Full Abstraction for a Shared Variable Parallel Language

Stephen Brookes

April 1993
CMU-CS-93-141

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science, LICS'93, Montreal, Quebec (Canada).

Abstract

We give a new denotational semantics for a shared variable parallel programming language and prove full abstraction: the semantics gives identical meanings to commands if and only if they induce the same partial correctness behavior in all program contexts. The meaning of a command is a set of "transition traces", which record the ways in which a command may interact with and be affected by its environment. We show how to modify the semantics to incorporate new program constructs, to allow for different levels of granularity or atomicity, and to model fair infinite computation, in each case achieving full abstraction with respect to an appropriate notion of program behavior.

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. Support also came from the National Science Foundation under Grant No. CCR-9006064.

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 the U.S. Government.

Keywords: denotational semantics, operational semantics, parallel programming, partial correctness, safety properties, liveness properties, fairness

1 Introduction

One of the fundamental purposes of semantics is to provide rigorous means of proving the correctness of programs with respect to behavioral specifications. For any particular language different semantic models may be suitable for reasoning about different behavioral notions, such as *partial correctness*, *total correctness*, and *deadlock-freedom*. Ideally one would like a semantics in which the meaning of one term coincides with the meaning of another term if and only if the terms induce the same behavior in each program context; this guarantees that one term may be replaced by the other in any context without affecting the behavior of the overall program, thus supporting compositional or modular reasoning about program behavior. Such a semantics is *equationally fully abstract* with respect to the given notion of behavior [10, 13, 15]. When the set of program behaviors is equipped with an approximation ordering and the semantic model has a partial order such that the meaning of one term is less than the meaning of another if and only if the behavior of the first term in each program context approximates the behavior of the second term in the same context, the semantics is *inequationally fully abstract* with respect to the given notion of program behavior and approximation. An inequationally fully abstract semantics is also equationally fully abstract.

The difficulty of finding fully abstract semantics is well known [2, 10, 13, 15]. Many standard semantic models are *correct*, in that whenever two terms induce different behavior in some context they denote different meanings, but *too concrete* since the converse may fail. Sometimes one can show that by adding extra syntactic constructs to the programming language the model becomes fully abstract. However, unless the extra constructs are computationally natural and the original language was clearly deficient because of their omission, the full abstraction problem for the original language is still important.

The standard state-transformation semantics for sequential while-programs is fully abstract with respect to partial correctness behavior. However, for a parallel version of this language [5, 11], in which parallel commands can interact by updating and reading shared variables, the full abstraction problem is more difficult. Parallel programs may exhibit non-deterministic behavior, depending on the scheduling of atomic actions, so the partial correctness behavior of a parallel command is naturally modelled as a non-deterministic state transformation, usually represented as a function from states to *sets* of states. However, the state transformation denoted by a parallel combination of commands cannot be determined solely from the state transformations denoted by the component commands; thus the state-transformation semantics for a parallel language is not even compositional, and is certainly not fully abstract. One needs a semantic model with more detailed structure, so that the possible interactions between commands executing in parallel may be modelled appropriately.

Hennessy and Plotkin [5] described a denotational semantics for this language, based on a recursively defined domain of *resumptions*, built with a powerdomain operator. However, the resumptions semantics is too concrete: **skip** and **skip**;**skip** denote different resumptions even though they induce the same partial correctness behavior in all contexts. They showed that with the addition of extra features to the programming language, the resumptions model becomes fully abstract. However, one of the extra constructs is a rather peculiar form of coroutine execution which allows counting of the number of atomic steps taken by a command executing in parallel. The problem remained of finding a fully abstract model for the original parallel language.

In this paper we solve this problem: we describe a new denotational semantics for this language, and we show that it is fully abstract with respect to partial correctness behavior. We model the meaning of a command as a set of *transition traces*. A transition trace is a finite sequence of pairs

of states recording a possible interaction sequence of the command with its environment; each pair of states represents the effect of a finite, possibly empty, sequence of atomic actions. The set of traces of a command is closed under two natural operations: "stuttering" (*cf.* Lamport [9]) and "mumbling". This model is conceptually simpler than the resumptions model, since it does not require the use of powerdomains or recursively defined domains. The model also validates a number of intuitively natural equations and inequations between programs which fail in the resumptions model.

We show that our semantic model is adaptable to a variety of settings: one may easily accommodate the addition of certain extra features to the programming language, and the results do not depend crucially on assumptions about the level of atomicity or granularity of execution. We show that the semantic model can be extended to model fair infinite computations, producing a fully abstract semantics with respect to the appropriate notion of behavior, in which both termination and non-termination are regarded as observable. This semantics may be used to reason about total correctness, and about safety and liveness properties, of parallel programs executing fairly.

Previous Work

We have already mentioned the relationship between our semantics and the resumptions model of Hennessy and Plotkin [5].

The idea of using sequences or traces of some kind to model the behavior of concurrent programs is widespread. For instance, several authors have used traces to build models of determinate or indeterminate dataflow networks, notably [7, 8, 14]. Indeed, others have also used sequences of pairs of states [3, 6, 12] in imperative settings. However, in these papers a pair of states represents a *single* atomic action while in our model it represents a *finite sequence* of atomic actions. The semantics presented in [3, 6] are for different languages and different notions of program behavior. Park's semantics [12] for the same language that we discuss is too concrete, distinguishing between **skip** and **skip**;**skip** again, because his traces record step-by-step behavior exactly. Our work shows how to adapt his semantics to obtain full abstraction. Abadi and Plotkin [1] use a trace model (prefix-closed sets of finite sequences of pairs of states, also closed under stuttering and mumbling) for reasoning about safety properties of reactive systems and the study of composition rules.

2 Syntax

We discuss a standard shared variable parallel language, as in [5, 11]. There are four syntactic sets: **Ide**, the set of identifiers, ranged over by I; **Exp**, the set of expressions, ranged over by E; **BExp**, the set of boolean expressions, ranged over by B; and **Com**, the set of commands, ranged over by C. Identifiers and expressions denote integer values, boolean expressions denote truth values, and the language contains the usual arithmetic and boolean operators and constants. For commands we specify the following grammar:

$$\begin{array}{rcl} C & ::= & \textbf{skip} \mid I{:=}E \mid C_1;C_2 \mid C_1 \| C_2 \mid \\ & & \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \mid \textbf{ while } B \textbf{ do } C \mid \\ & & \textbf{await } B \textbf{ then } C \end{array}$$

A command of the form **await** B **then** C is a *conditional critical region*, converting C into an atomic action that is enabled only in states satisfying B; we impose the (reasonable) syntactic restriction that C must be a finite sequence of assignments (or **skip**).

3 An operational semantics

We present a structural operational semantics similar to the semantics given in [5].

We use N for the set of integers, ranged over by n; and $V = \{\mathtt{tt}, \mathtt{ff}\}$ for the set of truth values, ranged over by v. A state is a finite partial function from identifiers to integer values. Let $S = [\mathbf{Ide} \rightarrow_p N]$ denote the set of states, ranged over by s. We write $\mathrm{dom}(s)$ for the domain of s, and $[s \mid I = n]$ for the state which agrees with s except that it gives identifier I the value n. We use notation like $[I_1 = n_1, \ldots, I_k = n_k]$ for states.

When s is a state defined on (at least) the free identifiers of E, we write $\langle E, s\rangle \rightarrow^* n$ to indicate that E evaluates to n in state s. Similarly for boolean expressions. We assume that the semantics of expressions and boolean expressions are given by semantic functions $\mathcal{E}$ and $\mathcal{B}$, characterized operationally by:

$$\mathcal{E}[\![E]\!] = \{(s, n) \mid \langle E, s\rangle \rightarrow^* n\}$$
$$\mathcal{B}[\![B]\!] = \{(s, v) \mid \langle B, s\rangle \rightarrow^* v\}.$$

For command execution we specify a set of configurations

$$\mathbf{Conf} = \{\langle C, s\rangle \in \mathbf{Com} \times S \mid \mathrm{free}[\![C]\!] \subseteq \mathrm{dom}(s)\},$$

a subset of *successfully terminated* configurations, and a transition relation $\rightarrow \subseteq \mathbf{Conf} \times \mathbf{Conf}$. The successfully terminated configurations are those for which $\langle C, s\rangle term$ is provable. A configuration that is not successfully terminated but has no enabled transition is *deadlocked*. The transition rules, given in Figure 1, specify that boolean expression evaluations, assignments, and conditional critical regions are atomic actions. Later we will show how to adapt our semantics to model finer levels of atomicity or granularity of execution.

4 Partial correctness behavior

We define the partial correctness behavior function $\mathcal{M} : \mathbf{Com} \rightarrow \mathcal{P}(S \times S)$ by:

$$\mathcal{M}[\![C]\!] = \{(s, s') \mid \langle C, s\rangle \rightarrow^* \langle C', s'\rangle term\},$$

and we put $\mathcal{M}[\![C]\!]s = \{s' \mid (s, s') \in \mathcal{M}[\![C]\!]\}$.

This induces a preorder $\sqsubseteq_{\mathcal{M}}$ and an equivalence relation $\equiv_{\mathcal{M}}$ on commands:

$$C \sqsubseteq_{\mathcal{M}} C' \iff \forall s.(\mathrm{free}[\![C]\!] \cup \mathrm{free}[\![C']\!] \subseteq \mathrm{dom}(s) \Rightarrow \mathcal{M}[\![C]\!]s \subseteq \mathcal{M}[\![C']\!]s)$$
$$C \equiv_{\mathcal{M}} C' \iff C \sqsubseteq_{\mathcal{M}} C' \;\&\; C' \sqsubseteq_{\mathcal{M}} C.$$

Partial correctness equivalence is not a *substitutive* relation, since we have:

$$x{:=}1; x{:=}x + 1 \;\equiv_{\mathcal{M}}\; x{:=}2$$
$$(x{:=}1; x{:=}x + 1) \| x{:=}2 \;\not\equiv_{\mathcal{M}}\; x{:=}2 \| x{:=}2.$$

We therefore define the substitutive preorder $\leq_{\mathcal{M}}$ and substitutive equivalence relation $=_{\mathcal{M}}$:

$$C \leq_{\mathcal{M}} C' \iff \forall P[\cdot].(P[C] \sqsubseteq_{\mathcal{M}} P[C'])$$
$$C =_{\mathcal{M}} C' \iff C \leq_{\mathcal{M}} C' \;\&\; C' \leq_{\mathcal{M}} C,$$

where $P[\cdot]$ ranges over program contexts, that is, programs with a hole (denoted $[\cdot]$) into which a command may be substituted; and $P[C]$ denotes the program obtained by substituting C into the hole. Thus $C =_{\mathcal{M}} C'$ if and only if C and C' are interchangeable in all program contexts without affecting partial correctness.

$$\langle \mathbf{skip}, s\rangle term$$

$$\frac{\langle E, s\rangle \rightarrow^* n}{\langle I{:=}E, s\rangle \rightarrow \langle \mathbf{skip}, [s \mid I = n]\rangle}$$

$$\frac{\langle C_1, s\rangle \rightarrow \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \rightarrow \langle C_1'; C_2, s'\rangle}$$

$$\frac{\langle C_1, s\rangle term}{\langle C_1; C_2, s\rangle \rightarrow \langle C_2, s\rangle}$$

$$\frac{\langle C_1, s\rangle \rightarrow \langle C_1', s'\rangle}{\langle C_1 \| C_2, s\rangle \rightarrow \langle C_1' \| C_2, s'\rangle}$$

$$\frac{\langle C_2, s\rangle \rightarrow \langle C_2', s'\rangle}{\langle C_1 \| C_2, s\rangle \rightarrow \langle C_1 \| C_2', s'\rangle}$$

$$\frac{\langle C_1, s\rangle term \quad \langle C_2, s\rangle term}{\langle C_1 \| C_2, s\rangle term}$$

$$\frac{\langle B, s\rangle \rightarrow^* \mathtt{tt}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \rightarrow \langle C_1, s\rangle}$$

$$\frac{\langle B, s\rangle \rightarrow^* \mathtt{ff}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \rightarrow \langle C_2, s\rangle}$$

$$\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \rightarrow \langle \mathbf{if}\ B\ \mathbf{then}\ C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{else}\ \mathbf{skip}, s\rangle$$

$$\frac{\langle B, s\rangle \rightarrow^* \mathtt{tt} \quad \langle C, s\rangle \rightarrow^* \langle C', s'\rangle term}{\langle \mathbf{await}\ B\ \mathbf{then}\ C, s\rangle \rightarrow \langle \mathbf{skip}, s'\rangle}$$

Figure 1: Operational semantics for commands

5 Denotational semantics

Resumptions

Hennessy and Plotkin [5] gave a denotational semantics based on a domain R of "resumptions", defined recursively by the domain equation

$$R = S \to \mathcal{P}(S + (R \times S)),$$

where $\mathcal{P}$ is a suitable powerdomain constructor, $+$ denotes the separated sum and $\times$ denotes the cartesian product of domains. However, the resumptions semantics makes many unnecessary distinctions between programs: for instance **skip** and **skip**;**skip** denote different resumptions even though they induce the same partial correctness properties in all contexts.

Hennessy and Plotkin added a form of "coroutine" composition C_1 **co** C_2 to the syntax of the programming language, together with a non-deterministic choice operation C_1 **or** C_2. The operational behavior of C_1 **co** C_2 is to perform single atomic steps alternately from C_1 and C_2 until one of them terminates, and C_1 **or** C_2 can behave either like C_1 or like C_2. These two extra constructs permit program contexts to be built which can count the number of atomic actions taken by a command, thus distinguishing between **skip** and **skip**;**skip**. The resumptions model then becomes fully abstract for this extended language. Nevertheless, this coroutine construct seems rather *ad hoc* and the full abstraction problem for the original language remained open.

Transition traces

The main problem with the resumptions model is that it represents explicitly the one-step transition relation $\to$ and is therefore forced to distinguish between too many commands. Instead we design a semantic model based on the reflexive, transitive closure of the transition relation (denoted $\to^*$).

Informally, a *transition trace* of a command C is defined to be a finite sequence of pairs of states $(s_0, s_0')(s_1, s_1') \ldots (s_k, s_k')$ such that it is possible for C to perform a computation from s_0 to s_k' if execution is interrupted k times, the i^{th} interruption changing the state from s_i' to s_{i+1} $(0 \leq i < k)$. A transition trace of this form is *interference-free* iff $s_i' = s_{i+1}$ for each i. The degenerate case $(k = 0)$ yields simply a pair (s, s') such that C has a computation from s terminating in s'. Formally, we write $\mathcal{T}[\![C]\!]$ for the set of transition traces of C, characterized operationally by:

$$\begin{aligned}\mathcal{T}[\![C]\!] = \{&(s_0, s_0')(s_1, s_1') \ldots (s_k, s_k') \mid \\ &\langle C, s_0\rangle \to^* \langle C_1, s_0'\rangle \ \& \\ &\langle C_1, s_1\rangle \to^* \langle C_2, s_1'\rangle \ \& \\ &\ldots\ldots\ldots\ldots\ldots\ldots \ \& \\ &\langle C_k, s_k\rangle \to^* \langle C', s_k'\rangle \mathit{term}\}.\end{aligned}$$

Proposition 5.1 *For all commands C, $\mathcal{M}[\![C]\!] = \{(s, s') \mid (s, s') \in \mathcal{T}[\![C]\!]\}$.*

This operational characterization of $\mathcal{T}$ has some obvious but important consequences following from the fact that $\to^*$ is reflexive and transitive:

Proposition 5.2 *The set of transition traces of a command C is closed under "stuttering" and "mumbling": for all α, $\beta \in (S \times S)^*$ and all s, s', $s'' \in S$,*

$$\begin{aligned}&\alpha\beta \in \mathcal{T}[\![C]\!] \ \Rightarrow \ \alpha(s, s)\beta \in \mathcal{T}[\![C]\!] \\ &\alpha(s, s')(s', s'')\beta \in \mathcal{T}[\![C]\!] \ \Rightarrow \ \alpha(s, s'')\beta \in \mathcal{T}[\![C]\!].\end{aligned}$$

Given a set T of transition traces, we let $T^{\dagger}$, the *closure* of T, be the smallest set containing T and closed under stuttering and mumbling. We say that T is *closed* if $T = T^{\dagger}$. By the above result, $\mathcal{T}[\![C]\!]$ is closed.

Let $\Sigma = S \times S$, and let $\mathcal{P}^{\dagger}(\Sigma^+)$ denote the set of closed sets of (non-empty) traces, ordered by inclusion. It is easy to see that this forms a complete lattice, with least element the empty set and with least upper bounds given by unions.

The standard notion of *concatenation* for finite sequences can be adapted easily to this setting. When T_1 and T_2 are closed sets of traces we define

$$T_1;T_2 \;=\; \{\alpha\beta \mid \alpha \in T_1 \;\&\; \beta \in T_2\}^{\dagger}.$$

We also extend the Kleene-star operation to closed sets of traces in the obvious way: T^* denotes the smallest set containing T and the empty trace, closed under stuttering, mumbling and concatenation.

Similarly, the standard notion of *interleaving* on finite traces is given inductively by:

$$\begin{aligned} \alpha\|\epsilon &= \epsilon\|\alpha = \{\alpha\} \\ \sigma\alpha\|\rho\beta &= \{\sigma\gamma \mid \gamma \in \alpha\|\rho\beta\} \cup \{\rho\gamma' \mid \gamma' \in \sigma\alpha\|\beta\}, \end{aligned}$$

where σ and ρ range over Σ, α and β range over Σ^*, and ϵ is the empty trace[1]. When T_1 and T_2 are closed sets of traces we define

$$T_1\|T_2 = \bigcup\{\alpha\|\beta \;\mid\; \alpha \in T_1 \;\&\; \beta \in T_2\}^{\dagger}.$$

We can now give a denotational characterization for $\mathcal{T}$. To simplify the presentation, and to facilitate comparison with later developments, it is convenient to define $\mathcal{T}[\![B]\!] = \{(s,s) \mid (s,\mathtt{tt}) \in \mathcal{B}[\![B]\!]\}^{\dagger}$.

Proposition 5.3 *The (finite) transition traces semantic function $\mathcal{T} : \mathbf{Com} \to \mathcal{P}^{\dagger}(\Sigma^+)$ is characterized uniquely by the following clauses:*

$$\begin{aligned} &\mathcal{T}[\![\mathbf{skip}]\!] = \{(s,s) \mid s \in S\}^{\dagger} \\ &\mathcal{T}[\![I{:=}E]\!] = \{(s,[s \mid I = n]) \mid (s,n) \in \mathcal{E}[\![E]\!]\}^{\dagger} \\ &\mathcal{T}[\![C_1;C_2]\!] = \mathcal{T}[\![C_1]\!];\mathcal{T}[\![C_2]\!] \\ &\mathcal{T}[\![C_1\|C_2]\!] = \mathcal{T}[\![C_1]\!]\|\mathcal{T}[\![C_2]\!] \\ &\mathcal{T}[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!] = \mathcal{T}[\![B]\!];\mathcal{T}[\![C_1]\!] \cup \mathcal{T}[\![\neg B]\!];\mathcal{T}[\![C_2]\!] \\ &\mathcal{T}[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = (\mathcal{T}[\![B]\!];\mathcal{T}[\![C]\!])^*;\mathcal{T}[\![\neg B]\!] \\ &\mathcal{T}[\![\mathbf{await}\ B\ \mathbf{then}\ C]\!] = \{(s,s') \in \mathcal{T}[\![C]\!] \mid (s,s) \in \mathcal{T}[\![B]\!]\}^{\dagger}. \end{aligned}$$

Note that all operations on closed sets of traces used in this semantic definition are monotone (even continuous) with respect to set inclusion. An alternative (and equivalent) definition of the trace semantics of loops can be given using least fixed points:

$$\mathcal{T}[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = \mu T.(\mathcal{T}[\![B]\!];\mathcal{T}[\![C]\!];T \;\cup\; \mathcal{T}[\![\neg B]\!]).$$

[1] Although transition traces are always non-empty, some of our definitions are simpler if we include the empty trace.

6 Full abstraction

Given the assumption that expression evaluation is atomic, the only important aspect of an expression's operational behavior in the transition rules for commands is its final value. It follows trivially that two expressions induce the same partial correctness behavior in all program contexts if and only if they evaluate to the same results in all states. Thus, $\mathcal{E}$ is fully abstract for the expression sub-language, and $\mathcal{B}$ is fully abstract for the boolean expression sub-language.

We now show that the transition traces semantics for commands is (inequationally) fully abstract with respect to partial correctness behavior.

We define $\mathcal{T}[\![C]\!]s = \{s'\alpha \mid (s,s')\alpha \in \mathcal{T}[\![C]\!]\}$ and:

$$C \sqsubseteq_{\mathcal{T}} C' \iff \forall s.(\mathrm{free}[\![C]\!] \cup \mathrm{free}[\![C']\!] \subseteq \mathrm{dom}(s) \;\Rightarrow\; \mathcal{T}[\![C]\!]s \subseteq \mathcal{T}[\![C']\!]s)$$
$$C \equiv_{\mathcal{T}} C' \iff C \sqsubseteq_{\mathcal{T}} C' \;\&\; C' \sqsubseteq_{\mathcal{T}} C.$$

Proposition 6.1 *The transition traces semantics $\mathcal{T}$ is inequationally fully abstract: for all commands C and C', $C \sqsubseteq_{\mathcal{T}} C' \iff C \leq_{\mathcal{M}} C'$.*

Proof: Suppose $C \sqsubseteq_{\mathcal{T}} C'$. Since $\mathcal{T}$ is a denotational semantics, for each program context $P[\cdot]$ the only relevant aspect of C in determining $\mathcal{T}[\![P[C]]\!]$ is $\mathcal{T}[\![C]\!]$. Moreover, all operations used in the semantic definitions are monotone with respect to set inclusion. Thus we get $\mathcal{T}[\![P[C]]\!] \subseteq \mathcal{T}[\![P[C']]\!]$. But then for all relevant states s,

$$\begin{aligned}\mathcal{M}[\![P[C]]\!]s &= \{s' \mid (s,s') \in \mathcal{T}[\![P[C]]\!]\} \\ &\subseteq \{s' \mid (s,s') \in \mathcal{T}[\![P[C']]\!]\} \\ &= \mathcal{M}[\![P[C']]\!]s.\end{aligned}$$

This shows that $C \sqsubseteq_{\mathcal{T}} C' \;\Rightarrow\; C \leq_{\mathcal{M}} C'$.

Since states are finite, for each state s there is a boolean expression IS_s that evaluates to `tt` from s' if s' agrees with s on $\mathrm{dom}(s)$, and evaluates to `ff` otherwise. Similarly there is a command MAKE_s such that

$$\langle \mathrm{MAKE}_s, s' \rangle \to^* \langle \mathbf{skip}, s \rangle$$

for all states such that $\mathrm{dom}(s') = \mathrm{dom}(s)$. Such a command can easily be defined as a finite sequence of assignments to the identifiers in $\mathrm{dom}(s)$.

Now suppose $C \not\sqsubseteq_{\mathcal{T}} C'$, so that there is some transition trace $\alpha = (s_0, s_0')(s_1, s_1') \ldots (s_k, s_k')$ belonging to $\mathcal{T}[\![C]\!]$ and not $\mathcal{T}[\![C']\!]$. Let DO_α be the command

$$\begin{array}{l}\mathbf{await}\ \mathrm{IS}_{s_0'}\ \mathbf{then}\ \mathrm{MAKE}_{s_1}; \\ \mathbf{await}\ \mathrm{IS}_{s_1'}\ \mathbf{then}\ \mathrm{MAKE}_{s_2}; \\ \ldots \\ \mathbf{await}\ \mathrm{IS}_{s_{k-1}'}\ \mathbf{then}\ \mathrm{MAKE}_{s_k}.\end{array}$$

Let $P_\alpha[\cdot]$ be the program context $[\cdot] \| \mathrm{DO}_\alpha$. By assumption that $\alpha \in \mathcal{T}[\![C]\!] - \mathcal{T}[\![C']\!]$ it follows that

$$(s_0, s_k') \in \mathcal{M}[\![P_\alpha[C]]\!] - \mathcal{M}[\![P_\alpha[C']]\!],$$

so $C \not\leq_{\mathcal{M}} C'$. Thus, $C \not\sqsubseteq_{\mathcal{T}} C'$ implies $C \not\leq_{\mathcal{M}} C'$. That completes the proof. ∎

For example, consider the commands $C = x{:=}1; x{:=}x+1$ and $C' = x{:=}1; x{:=}2$. They have the same partial correctness semantics but different transition traces: clearly

$$\alpha = ([x=0],[x=1])([x=0],[x=1])$$

$\mathbf{skip}; C \equiv C \equiv C; \mathbf{skip}$
$(C_1; C_2); C_3 \equiv C_1; (C_2; C_3)$
$C \| \mathbf{skip} \equiv C$
$C_1 \| C_2 \equiv C_2 \| C_1$
$(C_1 \| C_2) \| C_3 \equiv C_1 \| (C_2 \| C_3)$
$C_1; (C_2 \| C) \sqsubseteq (C_1; C_2) \| C$
$(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2); C \equiv \mathbf{if}\ B\ \mathbf{then}\ C_1; C\ \mathbf{else}\ C_2; C$
$\mathbf{if}\ (B_1 \& B_2)\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \sqsubseteq \mathbf{if}\ B_1\ \mathbf{then}\ (\mathbf{if}\ B_2\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2)\ \mathbf{else}\ C_2$
$\mathbf{while}\ B\ \mathbf{do}\ C \equiv \mathbf{if}\ B\ \mathbf{then}\ C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{else}\ \mathbf{skip}$
$\mathbf{await}\ (B_1 \& B_2)\ \mathbf{then}\ C\ \equiv \mathbf{await}\ B_1\ \mathbf{then}\ (\mathbf{await}\ B_2\ \mathbf{then}\ C)$
$\mathbf{await}\ \mathbf{false}\ \mathbf{then}\ C \sqsubseteq C'$

Figure 2: Some laws of parallel programming

is a transition trace of C but not of C'. The context $P_\alpha[\cdot]$ built in the proof above is

$$[\cdot] \parallel \mathbf{await}\ x = 1\ \mathbf{then}\ x{:=}0$$

and it is clear that $P_\alpha[C]$ may terminate with $x = 1$ but that $P_\alpha[C']$ cannot.

Similarly, consider the commands $x{:=}0$ and $x{:=}0; x{:=}0$. It is easy to see that $\mathcal{T}[\![x{:=}0]\!] \subseteq \mathcal{T}[\![x{:=}0; x{:=}0]\!]$, and this inclusion is proper. The transition trace $([x = 1], [x = 0])([x = 1], [x = 0])$ is possible for $x{:=}0; x{:=}0$ but not for $x{:=}0$. These two commands can be distinguished by running them in parallel with the command $\mathbf{await}\ x = 0\ \mathbf{then}\ x{:=}1$.

7 Laws of parallel programming

We can use this semantics to prove equations and inequations between programs, with the guarantee that these laws may be safely used for reasoning about partial correctness, in any program context. Some examples are given in Figure 2, in which $\equiv$ stands for $\equiv_{\mathcal{T}}$ and $\sqsubseteq$ stands for $\sqsubseteq_{\mathcal{T}}$. The majority of these laws fail in the resumptions model and in Park's model. The laws may be easily validated in our semantics, taking advantage of natural algebraic identities involving $T_1; T_2$, $T_1 \| T_2$, and T^*.

A consequence of these laws is the inequality $C_1; C_2 \sqsubseteq C_1 \| C_2$. If the expression language is deterministic, so that for all E and s the set $\mathcal{E}[\![E]\!]s$ contains at most one value, we also obtain the inequation:

$$I{:=}[E_1/I]E_2 \sqsubseteq I{:=}E_1; I{:=}E_2,$$

where $[E_1/I]E_2$ denotes the expression obtained by substituting E_1 for each free occurrence of I in E_2, with appropriate changes of bound variable to avoid capturing any free identifiers of E_1.

This semantics identifies deadlock (e.g. $\mathbf{await}\ \mathbf{false}\ \mathbf{then}\ C$) with divergence (e.g. $\mathbf{while}\ \mathbf{true}\ \mathbf{do}\ \mathbf{skip}$). This is reasonable, since a deadlocked program and a diverging program vacuously satisfy the same partial correctness properties in every program context. In addition, since assignment is atomic, this semantics satisfies the law $I{:=}I \equiv \mathbf{skip}$.

8 Finer granularity

Our semantics can be adapted to deal with finer levels of granularity. For instance, we might allow interruption of an assignment $I{:=}E$ during the evaluation of E, and interruption of a conditional

during the evaluation of its test. To make the discussion precise, suppose that we have the following abstract syntax for boolean expressions and integer expressions:

$$B ::= \mathbf{true} \mid \mathbf{false} \mid \neg B \mid B_1 \& B_2 \mid E_1 \leq E_2$$
$$E ::= 0 \mid 1 \mid I \mid E_1 + E_2 \mid \mathbf{if}\ B\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2$$

To adapt the operational semantics we introduce the set $\mathbf{BExp}'$ of extended boolean expressions, defined by adding the clauses $B ::= v$ $(v \in V)$ to the grammar for **BExp**, and the set $\mathbf{Exp}'$ of extended integer expressions, defined by adding $E ::= n$ $(n \in N)$ to the grammar for **Exp**. We use configurations of form $\langle E, s\rangle$ and $\langle B, s\rangle$, where E and B are extended expressions. A configuration of form $\langle n + E_2, s\rangle$ (with $n \in N$) represents a stage in evaluation of a sum expression where the left-hand expression has been evaluated to the integer n and the right-hand expression remaining to be computed is E_2; a configuration of form $n \in N$ represents the final result of evaluation.

A fine-grained operational semantics for expressions is described in Figures 3 and 4. Note that the transition rules specify that a conjunction $B_1 \& B_2$ is evaluated from left-to-right with a short-circuit strategy, avoiding evaluation of B_2 if B_1 evaluates to `ff`. On the other hand we specify that in a sum expression $E_1 + E_2$ the two sub-expressions are evaluated in parallel. These choices were made solely for illustration, and the transition rules may easily be modified to model different evaluation strategies without affecting the general properties of our semantics.

Now that expression evaluation is no longer atomic, the semantic functions $\mathcal{E}$ and $\mathcal{B}$ are not fully abstract. Instead we need to extend the transition traces semantics to cover expressions, to allow for the possibility that the state may change during evaluation. Since we assume that expression evaluation never causes any side-effects, we can use a slightly simpler trace structure than for commands[2]:

$$\begin{aligned}\mathcal{T}[\![B]\!] = \{&((s_0,s_0)(s_1,s_1)\ldots(s_k,s_k),v) \mid\\ &\langle B, s_0\rangle \to^* \langle B_1, s_0\rangle \ \&\\ &\langle B_1, s_1\rangle \to^* \langle B_2, s_1\rangle \ \&\\ &\ldots\ldots\ldots\ldots\ldots\ \&\\ &\langle B_k, s_k\rangle \to^* v\}\end{aligned}$$

$$\begin{aligned}\mathcal{T}[\![E]\!] = \{&((s_0,s_0)(s_1,s_1)\ldots(s_k,s_k),n) \mid\\ &\langle E, s_0\rangle \to^* \langle E_1, s_0\rangle \ \&\\ &\langle E_1, s_1\rangle \to^* \langle E_2, s_1\rangle \ \&\\ &\ldots\ldots\ldots\ldots\ldots\ \&\\ &\langle E_k, s_k\rangle \to^* n\}.\end{aligned}$$

Thus a trace $((s_0,s_0)(s_1,s_1)\ldots(s_k,s_k),v) \in \mathcal{T}[\![B]\!]$ means that there is an evaluation of B from initial state s_0 resulting in value v, during which the environment makes k interruptions, the i^{th} interruption changing the state to s_i. In particular allowing no interruptions corresponds to the definition of $\mathcal{B}$, and $\mathcal{B}[\![B]\!] = \{(s,n) \mid ((s,s),n) \in \mathcal{T}[\![B]\!]\}$. Note that the traces of an expression are again closed under (the obvious analogues of) stuttering and mumbling. For boolean expressions this amounts to the following:

Proposition 8.1 *For all boolean expressions B, all states s, all $\alpha, \beta \in \Sigma^*$, and all truth values v,*

$$(\alpha\beta, v) \in \mathcal{T}[\![B]\!] \ \Rightarrow\ (\alpha(s,s)\beta, v) \in \mathcal{T}[\![B]\!]$$
$$(\alpha(s,s)(s,s)\beta, v) \in \mathcal{T}[\![B]\!] \ \Rightarrow\ (\alpha(s,s)\beta, v) \in \mathcal{T}[\![B]\!].$$

[2]Actually, we could have used traces of form $(s_0 s_1 \ldots s_k, v)$, with minor modifications in what follows. Our notation is deliberately chosen so as to simplify some of the details that follow.

$$\langle \mathbf{true}, s\rangle \to \mathtt{tt}$$

$$\langle \mathbf{false}, s\rangle \to \mathtt{ff}$$

$$\frac{\langle B, s\rangle \to \langle B', s\rangle}{\langle \neg B, s\rangle \to \langle \neg B', s\rangle}$$

$$\frac{\langle B, s\rangle \to \mathtt{tt}}{\langle \neg B, s\rangle \to \mathtt{ff}}$$

$$\frac{\langle B, s\rangle \to \mathtt{ff}}{\langle \neg B, s\rangle \to \mathtt{tt}}$$

$$\frac{\langle B_1, s\rangle \to \langle B_1', s\rangle}{\langle B_1 \& B_2, s\rangle \to \langle B_1' \& B_2, s'\rangle}$$

$$\frac{\langle B_1, s\rangle \to \mathtt{tt}}{\langle B_1 \& B_2, s\rangle \to \langle B_2, s\rangle}$$

$$\frac{\langle B_1, s\rangle \to \mathtt{ff}}{\langle B_1 \& B_2, s\rangle \to \mathtt{ff}}$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s\rangle}{\langle E_1 \leq E_2, s\rangle \to \langle E_1' \leq E_2, s\rangle}$$

$$\frac{\langle E_2, s\rangle \to \langle E_2', s\rangle}{\langle E_1 \leq E_2, s\rangle \to \langle E_1 \leq E_2', s\rangle}$$

$$\langle m \leq n, s\rangle \to \mathtt{tt} \qquad \text{if } m \leq n$$

$$\langle m \leq n, s\rangle \to \mathtt{ff} \qquad \text{if } m > n$$

Figure 3: A fine-grained operational semantics for boolean expressions

$$\langle 0, s\rangle \to 0$$

$$\langle 1, s\rangle \to 1$$

$$\langle I, s\rangle \to s[\![I]\!]$$

$$\frac{\langle B, s\rangle \to \langle B', s\rangle}{\langle \textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2, s\rangle \to \langle \textbf{if } B' \textbf{ then } E_1 \textbf{ else } E_2, s\rangle}$$

$$\frac{\langle B, s\rangle \to \texttt{tt}}{\langle \textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2, s\rangle \to \langle E_1, s\rangle}$$

$$\frac{\langle B, s\rangle \to \texttt{ff}}{\langle \textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2, s\rangle \to \langle E_2, s\rangle}$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s\rangle}{\langle E_1 + E_2, s\rangle \to \langle E_1' + E_2, s\rangle}$$

$$\frac{\langle E_2, s\rangle \to \langle E_2', s\rangle}{\langle E_1 + E_2, s\rangle \to \langle E_1 + E_2', s\rangle}$$

$$\langle m + n, s\rangle \to k \qquad \text{if } m + n = k$$

Figure 4: A fine-grained operational semantics for integer expressions

$$\langle \mathbf{skip}, s\rangle \mathit{term}$$

$$\frac{\langle E, s\rangle \to \langle E', s\rangle}{\langle I{:=}E, s\rangle \to \langle I{:=}E', s\rangle}$$

$$\langle I{:=}n, s\rangle \to \langle \mathbf{skip}, [s \mid I = n]\rangle$$

$$\frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to \langle C_1'; C_2, s'\rangle}$$

$$\frac{\langle C_1, s\rangle \mathit{term}}{\langle C_1; C_2, s\rangle \to \langle C_2, s\rangle}$$

$$\frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1 \| C_2, s\rangle \to \langle C_1' \| C_2, s'\rangle}$$

$$\frac{\langle C_2, s\rangle \to \langle C_2', s'\rangle}{\langle C_1 \| C_2, s\rangle \to \langle C_1 \| C_2', s'\rangle}$$

$$\frac{\langle C_1, s\rangle \mathit{term} \quad \langle C_2, s\rangle \mathit{term}}{\langle C_1 \| C_2, s\rangle \mathit{term}}$$

$$\frac{\langle B, s\rangle \to \langle B', s\rangle}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \to \langle \mathbf{if}\ B'\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle}$$

$$\frac{\langle B, s\rangle \to \mathtt{tt}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \to \langle C_1, s\rangle}$$

$$\frac{\langle B, s\rangle \to \mathtt{ff}}{\langle \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2, s\rangle \to \langle C_2, s\rangle}$$

$$\langle \mathbf{while}\ B\ \mathbf{do}\ C, s\rangle \to \langle \mathbf{if}\ B\ \mathbf{then}\ C; \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{else}\ \mathbf{skip}, s\rangle$$

$$\frac{\langle B, s\rangle \to^* \mathtt{tt} \quad \langle C, s\rangle \to^* s'}{\langle \mathbf{await}\ B\ \mathbf{then}\ C, s\rangle \to s'}$$

Figure 5: A fine-grained operational semantics for commands

We write $\mathcal{P}^\dagger(\Sigma^+ \times V)$ for the set of closed sets, ordered again by inclusion. Similar properties hold for integer expressions, so that $\mathcal{T}[\![E]\!]$ is a closed subset of $\Sigma^+ \times N$.

So far we have characterized $\mathcal{T}[\![B]\!]$ and $\mathcal{T}[\![E]\!]$ operationally. As with commands, we can also give denotational definitions. We give the details only for boolean expressions.

Proposition 8.2 *The fine-grained trace semantics $\mathcal{T} : \mathbf{BExp} \to \mathcal{P}^\dagger(\Sigma^+ \times V)$ is uniquely characterized by the following clauses:*

$$\begin{array}{l}
\mathcal{T}[\![\mathbf{true}]\!] = \{((s,s),\mathtt{tt}) \mid s \in S\}^\dagger \\
\mathcal{T}[\![\mathbf{false}]\!] = \{((s,s),\mathtt{ff}) \mid s \in S\}^\dagger \\
\mathcal{T}[\![\neg B]\!] = \{(\alpha, \neg v) \mid (\alpha, v) \in \mathcal{T}[\![B]\!]\}, \quad \textit{where } \neg\mathtt{tt} = \mathtt{ff}, \neg\mathtt{ff} = \mathtt{tt} \\
\mathcal{T}[\![B_1 \& B_2]\!] = \{(\alpha,\mathtt{ff}) \mid (\alpha,\mathtt{ff}) \in \mathcal{T}[\![B_1]\!]\} \ \cup \ \{(\alpha\beta, v) \mid (\alpha,\mathtt{tt}) \in \mathcal{T}[\![B_1]\!] \ \&\ (\beta, v) \in \mathcal{T}[\![B_2]\!]\}^\dagger \\
\mathcal{T}[\![E_1 \le E_2]\!] = \{(\gamma, m \le n) \mid (\alpha, m) \in \mathcal{T}[\![E_1]\!] \ \&\ (\beta, n) \in \mathcal{T}[\![E_2]\!] \ \&\ \gamma \in \alpha \| \beta\}^\dagger.
\end{array}$$

An operational characterization of the fine-grained trace semantics of commands is given exactly as before, but using the fine-grained transition relation $\to$ from Figure 5:

$$\begin{array}{rl}
\mathcal{T}[\![C]\!] = \{(s_0, s_0')(s_1, s_1') \ldots (s_k, s_k') \mid & \\
& \langle C, s_0\rangle \to^* \langle C_1, s_0'\rangle \ \& \\
& \langle C_1, s_1\rangle \to^* \langle C_2, s_1'\rangle \ \& \\
& \ldots\ldots\ldots\ldots\ldots\ \& \\
& \langle C_k, s_k\rangle \to^* \langle C', s_k'\rangle \mathit{term}\}.
\end{array}$$

In the following denotational definition for $\mathcal{T}[\![C]\!]$ we identify $\mathcal{T}[\![B]\!]$ with the set $\{\alpha \mid (\alpha, \mathtt{tt}) \in \mathcal{T}[\![B]\!]\}$.

Proposition 8.3 *The fine-grained trace semantics of commands is uniquely characterized by the following clauses:*

$$\begin{array}{l}
\mathcal{T}[\![\mathbf{skip}]\!] = \{(s,s) \mid s \in S\}^\dagger \\
\mathcal{T}[\![I{:=}E]\!] = \{\alpha(s, [s \mid I = n]) \mid (\alpha, n) \in \mathcal{T}[\![E]\!]\}^\dagger \\
\mathcal{T}[\![C_1; C_2]\!] = \mathcal{T}[\![C_1]\!]; \mathcal{T}[\![C_2]\!] \\
\mathcal{T}[\![C_1 \| C_2]\!] = \mathcal{T}[\![C_1]\!] \| \mathcal{T}[\![C_2]\!] \\
\mathcal{T}[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!] = \mathcal{T}[\![B]\!]; \mathcal{T}[\![C_1]\!] \cup \mathcal{T}[\![\neg B]\!]; \mathcal{T}[\![C_2]\!] \\
\mathcal{T}[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = (\mathcal{T}[\![B]\!]; \mathcal{T}[\![C]\!])^*; \mathcal{T}[\![\neg B]\!] \\
\mathcal{T}[\![\mathbf{await}\ B\ \mathbf{then}\ C]\!] = \{(s,s') \in \mathcal{T}[\![C]\!] \mid (s,s) \in \mathcal{T}[\![B]\!]\}^\dagger.
\end{array}$$

Again all operations on trace sets used in this semantics are monotone (even continuous) with respect to set inclusion.

Of course, since the operational semantics of commands is now fine-grained, we are now interested in a fine-grained version of partial correctness behavior, which we still call $\mathcal{M}$, defined as before but using the fine-grained transition relation of Figure 5.

Proposition 8.4 *The fine-grained semantics is fully abstract with respect to fine-grained partial correctness: for all terms t and t' of the same syntactic type, $t \sqsubseteq_{\mathcal{T}} t' \iff t \le_{\mathcal{M}} t'$.*

Proof: For commands the proof is similar to the proof of Proposition 6.1.

For boolean expressions t and t' with different transition traces it is easy to construct a context of form $C\|\mathbf{if}\ [\cdot]\ \mathbf{then}\ z{:=}0\ \mathbf{else}\ z{:=}1$ (for a suitably chosen C) that distinguishes between them.

For integer expressions with different transition traces we can find a discriminating context of form $C\|z{:=}[\cdot]$. ∎

For example, the boolean expressions $x \leq x$ and **true** are not semantically equivalent, and they may induce different behavior in contexts such as

$$x{:=}1; (x{:=}0 \| \textbf{if } [\cdot] \textbf{ then } y{:=}1 \textbf{ else } y{:=}2).$$

The relationships given in Figure 2 continue to hold for the fine-grained semantics. However, the identity $I{:=}I \equiv \textbf{skip}$ fails because assignment is not atomic. For example,

$$x{:=}0; [x{:=}x \| x{:=}1] \not\equiv_{\mathcal{M}} x{:=}0; [\textbf{skip} \| x{:=}1].$$

This is because $([x = 0], [x = 0])([x = 1], [x = 0])$ is a transition trace of $x{:=}x$ but not of **skip**. Instead we get the inequality $\textbf{skip} \sqsubseteq I{:=}I$.

9 Fairness and strong correctness

So far we have ignored the possibility of infinite computation and non-termination. This was appropriate for reasoning about partial correctness. However, many parallel programs are designed specifically not to terminate, and we would like a semantics suitable for reasoning about total correctness, and about safety and liveness properties, in addition to partial correctness. Moreover, when reasoning about parallel programs it is often natural to make a *fairness* assumption [12]: when running commands in parallel, no individual command is forever denied its turn for execution. It is well known that the assumption of fairness implies unbounded nondeterminism, and that in many models (typically using powerdomains) this causes lack of continuity of various semantic functions [2, 12].

Despite this, we can model fair infinite execution of parallel programs simply by extending our transition trace model to include fair infinite traces. A (fair) infinite trace of a command C is a sequence

$$(s_0, s_0')(s_1, s_1') \ldots (s_n, s_n')(s_{n+1}, s_{n+1}') \ldots$$

describing a (fair) infinite computation of C from initial state s_0 during which execution is interrupted infinitely often, the i^{th} interruption changing the state from s_i' to s_{i+1} (for each $i \geq 0$). Each (s_i, s_i') represents a finite (possibly empty) sequence of atomic actions performed by the command, and infinitely many of these action sequences must be non-empty[3].

Every finite transition trace of C is fair. In order to characterize the fair infinite computations of a command operationally, the fairness condition must be applied to each parallel sub-command of C: care must be taken to keep track of which syntactic component of C performs each atomic action in a computation. See for example [4].

Let $\mathcal{T}[\![C]\!]$ now denote the set of fair transition traces of C. For obvious reasons only finitely many interruptions can occur between successive atomic actions by C; consequently, $\mathcal{T}[\![C]\!]$ is again closed under stuttering and mumbling, where we allow finitely many stutters or mumbles between successive stages in a trace. We continue to use the notation $T^\dagger$ for the closure of T, where T now ranges over $\Sigma^\infty = \Sigma^+ \cup \Sigma^\omega$, the set of finite or infinite transition traces. Let $\mathcal{P}^\dagger(\Sigma^\infty)$ denote the set of closed sets of finite or infinite traces. This again forms a complete lattice under set inclusion.

[3]For example, this requirement guarantees that C has an infinite interference-free trace beginning in state s iff $\langle C, s\rangle$ has a fair infinite computation.

We extend concatenation to fair traces in the obvious way: $\alpha\beta$ is defined to be α if α is an infinite sequence. Then we define $T_1;T_2$ and T^* on closed sets of finite or infinite traces as before. We also define[4]

$$T^\omega = \{\alpha_0\alpha_1 \ldots \alpha_n \ldots \mid \forall n \geq 0.\alpha_n \in T\}^\dagger.$$

For α and β in Σ^∞ let $\alpha\|\beta$ be the set of all traces built by fairly interleaving α with β. Perhaps the simplest way to define $\alpha\|\beta$ formally, following Park [12], is:

$$\begin{array}{l}
\alpha\|\beta = \{\gamma \mid (\alpha,\beta,\gamma) \in \mathit{fairmerge}\} \\
\mathit{fairmerge} = (L^*RR^*L)^\omega \cup (L \cup R)^*A \\
L = \{(\sigma,\epsilon,\sigma) \mid \sigma \in \Sigma\} \\
R = \{(\epsilon,\sigma,\sigma) \mid \sigma \in \Sigma\} \\
A = \{(\alpha,\epsilon,\alpha) \mid \alpha \in \Sigma^\infty\} \ \cup \ \{(\epsilon,\beta,\beta) \mid \beta \in \Sigma^\infty\},
\end{array}$$

where we extend concatenation to work on sets and on triples of traces in the obvious way: $AB = \{\alpha\beta \mid \alpha \in A, \beta \in B\}$ and $(\alpha_1,\alpha_2,\alpha_3)(\beta_1,\beta_2\beta_3) = (\alpha_1\beta_1,\alpha_2\beta_2,\alpha_3\beta_3)$. When α and β are finite this definition of $\alpha\|\beta$ coincides with the inductive definition given earlier. Then we define a fair interleaving operator on closed sets of traces by:

$$T_1\|T_2 = \bigcup\{\alpha_1\|\alpha_2 \mid \alpha_1 \in T_1 \ \& \ \alpha_2 \in T_2\}^\dagger.$$

With these definitions in hand, we can define $\mathcal{T}$ denotationally. Apart from the above modifications to $T_1;T_2$ (and therefore also T^*) and $T_1\|T_2$, the only change in the semantic clauses concerns the meaning of a loop. We give details only for the coarse-grained case; the corresponding fine-grained version is obtainable similarly.

Definition 9.1 The fair transition traces semantic function $\mathcal{T} : \mathbf{Com} \to \mathcal{P}^\dagger(\Sigma^\infty)$ is defined by the following clauses:

$$\begin{array}{l}
\mathcal{T}[\![\mathbf{skip}]\!] = \{(s,s) \mid s \in S\}^\dagger \\
\mathcal{T}[\![I{:=}E]\!] = \{(s,[s \mid I = n]) \mid (s,n) \in \mathcal{E}[\![E]\!]\}^\dagger \\
\mathcal{T}[\![C_1;C_2]\!] = \mathcal{T}[\![C_1]\!];\mathcal{T}[\![C_2]\!] \\
\mathcal{T}[\![C_1\|C_2]\!] = \mathcal{T}[\![C_1]\!]\|\mathcal{T}[\![C_2]\!] \\
\mathcal{T}[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!] = \mathcal{T}[\![B]\!];\mathcal{T}[\![C_1]\!] \cup \mathcal{T}[\![\neg B]\!];\mathcal{T}[\![C_2]\!] \\
\mathcal{T}[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = (\mathcal{T}[\![B]\!];\mathcal{T}[\![C]\!])^*;\mathcal{T}[\![\neg B]\!] \cup (\mathcal{T}[\![B]\!];\mathcal{T}[\![C]\!])^\omega \\
\mathcal{T}[\![\mathbf{await}\ B\ \mathbf{then}\ C]\!] = \{(s,s') \in \mathcal{T}[\![C]\!] \mid (s,s) \in \mathcal{T}[\![B]\!]\}^\dagger
\end{array}$$

•

Yet again all operations on trace sets used in this semantics are monotone (even continuous) with respect to set inclusion. However, the least fixed point characterization for loop semantics no longer applies. Instead, the loop semantics corresponds to what might be called an "operational fixed point" of the function $\lambda T.(\mathcal{T}[\![B]\!];\mathcal{T}[\![C]\!];T \ \cup \ \mathcal{T}[\![\neg B]\!])$.

We now need a notion of behavior that takes into account the possibility of non-termination. We therefore introduce a pseudo-state $\perp$ to represent non-termination, and let $S_\perp = S \cup \{\perp\}$.

[4]Note that since ϵ is not a member of T there is no need to define what ϵ^ω means.

Definition 9.2 The strong correctness behavior function $\mathcal{M} : \mathbf{Com} \to \mathcal{P}(S \times S_\perp)$ is given by:

$$\begin{aligned}\mathcal{M}[\![C]\!] = \{(s,s') \mid \langle C,s\rangle \to^* \langle C',s'\rangle term\} \cup \\ \{(s,\perp) \mid \langle C,s\rangle \to^\omega\},\end{aligned}$$

where $\langle C,s\rangle \to^\omega$ means that there is an infinite fair computation of C starting from s. •

This behavior function can also be obtained from the trace semantics, since $\langle C,s\rangle \to^\omega$ holds if and only if C has an infinite interference-free trace starting from s.

Proposition 9.3 *For all commands C,*

$$\begin{aligned}\mathcal{M}[\![C]\!] = \ & \{(s,s') \mid (s,s') \in \mathcal{T}[\![C]\!]\} \cup \\ & \{(s,\perp) \mid (s,s_1)(s_1,s_2)\ldots(s_n,s_{n+1})\ldots \in \mathcal{T}[\![C]\!]\}.\end{aligned}$$

Proposition 9.4 *The fair trace semantics is fully abstract with respect to strong correctness: for all commands C and C', $C \sqsubseteq_{\mathcal{T}} C' \iff C \leq_{\mathcal{M}} C'$.*

Proof: Similar to that of Proposition 6.1, extended to deal with infinite traces. The most difficult part is to show that when α is an *infinite* trace of C that is not also a trace of C', there is some finite prefix β of α such that the behavior of C "after β" is distinguishable from the behavior of C' "after β". The proof of this *finite distinguishability* property uses König's Lemma and the fact that for any command C and any pair of states s and s' the set of C'' such $\langle C,s\rangle \to^* \langle C'',s'\rangle$ is finite. ∎

The laws given in Figure 2 continue to hold for the fair trace semantics, except that the inequation

$$C_1;(C_2 \| C) \sqsubseteq (C_1;C_2)\|C$$

may fail if C_1 has infinite traces. Nevertheless, the inequation still holds if C_1 is loop-free. Note that the fair trace semantics no longer identifies **await false then skip** with **while true do skip**, since the former denotes the empty set and the latter denotes the set of all infinite stuttering sequences.

10 Total correctness

We remarked earlier that the finite trace semantics for a loop **while** B **do** C has an equivalent formulation as the least fixed point of the function

$$\lambda T.(\mathcal{T}[\![B]\!];\mathcal{T}[\![C]\!];T \ \cup \ \mathcal{T}[\![\neg B]\!]).$$

In the fair trace semantics, the loop's meaning is still a fixed point of this functional, but not the least. For instance, the loop **while true do skip** has for its fair traces all infinite stuttering sequences, whereas in the least fixed point semantics this loop denotes the empty set. This example also shows that the fair trace semantics does not correspond to the use of the *greatest* fixed point either. There is, therefore, a third form of semantics, obtained by using greatest fixed points in the semantic clause for loops. Under this semantics the above loop has *all* possible traces.

The trace sets constructed in this semantics enjoy a further closure property in addition to stuttering and mumbling:

- if $\alpha\beta \in \mathcal{T}[\![C]\!]$ and $\beta \in \Sigma^\omega$ is interference-free, then for all $\gamma \in \Sigma^\infty$ we also have $\alpha\gamma \in \mathcal{T}[\![C]\!]$.

We call this "closure under chattering". This closure property has the effect of identifying all commands that may fail to terminate. This form of trace semantics is fully abstract with respect to *total correctness* behavior, defined by

$$\begin{aligned}\mathcal{M}[\![C]\!] = \{(s,s') \mid \langle C,s\rangle \to^* \langle C',s'\rangle \mathit{term}\} \cup \\ \{(s,s') \mid \langle C,s\rangle \to^\omega \ \ \& \ s' \in S_\perp\}.\end{aligned}$$

11 Robustness

The full abstraction results given above relied only on certain general properties: monotonicity of the semantic definitions, compositionality, finite distinguishability, and the fact that the behavior of a program is embedded in its trace set. We can therefore extend these results to deal with any additional program constructs that do not violate these properties[5]. For instance, we may add a non-deterministic choice construct C_1 **or** C_2, with operational semantics given by:

$$\begin{aligned}\langle C_1 \textbf{ or } C_2, s\rangle &\to \langle C_1, s\rangle \\ \langle C_1 \textbf{ or } C_2, s\rangle &\to \langle C_2, s\rangle.\end{aligned}$$

Then $\mathcal{T}[\![C_1 \textbf{ or } C_2]\!] = \mathcal{T}[\![C_1]\!] \cup \mathcal{T}[\![C_2]\!]$, and all of the previous development goes through with minor modifications. The semantics is still fully abstract, and the laws of programming given earlier continue to hold. In addition, $C \sqsubseteq C'$ if and only if $(C \textbf{ or } C') = C'$, **or** is idempotent, commutative and associative, and **or** distributes through sequential and parallel composition.

The coarse-grained semantics satisfies the law

$$I_1{:=}E_1 \| I_2{:=}E_2 \equiv (I_1{:=}E_1; I_2{:=}E_2) \textbf{ or } (I_2{:=}E_2; I_1{:=}E_1),$$

but this fails in the fine-grained case: for example, when assignment is not atomic the parallel command $x{:=}x+1 \| x{:=}x+1$ has the trace $([x=0],[x=1])$, and this is not a trace of $x{:=}x+1; x{:=}x+1$.

12 Summary and Conclusions

We have introduced transition traces and used them as the basis for a variety of fully abstract semantics for a shared variable parallel programming language. Our results apply in coarse- and fine-grained versions to yield full abstraction with respect to three forms of program behavior: partial, strong, and total correctness. In each case, extra language features may be added without invalidating full abstraction, provided certain general semantic properties are preserved; in particular, the trace semantics of the new features must be definable compositionally and monotonically. This shows the flexibility and generality of our ideas and results.

Program constructs or operational assumptions (such as fairness) that give rise to unbounded nondeterminism do not appear to cause severe semantic problems in this framework. For instance, it is almost trivial to add a random assignment command $I{:=}?$ to the syntax, with the following semantics:

$$\mathcal{T}[\![I{:=}?]\!] = \{(s,[s \mid I = n]) \mid s \in S \ \& \ n \in N\}^\dagger.$$

This would not affect the validity of any of our results.

---

[5]Of course, the coroutine construct C_1 **co** C_2 from Hennessy-Plotkin cannot be handled by our semantics, since $\mathcal{T}[\![C_1 \textbf{ co } C_2]\!]$ cannot be determined from $\mathcal{T}[\![C_1]\!]$ and $\mathcal{T}[\![C_2]\!]$.

It is interesting to compare our results with the work of Apt and Plotkin [2], who proved that for a sequential while-loop language with random assignment there is no denotational continuous least fixed point semantics that is fully abstract with respect to strong correctness. Our fair trace model provides a denotational continuous semantics for a parallel version of this language, and is fully abstract for strong correctness; but this is not a least fixed point semantics. The corresponding least fixed point semantics is fully abstract for partial correctness, and the corresponding greatest fixed point semantics is fully abstract for total correctness. For the sequential language there is no need to use traces to achieve full abstraction, as the behavior functions can be defined compositionally. When our definitions are adapted to the sequential setting they yield three fully abstract semantics for the Apt-Plotkin language, with respect to partial, strong, and total correctness respectively, again corresponding to the three interpretations of while-loops.

We plan further research into the use of transition trace semantics. In particular, with appropriate adjustments to represent deadlock, we can give a deadlock-sensitive transition trace semantics that can be used to reason about deadlock-freedom.

References

[1] M. Abadi and G. D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, June 1993.

[2] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *JACM*, 33(4):724–767, October 1986.

[3] F. de Boer, J. Kok, C. Palamidessi, and J. Rutten. The failure of failures in a paradigm for asynchronous communication. In J. Baeten and J. Groote, editors, *Concur'91*, number 527 in Lecture Notes in Computer Science, pages 111–126. Springer-Verlag, 1991.

[4] N. Francez. *Fairness*. Springer-Verlag, 1986.

[5] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer Verlag, 1979.

[6] E. Horita, J. de Bakker, and J. Rutten. Fully abstract denotational models for nonuniform concurrent languages, June 1990. Technical Report CS-R9027, Centre for Mathematics and Computer Science, Amsterdam.

[7] B. Jonsson. A fully abstract trace semantics for dataflow networks. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[8] R. M. Keller and P. Panangaden. Semantics of digital networks containing indeterminate operators. *Distributed Computing*, 1(4):235–245, 1986.

[9] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9^{th} World Congress*. IFIP, North Holland, September 1983.

[10] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[11] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[12] D. Park. On the semantics of fair parallelism. In D. Bjørner, editor, *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*, pages 504–526. Springer-Verlag, 1979.

[13] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[14] J. R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proceedings of the 30^{th} Annual Symposium on Foundations of Computer Science*, pages 170–177. IEEE Press, 1989.

[15] A. Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman, 1988.