaphores. Accordingly we define for each action λ a *resource enabling* relation $(A_1, A_2) \xrightarrow{\lambda} (A_1', A_2)$ on disjoint pairs of resource sets, to specify when a process holding resources A_1, in an environment that holds A_2, can perform this action, and the action's effect on resources. Although the special case where A_1 and A_2 are empty deserves special attention, since it corresponds to the typical initial assumptions, the general case has a simpler inductive formulation.

Definition 5.15 Let A_1 and A_2 be disjoint sets of resource names. The

[4] Note that if α is catenable, so is $cat(\alpha)$, and $cat(cat(\alpha)) = cat(\alpha)$. If $\alpha \asymp \beta$ and α and β are catenable, so is $\alpha\beta$, and $cat(\alpha\beta) = cat(\alpha); cat(\beta)$.

resource enabling relations $\xrightarrow{\lambda}$ are given by:

$$\begin{array}{ll}
(A_1, A_2) \xrightarrow{try(r)} (A_1, A_2) & \text{always} \\
(A_1, A_2) \xrightarrow{acq(r)} (A_1 \cup \{r\}, A_2) & \text{iff } r \notin A_1 \cup A_2 \\
(A_1, A_2) \xrightarrow{rel(r)} (A_1 - \{r\}, A_2) & \text{iff } r \in A_1 \\
(A_1, A_2) \xrightarrow{\lambda} (A_1, A_2) & \text{always, if } \lambda \text{ is not a resource action}
\end{array}$$

This generalizes in the obvious way to a sequence of actions, and we write $(A_1, A_2) \xrightarrow{\alpha} \cdot$ to indicate that a process holding resources A_1 in an environment holding A_2 can perform the trace α.

Definition 5.16 Let $\lambda_0 = (\sigma_0, \sigma_0')_{X_0}$ and $\lambda_1 = (\sigma_1, \sigma_1')_{X_1}$. We say that λ_0 and λ_1 interfere, written $\lambda_0 \bowtie \lambda_1$, if

- $\sigma_0 \smile \sigma_1$, i.e. $\sigma_0 \rceil \mathtt{dom}(\sigma_1) = \sigma_1 \rceil \mathtt{dom}(\sigma_0)$
- $\mathtt{dom}(\sigma_0) \cap \mathtt{dom}(\sigma_1) \not\subseteq X_0 \cap X_1$

When $\lambda_0 \bowtie \lambda_1$ define $\lambda_0 \bullet \lambda_1 = (\sigma_0 \cup \sigma_1, \mathit{abort})$.

If the overlap is on variables which are read-only for both footsteps, there is no race so we do not count this as an interfering case.

For abnormal footsteps we characterize the interfering cases as follows:

Definition 5.17 When λ_0 is $(\sigma_0, \mathit{abort})$ and/or λ_1 is $(\sigma_1, \mathit{abort})$ we specify that $\lambda_0 \bowtie \lambda_1$ holds if $\sigma_0 \smile \sigma_1$. And if λ_0 is $(\sigma_0, \perp)_{X_0}$ and λ_1 is $(\sigma_1, \sigma_1')_{X_1}$ we specify that $\lambda_0 \bowtie \lambda_1$ holds if $\sigma_0 \smile \sigma_1$ and $\mathtt{dom}(\sigma_0) \cap \mathtt{dom}(\sigma_1) \not\subseteq X_0 \cap X_1$. The cases for $(\sigma_0, \perp)_{X_0}$ and $(\sigma_1, \perp)_{X_1}$, and for $(\sigma_0, \perp)_{X_0}$ and $(\sigma_1, \mathit{abort})$, are analogous.

We now define, for each pair (A_0, A_1) of disjoint resource sets and each pair (α_0, α_1) of action sequences, the set $\alpha_{0A_0} \|_{A_1} \alpha_1$ of all *mutex fairmerges* of α_0 using A_0 with α_1 using A_1. This is the set of all fair interleavings of α_0 with α_1 in which each step respects the resource constraints and a race condition causes an error. The definition for finite traces is inductive in the product of the lengths of the traces, and we include the empty sequence ϵ, to allow a simpler formulation: the base case is when one of the traces is empty. Note that the set $\alpha_{0A_0} \|_{A_1} \alpha_1$ may be empty: for instance when α_0 is the empty sequence but α_1 is not permitted because it needs a resource belonging to A_0. Again, although the case where both resource sets is empty plays a special role in the sequel, the more general case has a naturally inductive formulation.

Definition 5.18 For finite traces α and β, and disjoint resource sets A_0, A_1,

the set of traces $\alpha_{A_0}\|_{A_1}\beta$ is given inductively by:

$$\alpha_{A_0}\|_{A_1}\epsilon = \{\alpha \mid (A_0, A_1) \xrightarrow{\alpha} \cdot\}$$
$$\epsilon_{A_0}\|_{A_1}\beta = \{\beta \mid (A_1, A_0) \xrightarrow{\beta} \cdot\}$$
$$(\lambda_0\alpha_0)_{A_0}\|_{A_1}(\lambda_1\alpha_1)$$
$$= \{\lambda_0 \bullet \lambda_1\} \quad \text{if } \lambda_0 \bowtie \lambda_1$$
$$= \{\lambda_0; \beta \mid (A_0, A_1) \xrightarrow{\lambda_0} (A_0', A_1) \ \&\ \beta \in \alpha_{0A_0'}\|_{A_1}(\lambda_1\alpha_1) \ \&\ \lambda_0 \asymp \beta\}$$
$$\cup \{\lambda_1; \beta \mid (A_1, A_0) \xrightarrow{\lambda_1} (A_1', A_0) \ \&\ \beta \in (\lambda_0\alpha_0)_{A_0}\|_{A_1'}\alpha_1 \ \&\ \lambda_1 \asymp \beta\}$$
$$\text{if } \neg(\lambda_0 \bowtie \lambda_1)$$

Again we remark that the definition of parallel composition builds in the resource constraints, only includes catenable cases, and combines adjacent footsteps. The definition of the set of mutex fairmerges extends to infinite traces in the usual way [17,7,6].

The fairmerge definition for pairs of traces extends in the obvious pointwise way to sets of traces: when T_0 and T_1 are sets of traces we let $T_{0A_0}\|_{A_1}T_1$ be the union of the sets $\alpha_{0A_0}\|_{A_1}\alpha_1$ as α_0 ranges over T_0 and α_1 ranges over T_1. We will be mainly concerned with the case where $A_0 = A_1 = \{\}$, representing what happens when two processes start up with no initial resources. Indeed, we may abbreviate $T_{0\{\}}\|_{\{\}}T_1$ as $T_0\|T_1$, omitting the empty subscripts.

6 Semantics of expressions

Let $Tag = \mathcal{P}(\mathbf{Var})$ be the set of *read-only sets* used to annotate footsteps. Recall that **Var** is the union of **Ide** and V_{int}. For integer expressions e we will define a footstep trace set $[\![e]\!] \subseteq (\mathbf{St} \times V_{int})_{Tag}$. Similarly for boolean expressions we will define a footstep trace set $[\![b]\!] \subseteq (\mathbf{St} \times V_{bool})_{Tag}$. The footstep trace set $[\![e]\!]$ of an integer expression will contain entries of the form $(\sigma, v)_X$ where σ is a finite piece of state in which e evaluates to the integer value v, and X is the set of variables whose values are read during the evaluation.

We can also give a semantics for list expressions E along similar lines, so that the value of a list expression is a list of integer values: $[\![E]\!] \subseteq (\mathbf{St} \times V_{int}^*)_{Tag}$. We omit the details, which are straightforward. [5]

Our semantic clauses for expressions can be obtained from the corresponding clauses in the Reynolds semantics [19] by inserting and propagating the relevant information about read-only variables.

[5] It would also be easy to incorporate into this semantic model "impure" expressions such as $[e]$, whose value depends on the heap.

Definition 6.1 The semantic functions

$$[\![-]\!] : \mathbf{Exp}_{int} \to \mathcal{P}((\mathbf{St} \times V_{int})_{Tag})$$
$$[\![-]\!] : \mathbf{Exp}_{bool} \to \mathcal{P}((\mathbf{St} \times V_{bool})_{Tag})$$

are defined inductively by the following clauses:

$$\begin{aligned}
&[\![n]\!] = \{([\,], n)_{\{\}}\}\\
&[\![i]\!] = \{([i : v], v)_{\{i\}} \mid v \in V_{int}\}\\
&[\![e_1 + e_2]\!] = \{(\sigma_1 \cup \sigma_2, v_1 + v_2)_{X_1 \cup X_2} \mid\\
&\qquad\qquad (\sigma_1, v_1)_{X_1} \in [\![e_1]\!] \;\&\; (\sigma_2, v_2)_{X_2} \in [\![e_2]\!] \;\&\; \sigma_1 \smile \sigma_2\}\\
&[\![\mathbf{true}]\!] = \{([\,], \mathit{true})_{\{\}}\}\\
&[\![e_1 = e_2]\!] = \{(\sigma_1 \cup \sigma_2, v_1 = v_2)_{X_1 \cup X_2} \mid\\
&\qquad\qquad (\sigma_1, v_1)_{X_1} \in [\![e_1]\!] \;\&\; (\sigma_2, v_2)_{X_2} \in [\![e_2]\!] \;\&\; \sigma_1 \smile \sigma_2\}\\
&[\![b_1 \;\mathbf{and}\; b_2]\!] =\\
&\quad \{(\sigma_1, \mathit{false})_{X_1} \mid (\sigma_1, \mathit{false})_{X_1} \in [\![b_1]\!]\}\\
&\quad \cup \{(\sigma_1 \cup \sigma_2, t)_{X_1 \cup X_2} \mid (\sigma_1, \mathit{true})_{X_1} \in [\![b_1]\!] \;\&\; (\sigma_2, t)_{X_2} \in [\![b_2]\!] \;\&\; \sigma_1 \smile \sigma_2\}\\
&[\![\mathbf{if}\; b \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2]\!] =\\
&\quad \{(\sigma_0 \cup \sigma_1, v_1)_{X_0 \cup X_1} \mid (\sigma_0, \mathit{true})_{X_0} \in [\![b]\!] \;\&\; (\sigma_1, v_1)_{X_1} \in [\![e_1]\!] \;\&\; \sigma_0 \smile \sigma_1\}\\
&\quad \cup \{(\sigma_0 \cup \sigma_2, v_2)_{X_0 \cup X_2} \mid (\sigma_0, \mathit{false})_{X_0} \in [\![b]\!] \;\&\; (\sigma_2, v_2)_{X_2} \in [\![e_2]\!] \;\&\; \sigma_0 \smile \sigma_2\}
\end{aligned}$$

Here we have specified that a conjunction b_1 **and** b_2 is evaluated using the usual short-circuit; this detail is not crucial in what follows, and the ensuing development can be modified as needed if we use another interpretation such as strict, left-to-right evaluation.

There is no need to use a more elaborate semantic structure allowing for the possibility of interference during expression evaluation, since the model is intended only to describe executions in an environment that obeys the Dijkstra principle: interference is assumed to occur only on synchronization. It is therefore safe to ignore intermediate states. We have also assumed in the above definitions that expression evaluation always terminates, so that we did not need to include footsteps of the form $(\sigma, \perp)_X$ to express non-terminating cases. It is straightforward to extend the semantic definitions in this manner to include non-terminating expressions.

Example 6.2 To evaluate $x + y$ we need a value for x and a value for y:

$$[\![x + y]\!] = \{([x : v, y : v'], v + v')_{\{x,y\}} \mid v, v' \in V_{int}\}$$

Similarly, to evaluate $x + x$ we need (only) a value for x:

$$[\![x + x]\!] = \{([x : v], 2v)_{\{x\}} \mid v \in V_{int}\}.$$

Note that $[\![x + y]\!] = [\![y + x]\!]$, so that this expression semantics abstracts away from evaluation order.

A boolean expression gives rise to a pair of sets of single-footstep traces, in the obvious way: we distinguish the traces describing "true" cases from the traces describing "false" cases.

Definition 6.3 For a boolean expression b, let $[\![b]\!]_{true}, [\![b]\!]_{false} \subseteq \mathbf{Tr}$ be given by

$$[\![b]\!]_{true} = \{(\sigma,\sigma)_X \mid (\sigma, true)_X \in [\![b]\!]\}$$
$$[\![b]\!]_{false} = \{(\sigma,\sigma)_X \mid (\sigma, false)_X \in [\![b]\!]\}$$

Example 6.4

$$[\![\mathbf{true}]\!]_{true} = \{([\,],[\,])_{\{\}}\} = \{\delta\}$$
$$[\![\mathbf{true}]\!]_{false} = \{\}$$

Example 6.5 Let b be the expression $(x = y)\,\mathbf{and}\,(y = z)$.

$$[\![b]\!]_{false} = \{[x:v, y:v'], [x:v, y:v'])_{\{x,y\}} \mid v \neq v'\}$$
$$\cup\ \{([x:v, y:v, z:v''], [x:v, y:v, z:v''])_{\{x,y,z\}} \mid v \neq v''\}$$
$$[\![b]\!]_{true} = \{([x:v, y:v, z:v], [x:v, y:v, z:v])_{\{x,y,z\}} \mid v \in V_{int}\}$$

7 Semantics of commands

For each command c we define the footstep trace set $[\![c]\!] \subseteq \mathbf{Tr}$, by structural induction. The definition ensures that for each c, every trace in $[\![c]\!]$ is obtained from a catenable sequence of actions, so the trace set specifies the program's behavior in a loosely connected environment.

Definition 7.1 The semantic function $[\![-]\!] : \mathbf{Com} \to \mathcal{P}(\mathbf{Tr})$ is defined induc-

tively by the following clauses:

$$
\begin{aligned}
&[\![\mathbf{skip}]\!] = \{([\,],[\,])_{\{\}}\}\\
&[\![i{:=}e]\!] = \{(\sigma \cup [i:v], [\sigma \mid i:v'])_{X\setminus i} \mid (\sigma,v')_X \in [\![e]\!] \;\&\; \sigma \smile [i:v]\}\\
&[\![[e]{:=}e']\!] = \{(\sigma \cup \sigma' \cup [v:v_0], [\sigma \cup \sigma' \mid v:v'])_{(X\cup X')\setminus v} \mid\\
&\qquad (\sigma,v)_X \in [\![e]\!] \;\&\; (\sigma',v')_{X'} \in [\![e']\!] \;\&\; \sigma \smile \sigma' \;\&\; (\sigma\cup\sigma') \smile [v;v_0]\}\\
&[\![i{:=}[e]]\!] = \{(\sigma \cup [i:v_0, v:v'], [\sigma \mid i:v', v:v'])_{(X\cup\{v\})\setminus i} \mid\\
&\qquad (\sigma,v)_X \in [\![e]\!] \;\&\; \sigma \smile [i:v_0, v:v']\}\\
&[\![i{:=}\mathbf{cons}(E)]\!] = \{(\sigma \cup [i:v], \sigma \cup [i:l, l:v_0, \ldots, l+n:v_n])_{X\setminus i} \mid\\
&\qquad (\sigma,[v_0,\ldots,v_n])_X \in [\![E]\!] \;\&\; \sigma \smile [i:v] \;\&\; l, l+1, \ldots, l+n \notin \mathtt{dom}(\sigma)\}\\
&[\![\mathbf{dispose}\ e]\!] = \{(\sigma \cup [v:v'], \sigma\backslash v)_{X\setminus v} \mid (\sigma,v)_X \in [\![e]\!] \;\&\; \sigma \smile [v:v']\}\\
&[\![c_1; c_2]\!] = \{\alpha_1;\alpha_2 \mid \alpha_1 \in [\![c_1]\!] \;\&\; \alpha_2 \in [\![c_2]\!] \;\&\; \alpha_1 \asymp \alpha_2\} = [\![c_1]\!];[\![c_2]\!]\\
&[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!] = [\![b]\!]_{true};[\![c_1]\!] \;\cup\; [\![b]\!]_{false};[\![c_2]\!]\\
&[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] = ([\![b]\!]_{true};[\![c]\!])^*;[\![b]\!]_{false} \;\cup\; ([\![b]\!]_{true};[\![c]\!])^\omega\\
&[\![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]\!] = \mathit{wait}^*; \mathit{enter} \;\cup\; \mathit{wait}^\omega\\
&\quad \text{where } \mathit{wait} = \mathit{acq}(r)[\![b]\!]_{false}\mathit{rel}(r) \;\cup\; \{\mathit{try}(r)\}\\
&\qquad \mathit{enter} = \mathit{acq}(r)[\![b]\!]_{true};[\![c]\!]\mathit{rel}(r)\\
&[\![\mathbf{resource}\ r\ \mathbf{in}\ c]\!] = \{\mathit{cat}(\alpha\backslash r) \mid \alpha \in [\![c]\!]_r\}\\
&[\![\mathbf{local}\ i = e\ \mathbf{in}\ c]\!] = \{(\sigma,\sigma)_X;(\alpha\backslash i) \mid\\
&\qquad\qquad (\sigma,v)_X \in [\![e]\!] \;\&\; \alpha \in [\![c]\!]_{i=v} \;\&\; (\sigma,\sigma)_X \asymp (\alpha\backslash i)\}
\end{aligned}
$$

Note that primitive commands such as **skip**, assignment, update, lookup, allocation, and disposal have traces consisting of a single footstep. The traces of a conditional critical region contain resource actions. Note the restriction, when building traces of $c_1; c_2$, to catenable combinations; similarly for conditional commands, loops, conditional critical regions, and parallel composition.

The semantic clause for **resource** r **in** c involves traces of c which assume that r is *local*, so that no concurrent process can access r, and that r is initially available. This reflects the scoping rules: the scope of the local resource name is c and excludes any processes running in parallel.

For a trace set T we let T_r be the set of traces in T which are *sequential for* r and consistent with r being initially available; more precisely, these are the traces α in T such that $\alpha\backslash r$ is catenable and $\{\} \stackrel{\alpha\lceil r}{\Longrightarrow} \cdot$. We write $\alpha\backslash r$ for the sequence obtained by deleting all actions in α that involve r. The catenability requirement on $\alpha\backslash r$ means that for each subsequence of α of the form $\beta\, \mathit{acq}(r)\lambda$ we have $\beta \asymp \lambda$. This constraint reflects the fact that r is a local resource, so there can be no interference from outside: no concurrent process can synchronize using the local resource. For a set A of resource names we

define

$$A \xrightarrow{try(r)} A \qquad \text{if } r \in A$$
$$A \xrightarrow{acq(r)} A \cup \{r\} \text{ if } r \notin A$$
$$A \xrightarrow{rel(r)} A - \{r\} \text{ if } r \in A$$

Thus we have $\{\} \xrightarrow{\alpha\lceil r} \cdot$ if and only if $\alpha\lceil r$ is executable assuming that r is a local resource which is initially available. Equivalently, this holds if and only if $\alpha\lceil r$ is a prefix of a trace in the set $(acq(r)\, try(r)^\infty\, rel(r))^\infty$.

The semantic clause for **local** $i = e$ **in** c involves a related notion of sequentiality: $[\![c]\!]_{i=v}$ is the set of traces of c which are *sequential for* i and consistent with the initial value v for i. A trace with these properties represents an execution in which no concurrent process can affect the value of i, and this corresponds to the fact that i is a local variable. For a trace α we define $\alpha\backslash i$ to be the trace obtained by deleting i from all states and read-only tags. When α is catenable and sequential for i, the trace $\alpha\backslash i$ will also be catenable.

There is considerable overlap with the corresponding Reynolds semantic clauses [19], and with action trace semantics [7]. The main differences involve the collapsing of consecutive state changes, book-keeping for read-only tags, and the restriction to catenable traces.

8 Examples

(i) $[\![x{:=}x+1]\!] = \{([x:v],[x:v+1])_{\{\}} \mid v \in V_{int}\}$

(ii) $[\![x{:=}x+1; x{:=}x+1]\!] = \{([x:v],[x:v+2])_{\{\}} \mid v \in V_{int}\} = [\![x{:=}x+2]\!]$

(iii) $[\![x{:=}x+1 \| x{:=}x+1]\!] = \{([x:v], abort) \mid v \in V_{int}\}$

(iv) $[\![\textbf{with } r \textbf{ do } x{:=}x+1]\!] = try(r)^*\, acq(r)\, [\![x{:=}x+1]\!]\, rel(r) \;\cup\; \{try(r)^\omega\}$

(v) $[\![\textbf{with } r \textbf{ do } x{:=}x+1 \| \textbf{with } r \textbf{ do } x{:=}x+1]\!]$ contains traces of the forms $acq(r)\,\alpha\, rel(r)\, acq(r)\,\beta\, rel(r)$, $acq(r)\,\alpha\, rel(r)\, try(r)^\omega$, and $try(r)^\omega$, where $\alpha, \beta \in [\![x{:=}x+1]\!]$, as well as traces of similar form containing additional $try(r)$ steps. Only traces of the first kind in which $\alpha \asymp \beta$ are sequential for r. It follows that

$$\begin{aligned}&[\![\textbf{resource } r \textbf{ in } (\textbf{with } r \textbf{ do } x{:=}x+1 \| \textbf{with } r \textbf{ do } x{:=}x+1)]\!]\\ &\quad = \{\alpha;\beta \mid \alpha,\beta \in [\![x{:=}x+1]\!] \;\&\; \alpha \asymp \beta\}\\ &\quad = [\![x{:=}x+1; x{:=}x+1]\!] = [\![x{:=}x+2]\!].\end{aligned}$$

(vi) The command $[x]{:=}x+1 \| [y]{:=}y+1$ has traces of the form $([x:v, y:v', v:n, v':n'], [x:v, y:v', v:v+1, v':v'+1])_{\{x,y\}}$

where $v \neq v'$, and traces of the form $([x : v, y : v, v : n], \mathit{abort})$, for all $v, n, v', n' \in V_{int}$.

The command **local** $x = 0$ **in** $([x]{:=}x + 1 \| [y]{:=}y + 1)$ has traces of the form $([y : v', 0 : n, v' : n'], [y : v', 0 : 1, v' : v' + 1])_{\{y\}}$ for all $v', n, n' \in V_{int}$ such that $v' \neq 0$, and traces of the form $([y : 0, 0 : n], \mathit{abort})$ for all $n \in V_{int}$. This command is semantically equivalent to $[0]{:=}1 \| [y]{:=}y + 1$.

(vii) $[\![x{:=}1; y{:=}2]\!] = \{([x{:}v, y{:}v'], [x{:}1, y{:}2])_{\{\}} \mid v, v' \in V_{int}\} = [\![y{:=}2; x{:=}1]\!]$.

(viii) Let c_1 and c_2 be the commands

$$\textbf{with } r_1 \textbf{ do with } r_2 \textbf{ do } x{:=}1$$

$$\textbf{with } r_2 \textbf{ do with } r_1 \textbf{ do } y{:=}2$$

The command **resource** r_1, r_2 **in** $(c_1 \| c_2)$ has the trace set

$$\{([\,], \bot)_{\{\}}\} \ \cup \ \{([x : v, y : v'], [x : 1, y : 2])_{\{\}} \mid v, v' \in V_{int}\}.$$

The first case represents the potential for deadlock, which occurs if the first thread acquires r_1 and the second acquires r_2.

(ix) It is easy to see that

$$[\![x{:=}\textbf{cons}(1)]\!] = \{([x : v_0], [x : v, v : 1])_{\{\}} \mid v_0, v \in V_{int}\}$$

$$[\![y{:=}\textbf{cons}(2)]\!] = \{([y : v_1], [y : v', v' : 2])_{\{\}} \mid v_1, v' \in V_{int}\}$$

It then follows that

$$\begin{aligned}
&[\![x{:=}\textbf{cons}(1) \| y{:=}\textbf{cons}(2)]\!] \\
&\quad = [\![x{:=}\textbf{cons}(1)]\!] \| [\![y{:=}\textbf{cons}(2)]\!] \\
&\quad = \{([x : v_0, y : v_1], [x : v, y : v', v : 1, v' : 2])_{\{\}} \mid v_0, v_1, v, v' \in V_{int}\} \\
&\quad = [\![x{:=}\textbf{cons}(1); y{:=}\textbf{cons}(2)]\!]
\end{aligned}$$

(x) $\textbf{dispose}(x); \textbf{dispose}(y)$ has trace set

$$\begin{aligned}
&\{([x : v, y : v', v : v_0, v' : v_0'], [x : v, y : v'])_{\{x,y\}} \mid v, v', v_0, v_0' \in V_{int} \ \&\ v \neq v'\} \\
&\qquad \cup \{([x : v, y : v, v : v'], \mathit{abort}) \mid v, v' \in V_{int}\}
\end{aligned}$$

(xi) **while true do** $([x]{:=}0; x{:=}x + 1)$ has the trace set

$$\{(\sigma, \bot)_{\{\}} \mid (x, v) \in \sigma \ \&\ \mathtt{dom}(\sigma) = \{x\} \cup \{n \in V_{int} \mid v \leq n\}\}.$$

(xii) Let PUT and GET be the following commands:

$$\mathit{PUT} :: \textbf{with } \mathit{buf} \textbf{ when } \mathit{full} = 0 \textbf{ do } (z{:=}x; \mathit{full}{:=}1)$$

$$\mathit{GET} :: \textbf{with } \mathit{buf} \textbf{ when } \mathit{full} = 1 \textbf{ do } (y{:=}z; \mathit{full}{:=}0)$$

The trace set of PUT is:

$$\begin{aligned}[\![PUT]\!] &= wait_0^*\, enter_0 \;\cup\; wait_0^\omega \\ wait_0 &= \{try(buf)\} \cup acq(buf)\, [\![full = 0]\!]_{false}\, rel(buf) \\ enter_0 &= acq(buf)\, [\![full = 0]\!]_{true}; [\![z{:=}x; full{:=}1]\!]\, rel(buf)\end{aligned}$$

Similarly, the trace set of GET is given by:

$$\begin{aligned}[\![GET]\!] &= wait_1^*\, enter_1 \;\cup\; wait_1^\omega \\ wait_1 &= \{try(buf)\} \cup acq(buf)\, [\![full = 1]\!]_{false}\, rel(buf) \\ enter_1 &= acq(buf)\, [\![full = 1]\!]_{true}; [\![y{:=}z; full{:=}0]\!]\, rel(buf)\end{aligned}$$

Let *prog* be the program

$$full{:=}0; \mathbf{resource}\ buf\ \mathbf{in}\ (x{:=}\mathbf{cons}(-); PUT) \| (GET; \mathbf{dispose}(y)).$$

The trace set of this program is

$$\begin{aligned}[\![prog]\!] = \{([full : v_0, x : v_1, y : v_2, z : v_3],\ [full : 0, x : v, y : v, z : v])_{\{\}} \\ \mid\ v, v_0, v_1, v_2, v_3 \in V_{int}\}.\end{aligned}$$

Note that this trace set consists of single footsteps, one for each possible initial state and heap cell chosen for x. In fact this is the same as the trace set of the sequential program $full{:=}0; x{:=}\mathbf{cons}(-); z{:=}x; y{:=}z; \mathbf{dispose}(y)$.

9 Semantic properties

As these examples suggest, the trace set of a resource-free command has a degenerate form, containing only single-step traces. (The program of example (xii) has no free resource names.) We can establish a more general property from which this result follows. We already introduced $\mathtt{res}(c)$, the set of resource names occurring free in c. Similarly, for a trace α, let $res(\alpha)$ be the set of resource names appearing in actions along α. The following result is easy to establish by structural induction.

Lemma 9.1 *For all commands c, and all traces $\alpha \in [\![c]\!]$, $res(\alpha) \subseteq \mathtt{res}(c)$.*

Given the structural constraints obeyed by all traces in $[\![c]\!]$, i.e. that successive footsteps get combined between resource actions, it follows that a command c with no free resource names will have only singleton traces.

Theorem 9.2 *For all commands c such that $\mathtt{res}(c) = \{\}$, every trace of c is a footstep, one of the three forms $(\sigma_0, \sigma_0')_X$, $(\sigma_0, abort)$, or $(\sigma_0, \bot)_X$, where σ_0 and σ_0' range over states.*

The trace set of such a command corresponds to a (non-deterministic) state transformation, represented as a subset of $\mathbf{St} \times (\mathbf{St} \cup \{abort, \bot\})$. We can make this correspondence explicit as follows.

A footstep specifies a "footprint": just the pieces of state needed to enable the step and describe its result [15,14]. Intuitively, a step of the form $(\sigma_0, \sigma_0')_X$ is executable (without error) from a "global" state σ if σ_0 is a subset of σ and the part of σ not needed for the step, which is assumed to remain unchanged, can be properly combined with σ_0' to produce the resulting state. On the other hand, if σ is consistent with σ_0 but missing part of σ_0 the step is still executable but leads to a runtime error because of an uninitialized variable. With this rationale, and similar considerations for the other forms of footsteps, we therefore define the *enabling relations* $\stackrel{\lambda}{\Longrightarrow} \subseteq \mathbf{St} \times (\mathbf{St} \cup \{abort, \bot\})$ as follows.

Definition 9.3 For a footstep λ the enabling relation $\stackrel{\lambda}{\Longrightarrow}$ is the least relation satisfying the following conditions:

- For a normal footstep $\lambda = (\sigma_0, \sigma_0')_X$
  - if $\sigma_0 \subseteq \sigma$ and $(\sigma - \sigma_0) \perp \sigma_0'$ then $\sigma \stackrel{(\sigma_0,\sigma_0')_X}{\Longrightarrow} (\sigma - \sigma_0) \cup \sigma_0'$
  - if $\sigma \restriction \mathtt{dom}(\sigma_0) \subset \sigma_0$ then $\sigma \stackrel{(\sigma_0,\sigma_0')_X}{\Longrightarrow} abort$
- For an error step $\lambda = (\sigma_0, abort)$,
  - if either $\sigma_0 \subseteq \sigma$ or $\sigma \restriction \mathtt{dom}(\sigma_0) \subset \sigma_0$ then $\sigma \stackrel{(\sigma_0,abort)}{\Longrightarrow} abort$
- For a non-terminating step $\lambda = (\sigma_0, \bot)_X$,
  - if $\sigma_0 \subseteq \sigma$ then $\sigma \stackrel{(\sigma_0,\bot)_X}{\Longrightarrow} \bot$
  - if $\sigma \restriction \mathtt{dom}(\sigma_0) \subset \sigma_0$ then $\sigma \stackrel{(\sigma_0,\bot)_X}{\Longrightarrow} abort$.

We interpret a resource action as having a trivial effect on the state, since such actions only affect and depend upon the availability of resources. Thus we will let $\sigma \stackrel{\lambda}{\Longrightarrow} \sigma$, for all states σ, when λ is a resource action. Since an error is terminal, we define $abort \stackrel{\lambda}{\Longrightarrow} abort$ for all actions λ; similarly we define $\bot \stackrel{\lambda}{\Longrightarrow} \bot$ for all λ.

We can then generalize further to obtain the effect of an arbitrary footstep trace α, by composing the effects of its actions in the appropriate order. For a finite sequence $\alpha = \lambda_0 \ldots \lambda_n$ we let $\sigma \stackrel{\alpha}{\Longrightarrow} \sigma'$ if and only if there is a sequence $\sigma_0, \ldots, \sigma_{n-1}$ such that $\sigma \stackrel{\lambda_0}{\Longrightarrow} \sigma_0 \stackrel{\lambda_1}{\Longrightarrow} \cdots \stackrel{\lambda_{n-1}}{\Longrightarrow} \sigma_{n-1} \stackrel{\lambda_n}{\Longrightarrow} \sigma'$. For an infinite sequence $\alpha = \lambda_0 \ldots \lambda_n \ldots$ we let $\sigma \stackrel{\alpha}{\Longrightarrow} \bot$ if there is an infinite sequence $\sigma_0, \ldots, \sigma_n, \ldots$ such that $\sigma \stackrel{\lambda_0}{\Longrightarrow} \sigma_0 \stackrel{\lambda_1}{\Longrightarrow} \cdots \stackrel{\lambda_{n-1}}{\Longrightarrow} \sigma_{n-1} \stackrel{\lambda_n}{\Longrightarrow} \sigma_n \cdots$

We then obtain, for a command c, a non-deterministic state transformation $|c| \subseteq \mathbf{St} \times (\mathbf{St} \cup \{abort, \bot\})$ in the obvious way.

Definition 9.4 For a program c, the state transformation denoted by c is

given by:

$$|c| = \bigcup\{(\sigma, \sigma') \mid \exists \alpha \in [\![c]\!].\ \sigma \stackrel{\alpha}{\Longrightarrow} \sigma'\}.$$

This definition describes not just the minimal pieces of state change caused by a program but specifies, for an arbitrary initial state, the possible results of executing the program without interference.

Note that for resource-free programs we can reason compositionally in the usual way about state transformations; for example, with the obvious notion of relational composition we have $|c_1; c_2| = |c_2| \circ |c_1|$, when c_1 and c_2 are resource-free. This simple formula echoes the familiar denotational semantic clause for sequential composition of non-deterministic guarded commands. Our semantics thus generalizes the standard relational semantics of non-deterministic sequential programs to the parallel setting in a natural manner.

With these definitions in hand we can now formulate precisely what it means for a command to be *error-free* from a given initial state.

Definition 9.5 The command c is *error-free* from state σ if no trace of c executable from σ leads to *abort*:

$$\forall \alpha \in [\![c]\!].\ \neg(\sigma \stackrel{\alpha}{\Longrightarrow} \mathit{abort}).$$

Equivalently, c is error-free from σ if and only if $(\sigma, \mathit{abort}) \notin |c|$.

Error-freedom implies the absence of race conditions and dangling pointers, as well as the avoidance of uninitialized variables. For instance, example (xii) above shows that the program

$$\mathit{full}{:=}0; \mathbf{resource}\ \mathit{buf}\ \mathbf{in}$$
$$(x{:=}\mathbf{cons}(-); \mathit{PUT}) || (\mathit{GET}; \mathbf{dispose}(y))$$

is error-free from any state σ with $\mathit{dom}(\sigma) \supseteq \{\mathit{full}, x, y, z\}$. Even though the first process allocates a heap cell and the second disposes it, there is no danger here. This is an example of *ownership transfer* [14].

In contrast, the program

$$\mathit{full}{:=}0; \mathbf{resource}\ \mathit{buf}\ \mathbf{in}$$
$$(x{:=}\mathbf{cons}(-); \mathit{PUT}; \mathbf{dispose}(x)) || (\mathit{GET}; \mathbf{dispose}(y))$$

is not error-free from any state, because even if the program is executed from a state in which full, x, y, z all have initial values both processes will attempt to dispose the same cell. If the program is started from a state in which one of the free identifiers is uninitialized there will also be a runtime error because the program will try to read or write this identifier. The footstep trace set for

this program is

$$\{([\mathit{full} : v_0, x : v_1, y : v_2, z : v_3],\ \mathit{abort}) \mid\ v_0, v_1, v_2, v_3 \in V_{\mathit{int}}\}.$$

Note that for each state σ there is a way to choose v_0, v_1, v_2, v_3 such that

$$\sigma \xrightarrow{([\mathit{full}:v_0, x:v_1, y:v_2, z:v_3],\ \mathit{abort})} \mathit{abort}.$$

10 Footstep traces and action traces

In previous work we introduced a semantics based on action traces [7] and used it to formulate a suitable notion of validity for a class of resource-sensitive partial correctness formulas and to establish soundness for a form of concurrent separation logic. The footstep trace model can be obtained by abstraction from the action traces model, retaining only the details relevant for reasoning about partial correctness and race-freedom. In fact, footstep trace semantics is strictly more abstract than action trace semantics; whenever two commands have the same action traces they will also have the same footstep traces, but the converse does not hold. Since action trace equivalence implies footstep trace equivalence, the footstep trace semantics satisfies all laws of program equivalence that hold for action trace semantics. In addition, footstep trace semantics satisfies "sequential" laws such as $[\![x{:=}x{+}1; x{:=}x{+}1]\!] = [\![x{:=}x{+}2]\!]$, which fails in action trace semantics.

We refer the reader to [7] for the action trace semantic definitions, but we will briefly summarize some of the main concepts in enough detail to justify our claim. Intuitively, the footstep trace set of a program can be constructed by identifying a subset of the program's action traces – the "loosely coupled" action traces of the program – and collapsing adjacent actions affecting the state in such traces, to avoid making intensional distinctions based on the relative order of such actions. An action trace is said to be loosely coupled if it represents an interactive computation of the program in which processes running concurrently are assumed to cause state changes only through synchronization, so that we need only allow for external state change when the program acquires a resource.

Let μ range over the action alphabet used in action trace semantics, as described by the following abstract grammar, in which i ranges over identifiers, $v, v', v_0, \ldots, v_n$ over integers, and r over resource names:

$$\begin{aligned}\mu ::= {} & \delta \mid i{=}v \mid i{:=}v \mid [v]{=}v' \mid [v]{:=}v' \mid \\ & \mathit{alloc}(v, [v_0, \ldots, v_n]) \mid \mathit{disp}(v) \mid \\ & \mathit{try}(r) \mid \mathit{acq}(r) \mid \mathit{rel}(r) \mid \mathit{abort}\end{aligned}$$

An action trace is a finite or infinite sequence of actions drawn from this alphabet.

Each action μ has an *effect* $\stackrel{\mu}{\Longrightarrow}$ represented as a partial function from $\mathbf{St}$ to $\mathbf{St} \cup \{abort\}$. The effect of an action specifies the states in which the action is enabled, and what happens to the state when the action occurs. For example, the read action $i{=}v$ is only enabled in σ if $\sigma(i) = v$, and causes no state change.

We then obtain for a finite action trace α an effect $\stackrel{\alpha}{\Longrightarrow} \subseteq \mathbf{St} \times (\mathbf{St} \cup \{abort\})$, by composing the actions sequentially. Similarly for an infinite action trace α we have $\stackrel{\alpha}{\Longrightarrow} \subseteq \mathbf{St} \times \{\bot\}$. As in [7] we can define an interference relation on actions: two actions interfere, written $\mu \bowtie \mu'$, if one action involves a write to an identifier or heap cell used by the other action.

For each action μ we define a set $\mathcal{F}(\mu)$ of footstep actions:

$$\mathcal{F}(\delta) = \{([\,],\ [\,])_{\{\}}\}$$
$$\mathcal{F}(i{=}v) = \{([i : v],\ [i : v])_{\{i\}}\}$$
$$\mathcal{F}(i{:=}v) = \{([i : v_0],\ [i : v])_{\{\}} \mid v_0 \in V_{int}\}$$
$$\mathcal{F}([v]{=}v') = \{([v : v'],\ [v : v'])_{\{v\}}\}$$
$$\mathcal{F}([v]{:=}v') = \{([v : v_0],\ [v : v'])_{\{\}} \mid v_0 \in V_{int}\}$$
$$\mathcal{F}(alloc(v, [v_0, \ldots, v_n])) = \{([\,],\ [v : v_0, v+1 : v_1, \ldots, v+n : v_n])_{\{\}}\}$$
$$\mathcal{F}(disp(v)) = \{([v : v_0],\ [\,])_{\{\}} \mid v_0 \in V_{int}\}$$
$$\mathcal{F}(\mu) = \{\mu\} \qquad \text{if } \mu \text{ is a resource action}$$
$$\mathcal{F}(rel(r)) = \{rel(r)\}$$
$$\mathcal{F}(try(r)) = \{try(r)\}$$
$$\mathcal{F}(abort) = \{abort\}$$

Note that a write action $i{:=}v$ determines a set of footsteps, each of which writes the value v to i; the set $\mathcal{F}(i{:=}v)$ contains a footstep for each possible initial value of i. For similar reasons update actions $[v]{:=}v'$, allocations, and disposals all generate non-trivial sets of footsteps. In all other cases the action μ determines a single footstep action.

The key relationships between the notions of effect and interference used in the two models can be summarized as follows:

- For all states σ and σ',

$$\sigma \stackrel{\mu}{\Longrightarrow} \sigma' \text{ iff } \exists \lambda \in \mathcal{F}(\mu).\ \sigma \stackrel{\lambda}{\Longrightarrow} \sigma'.$$

- For all actions μ, and all states σ, σ'':

$$(\exists \sigma'.\ \sigma \stackrel{\mu}{\Longrightarrow} \sigma' \stackrel{\mu'}{\Longrightarrow} \sigma'') \text{ iff } \exists \lambda \in \mathcal{F}(\mu), \lambda' \in \mathcal{F}(\mu').\ \lambda \asymp \lambda' \ \&\ \sigma \stackrel{\lambda;\lambda'}{\Longrightarrow} \sigma''.$$

- For all actions μ, μ':

$$\mu \bowtie \mu' \ \Leftrightarrow\ \exists \lambda \in \mathcal{F}(\mu), \lambda' \in \mathcal{F}(\mu').\ \lambda \bowtie \lambda'.$$

With these results at hand we can now state and prove the key technical result relating the two models.

Theorem 10.1 (i) *If two expressions denote the same action trace set, they also denote the same footstep trace set.*

(ii) *If two commands denote the same action trace set, they also denote the same footstep trace set.*

Proof (Sketch) The details for expressions are simpler, so we focus on commands. Let us say that an action trace is *loosely coupled* if it can be executed from some initial state, allowing interference only at synchronizations. We can formalize this property as follows. Define a *loosely coupled enabling relation* $\stackrel{\mu}{\hookrightarrow}$ for actions μ, so that:

- $\sigma \stackrel{\mu}{\hookrightarrow} \sigma'$ iff $\sigma \stackrel{\mu}{\Longrightarrow} \sigma'$, when μ is not an acquire action
- $\sigma \stackrel{\mu}{\hookrightarrow} \sigma'$ for all σ and σ', if μ is $acq(r)$ for some resource name r.

As before, this generalizes to action traces in the obvious way by composition, so that for a finite sequence $\alpha = \mu_1 \ldots \mu_n$ we let $\sigma \stackrel{\alpha}{\hookrightarrow} \sigma'$ iff there is a sequence of states $\sigma_1, \ldots, \sigma_{n-1}$ such that $\sigma \stackrel{\mu_1}{\hookrightarrow} \sigma_1 \stackrel{\mu_2}{\hookrightarrow} \cdots \stackrel{\mu_{n-1}}{\hookrightarrow} \sigma_{n-1} \stackrel{\mu_n}{\hookrightarrow} \sigma'$. We say that the action trace α is loosely coupled if there is a state σ such that $\sigma \stackrel{\alpha}{\hookrightarrow} \cdot$.

For a loosely coupled action trace α the set $\mathcal{F}(\alpha)$ represents the set of catenable footstep traces generated by α. Let $[\![c]\!]_{fs}$ be the footstep trace set of c, let $[\![c]\!]_{act}$ be the action trace set of c as defined in [7], and let $[\![c]\!]_{lc} \subseteq [\![c]\!]_{act}$ be the set of loosely coupled action traces of c. The following property can be proven by structural induction on c, based on the semantic clauses:

$$[\![c]\!]_{fs} = \{cat(\beta) \mid \beta \in \mathcal{F}(\alpha)\ \&\ \alpha \in [\![c]\!]_{lc}\}.$$

Hence action trace equivalence implies footstep trace equivalence. □

We have shown that, for all pairs of commands c_1 and c_2, if $[\![c_1]\!]_{act} = [\![c_2]\!]_{act}$ then $[\![c_1]\!]_{fs} = [\![c_2]\!]_{fs}$. This property makes precise our claim that footstep trace semantics is more abstract than action trace semantics. The fact that the converse is false is easily demonstrated by considering some example programs. Action trace sets contain traces in which the ordering of individual read and write actions is recorded; these intermediate details are ignored in the construction of the new model, because of the "mumbling" effect built into our def-

inition of concatenation. For example, the commands $x{:=}1; x{:=}x+1$ and $x{:=}2$ have the same footstep traces, but different action traces, since $x{:=}1; x{:=}x+1$ has action traces of the form $x{:=}1\, x{=}v\, x{:=}v+1$, for $v \in V_{int}$. Similarly the commands $x{:=}1; y{:=}2$ and $y{:=}2; x{:=}1$ have the same footstep traces but not the same action traces. In addition, the action trace model distinguishes between the commands **while true do** $x{:=}x+1$ and **while true do** $x{:=}x+2$, whereas they have the same footstep traces, namely $\{([x:v], \bot)_{\{\}} \mid v \in V_{int}\}$.

A more compelling illustration of the advantages of footstep traces over action traces is provided by some programs involving the one-place buffer resource and the code fragments PUT and GET. The footstep trace semantics of the programs

$$\begin{array}{l} \mathit{full}{:=}0; \mathbf{resource}\ \mathit{buf}\ \mathbf{in} \\ \qquad (x{:=}\mathbf{cons}(-); PUT) \| (GET; \mathbf{dispose}(y)) \end{array}$$

and

$$\begin{array}{l} \mathit{full}{:=}0; \mathbf{resource}\ \mathit{buf}\ \mathbf{in} \\ \qquad (x{:=}\mathbf{cons}(-); PUT; \mathbf{dispose}(x)) \| GET \end{array}$$

are identical, both equal to the set

$$\begin{array}{l} \{([\mathit{full}:v_0, x:v_1, y:v_2, z:v_3],\ [\mathit{full}:0, x:v, y:v, z:v])_{\{\}} \\ \qquad \mid\ v, v_0, v_1, v_2, v_3 \in V_{int}\}. \end{array}$$

Their action trace sets differ, because the action traces retain information about the relative ordering of steps: in the second program the disposal action may occur concurrently with GET but in the first program it follows the GET.

Furthermore, consider what happens when we iterate the buffer processes, running a sequence of puts concurrently with a sequence of gets. For all non-negative integers N, the program

$$\begin{array}{l} \mathit{full}{:=}0; \mathbf{resource}\ \mathit{buf}\ \mathbf{in} \\ \qquad (x{:=}\mathbf{cons}(-); PUT)^N \| (GET; \mathbf{dispose}(y))^N \end{array}$$

has *exactly the same* footstep traces as the above programs: again the trace set degenerates to a set of single footsteps, and the effect is still to (re)set *full* to 0 and to make x, y, and z all equal; in contrast, the action trace set of this program (using the semantics of [7]) suffers from a combinatorial explosion, and contains many traces that yield the same effect.

Similarly all programs of the form

$$\mathit{full}{:=}0; \mathbf{resource}\ \mathit{buf}\ \mathbf{in}$$
$$(x{:=}\mathbf{cons}(-); PUT; \mathbf{dispose}(x))^N \| GET^N$$

have the same trace set again.

Although these two families of programs (parameterized by N) are rather contrived, they demonstrate the potential of our new semantics for greatly reducing the need to consider interleavings. It is also worth remarking that the fact that each of these programs has the same set of footstep traces implies that they all satisfy exactly the same formulas of concurrent separation logic. This is by no means obvious; each family of programs requires a different proof strategy in the logic [14] because one needs to choose an appropriate resource invariant that expresses the ownership transfer policy implicit in the program structure. For programs in the first family ownership of the heap cell is deemed to transfer after a put operation, whereas this is not the case for programs in the second family.

11 Concurrent separation logic

Concurrent separation logic [7,14] is a Hoare-style formalism for specifying and proving resource-sensitive partial correctness formulas for parallel programs that share mutable state, modifying the inference rules of the Owicki-Gries logic to make judicious use of separating conjunction [18] in a manner proposed by Peter O'Hearn. A resource-sensitive partial correctness formula has the form $\Gamma \vdash \{p\}c\{q\}$, where Γ is a resource context specifying a partition of the shared variables of c among resources, together with a resource invariant for each resource. The pre- and post-conditions p and q, as well as the resource invariants, are separation logic formulas [18] subject to certain restrictions. O'Hearn [14] has shown that this logic allows correctness proofs for a number of interesting examples, involving a novel form of "ownership transfer" in which permission to manipulate a pointer may be deemed to transfer from one process to another. The soundness of this logic is far from obvious: the original Owicki-Gries rules are unsound in the presence of pointers, and John Reynolds has shown that O'Hearn's rules are not sound unless certain constraints are imposed on the class of formulas allowed as resource invariants.

Our earlier paper [7] introduced a notion of validity for concurrent separation logic formulas, phrased in terms of a resource-sensitive "local enabling" relation $\underset{\Gamma}{\Longrightarrow}$ based on *action trace semantics*, and proved soundness of the logic with respect to this notion of validity, assuming that resource invariants are *precise* separation logic formulas [18]. This semantics involved actions for

reading and writing to identifiers, lookup and update of heap locations, and allocation and disposal of heap locations, in addition to resource actions, an idle action, and an error action. The idea is that when $\sigma \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \sigma'$ the action trace α describes an execution in which a process performs the steps of α in a parallel context which obeys the Dijkstra principle by respecting the resource invariants and only accessing shared state when holding the relevant resources.

Using footstep traces instead of action traces leads to an analogous notion of local enabling, again describing executions in an environment that respects resources. We can then formulate the analogous notion of validity for concurrent separation logic formulas, based on footsteps. We say that $\Gamma \vdash \{p\}c\{q\}$ is *footstep-valid* if for all σ satisfying p, all footstep traces $\alpha \in [\![c]\!]$, and all $\sigma' \neq \perp$ such that $\sigma \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \sigma'$, σ' satisfies q. (As in [7] but using footstep traces rather than action traces to underpin the definition.) Although the difference in underlying model suggests that the notion of footstep-validity might differ from the notion of validity used in [7], in fact the two notions coincide. We will discuss some of the ramifications of this result after a brief summary of the main ideas behind the proof.

Theorem 11.1 *$\Gamma \vdash \{p\}c\{q\}$ is action-trace valid iff it is footstep-trace valid.*

Proof (Sketch) The resource context Γ specifies, for each resource occurring in c, a set of identifiers protected by that resource, and a resource invariant. By analogy with [7], we can define a "local enabling" relation $\stackrel{\lambda}{\Longrightarrow}_{\Gamma}$ for footstep actions λ, modifying the definition of $\stackrel{\lambda}{\Longrightarrow}$ to take account of the resource invariants and protection lists, and treating any violation of the resource discipline as an error. We show that for all μ and μ':

$$\sigma \stackrel{\mu}{\Longrightarrow}_{\Gamma} \sigma' \text{ iff } \exists \lambda \in \mathcal{F}(\mu).\ \sigma \stackrel{\lambda}{\Longrightarrow}_{\Gamma} \sigma'.$$

In addition, for all actions μ and μ', and all states σ, σ'',

$$(\exists \sigma'.\ \sigma \stackrel{\mu}{\Longrightarrow}_{\Gamma} \sigma' \stackrel{\mu'}{\Longrightarrow}_{\Gamma} \sigma'') \text{ iff } \exists \lambda \in \mathcal{F}(\mu), \lambda' \in \mathcal{F}(\mu').\ \lambda \asymp \lambda' \ \&\ \sigma \stackrel{\lambda;\lambda'}{\Longrightarrow}_{\Gamma} \sigma''.$$

With the obvious extension of $\mathcal{F}$ to sequences of actions, it then follows that for all action traces α, and all σ and σ'', $\sigma \stackrel{\alpha}{\Longrightarrow}_{\Gamma} \sigma''$ if and only if there is a footstep trace $\beta \in \mathcal{F}(\alpha)$ such that $\sigma \stackrel{\beta}{\Longrightarrow}_{\Gamma} \sigma''$. Hence the two notions of validity coincide. □

The following theorem can then be proven directly, using footstep trace semantics instead of action traces and following the proof strategy of [7]. It can also be deduced from the soundness result of [7] and the previous theorem.

Theorem 11.2 (Soundness of concurrent separation logic) *Concurrent separation logic is sound with respect to footstep-trace validity: every resource-*

sensitive partial correctness formula provable from the inference rules is footstep-trace valid.

As a consequence one can use footstep traces in preference to action traces to formalize a correctness proof of a parallel program, an advantage because of the larger step size, which may reduce the number of interleavings. Indeed, we have already shown earlier that footstep trace semantics supports *sequential reasoning* about synchronization-free code fragments: one can replace any synchronization-free code fragment by another which denotes the same state transformation, without affecting the footstep trace semantics of the overall program. One can use more elementary reasoning techniques to deal with synchronization-free code, such as the familiar Hoare-style inference rules for sequential programs.

12 Granularity

We have worked so far with states whose structure assumes that an integer-valued variable is stored as a single integer: we abstracted away from machine-level details such as word size. If instead we assume that programs are executed on a machine with a fixed, finite word size, it would be natural to introduce a more concrete model of state in which we represent an integer-valued variable as a list of word-valued component variables. For purely sequential programs it is well known that the more concrete model is unnecessary if we care only about high-level properties: a state transformation semantics based on integers is consistent with the more concrete semantics based on words. The same is true for the footstep trace semantics. It is straightforward to define a low-level word-based version of footstep traces, with variables represented as lists of components, each holding an integer small enough to fit inside a single word. Since the semantic definitions ensure that consecutive (and catenable) footsteps get collapsed into a single step, a sequence of word-level operations that implements an integer-level operation (like writing an integer to a variable) will give rise to a single word-level footstep. It can then be shown that the high-level footstep semantics is equipollent to the low-level, in that two programs have the same high-level semantics if and only if they have the same low-level semantics. This result is analogous to a similar result presented by the author at MFPS 2004 [8], which discussed high- and low-level versions of action trace semantics [7] and the relationship between the two versions. The result can be seen as a justification for adopting a more abstract model with no loss of generality.

13 Future work

If c_1 and c_2 have the same footstep traces, then for all program contexts $P[-]$, $P[c_1]$ and $P[c_2]$ satisfy the same concurrent separation logic formulas.[6] The converse fails: for example the commands $(x{:=}1; y{:=}2) \| x{:=}2$ and $x{:=}1 \| x{:=}2$ have different footstep traces, since the first command has traces of the form (σ, abort) where $x, y \in \mathtt{dom}(\sigma)$ whereas the second command does not need a value for y; nevertheless these commands satisfy the same concurrent separation logic formulas (none!), because they both cause a race. It would be interesting to see if we can further fine-tune the structure of footstep traces to avoid making distinctions like this.

We have indicated the greater succinctness of the new semantic model by applying it to some examples, and by proving a theorem to the effect that for synchronization-free programs the semantics takes a simple degenerate form equivalent to a (non-deterministic) state transformation. The semantic definitions may be used directly to calculate footstep trace sets and analyze program behavior, and this kind of manual analysis is made less tedious by the streamlined structure of the semantic model. The semantics also validates a number of laws of program equivalence which can be used to simplify program analysis: in particular, all laws of equivalence valid for action trace semantics are also valid for footstep traces, as was shown above. Nevertheless, the manual approach is still error-prone and for large-scale programs with many synchronization points we still face severe combinatorial problems, so that mechanical assistance would be an advantage. We believe that our approach may suggest a way to improve the efficiency of *model checking* [12,9], since our semantics is designed to avoid unnecessary interleaving and hence may help to tame the combinatorial explosion.

Recent papers [1,3] have proposed methods for reasoning about parallel programs based on *permissions*, building on earlier work of Boyland [4] and extending concurrent separation logic to deal with classic but logically tricky examples such as readers/writers [10]. Initial investigations suggest that the semantics presented in this paper offers a highly promising way to analyze such programs: the model's succinctness and its careful design to embody Dijkstra's principle facilitate compositional reasoning. We plan to use footstep trace semantics to prove the soundness of Bornat's permission-based logic [1], as it seems likely that the proof method used in [7] may be adjustable in a natural manner to incorporate permission accounting, by analogy with the way the original soundness proof formalizes and justifies reasoning about ownership transfer.

[6] A similar implication holds for action trace semantics, by the previous remarks.

References

[1] R. Bornat. *Variables as Resource in Separation Logic*. MFPS 2005, Proceedings. Martin Escardó (Ed.), Birmingham, England, May 18-21, 2005. To appear, Elsevier ENTCS (2006).

[2] P. Brinch Hansen. *Structured multiprogramming*. Comm. ACM, 15(7):574-578, July 1972.

[3] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. *Permission accounting in separation logic*. Proc. POPL 2005. Pages 259–270. SIGPLAN-SIGACT. January 2005.

[4] J. Boyland. *Checking interference with fractional permissions*. In: Static Analysis, 10^{th} International Symposium, R. Cousot, ed. Springer LNCS vol. 2694 (2003), pp. 55–72.

[5] P. Brinch Hansen. *Concurrent programming concepts*. ACM Computing Surveys 5(4):223-245, December 1973.

[6] S. D. 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.

[7] S. D. 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. Full version to appear in *Theoretical Computer Science* (2006).

[8] S. D. Brookes. *A Race-detecting Semantics for Concurrent Programs*. Slides from talk presented at MFPS 2004. Pittsburgh, May 2004.

[9] E. M. Clarke, O. Grumberg, and D. Peled. **Model Checking**. MIT Press, 1999.

[10] P. J. Courtois, F. Heymans, and D. Parnas. *Concurrent control with "readers" and "writers"*. Comm. ACM. 14 (1971), pp. 667–668.

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

[12] E. A. Emerson and E. M. Clarke. *Design and synthesis of synchronization skeletons using branching time temporal logic*. In *Logic of Programs*, Workshop, Yorktown Heights, NY, May 1981. Springer LNCS vol. 131 (1981).

[13] 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.

[14] P. W. O'Hearn. *Resources, Concurrency, and Local Reasoning*. Invited paper, CONCUR 2004, P. Gardner and N. Yoshida (Eds.), Springer LNCS 3170, London, August/September 2004, pp. 49-67. Full version to appear in *Theoretical Computer Science* (2006).

[15] P. W. O'Hearn, J. C. Reynolds, and H. Yang. *Local reasoning about programs that alter data structures*. Proc. CSL 2001, L. Fribourg, ed. Springer LNCS vol. 2142 (2001), pp. 1–19.

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

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

[18] J. C. Reynolds, *Separation logic: a logic for shared mutable data structures*, Invited paper. Proc. 17^{th} IEEE Conference on Logic in Computer Science, LICS 2002, pp. 55-74. IEEE Computer Society, 2002.

[19] 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.