Reprinted from INFORMATION AND COMPUTATION

Vol. 127, No. 2, June 15, 1996
Printed in Belgium

Full Abstraction for a Shared-Variable Parallel Language

STEPHEN BROOKES

School of Computer Science, Carnegie–Mellon University, Schenley Park, Pittsburgh, Pennsylvania 15213

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 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. © 1996 Academic Press, Inc.

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*, *deadlock-freedom*, and more general *liveness* and *safety* properties [14]. 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 said to be *equationally fully abstract* with respect to the given notion of behavior [17, 20, 22]. 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 said to be *inequationally fully abstract* with respect to the given notion of program behavior and approximation. Clearly an inequationally fully abstract semantics is also equationally fully abstract. Intuitively, a fully abstract semantics is at exactly the right level of abstraction to support compositional reasoning.

The difficulty of finding fully abstract semantics is well known [4, 17, 20, 22]. 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 [9, 18], 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. It is not even sufficient to model a program as a set of *sequences* of states, each sequence recording a possible execution history of the program, since this semantics still fails to be compositional. 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 [9] 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

0890-5401/96 $18.00

command executing in parallel. The problem remained of finding a fully abstract model for the original parallel language.

It is often reasonable to assume that program execution is *fair*, in that commands executing in parallel are not artificially prevented from making progress. In particular, execution is *weakly fair* if whenever a parallel command is continuously enabled it eventually gets to move [16, 8]. Even when parallel execution is being simulated on a shared uniprocessor, using a scheduler that interleaves the activities of parallel commands, it is easy to implement weak fairness using a round-robin scheduling strategy. The assumption of (weak) fairness allows us to abstract away from the details of any particular scheduling strategy and from the relative speeds of parallel processes. It is well known that fair execution gives rise to the phenomenon of unbounded nondeterminism—there are parallel programs guaranteed to terminate under a fair execution strategy but which may terminate in any of a (countably) infinite set of final states. The interaction of unbounded non-determinism with powerdomain constructions is problematic. For instance it is difficult to see how to adapt the Hennessy–Plotkin resumptions model to incorporate fairness; moreover in Apt and Plotkin's semantics [4], unbounded non-determinism causes lack of continuity of certain key functions. Park [19] provided a denotational semantics for shared-variable programs, including a formal treatment of fairness, but failed to achieve full abstraction for a familiar reason: Park's semantics distinguished between **skip** and **skip**; **skip**. This left open the problem of finding a fully abstract model for shared-variable programs with respect to a suitable notion of fair behavior.

In this paper we solve these problems. We first describe a new denotational semantics for shared-variable programs, and we show that it is fully abstract with respect to partial correctness behavior. We do not need to add a coroutine construct to the language in order to achieve full abstraction; instead, our proof makes essential use of a synchronization primitive (the "conditional critical region" construct) already present in the language. 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 [15]) 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 also show how to incorporate fairness into our semantic framework in a natural way, by including infinite transition traces representing fair computations. In doing this we build on the foundations laid by Park [19], with the key extra ingredient supplied by our use of *closed* trace sets. All operations on trace sets used in our semantic definitions are continuous, with respect to set inclusion. We prove full abstraction with respect to the appropriate notion of behavior, termed "strong correctness," in which both termination and divergence (non-termination) are regarded as observable. Our semantics may therefore be used to reason about total correctness, partial correctness, safety and liveness properties of parallel programs executing fairly.

We also show that our semantic models are 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.

2. SYNTAX

We discuss a standard shared-variable parallel language, as in [9, 18]. 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:

$$C ::= \textbf{skip} \mid I := E \mid C_1; C_2 \mid \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2 \mid$$
$$\textbf{while } B \textbf{ do } C \mid C_1 \parallel C_2 \mid \textbf{await } B \textbf{ then } C.$$

A command of the form **await** B **then** C is a *conditional critical region*,[1] 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**).[2]

We write $\text{free}[\![E]\!]$ to denote the set of identifiers occurring free in E, and similarly for boolean expressions. For commands we define

$$\text{free}[\![\textbf{skip}]\!] = \{\ \}$$
$$\text{free}[\![I := E]\!] = \{I\} \cup \text{free}[\![E]\!]$$
$$\text{free}[\![C_1; C_2]\!] = \text{free}[\![C_1]\!] \cup \text{free}[\![C_2]\!]$$

[1] Also known as a "conditional atomic action" [3].

[2] This restriction is motivated by pragmatic concerns: it is easy and inexpensive for a scheduler to suspend execution of other parallel commands while a finite sequence of assignments is performed, but not practical to suspend while an arbitrary command executes, since a loop may take an unbounded amount of time. Moreover, the restriction does not hamper the utility of the programming language and suffices for establishing full abstraction.

$\text{free}[\![\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2]\!] = \text{free}[\![B]\!] \cup \text{free}[\![C_1]\!] \cup \text{free}[\![C_2]\!]$

$\text{free}[\![\textbf{while } B \textbf{ do } C]\!] = \text{free}[\![B]\!] \cup \text{free}[\![C]\!]$

$\text{free}[\![\textbf{await } B \textbf{ then } C]\!] = \text{free}[\![B]\!] \cup \text{free}[\![C]\!]$

$\text{free}[\![C_1 \parallel C_2]\!] = \text{free}[\![C_1]\!] \cup \text{free}[\![C_2]\!]$.

3. OPERATIONAL SEMANTICS

We present a structural operational semantics similar to the semantics given in [9].

We use N for the set of (non-negative) integers, ranged over by n; and $V = \{\texttt{tt}, \texttt{ff}\}$ for the set of truth values, ranged over by v. A state is a finite partial function from identifiers to integer values. Let $S = \textbf{Ide} \to_p N$ denote the set of states, ranged over by s. We write $\text{dom}(s)$ 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 $\langle E, s\rangle \to^* n$ 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 $\mathscr{E}$ and $\mathscr{B}$, characterized operationally by

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

For command execution we specify a set of configurations

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

a transition relation $\to \subseteq \textbf{Conf} \times \textbf{Conf}$, and a subset of *successfully terminated* configurations. A configuration of the form $\langle C, s\rangle$ will represent a stage in a computation at which the remaining command to be executed is C, and the current state is s. A transition of the form $\langle C, s\rangle \to \langle C', s'\rangle$ represents a computation step in which an atomic step of C enabled in state s is performed, causing the state to change to s' and after which the remaining command to be executed is C'. The transition relation for commands is defined to be the smallest relation satisfying the syntax-directed transition rules given in Fig. 1. This means that a transition is possible if and only if it can be deduced from these rules. The successfully terminated configurations are those for which $\langle C, s\rangle$ *term* is provable from the rules in Fig. 1.

Note how the rules specify the atomicity of boolean expression evaluation, assignment, and conditional critical regions, by means of $\to^*$, the reflexive transitive closure of the transition relation. The transition rules for $C_1 \parallel C_2$ clearly specify the interleaving of atomic actions performed by C_1 with those performed by C_2, and we have specified that a parallel composition terminates only when both components have terminated.

$$\langle \textbf{skip}, s\rangle \textit{term}$$

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

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

$$\frac{\langle B, s\rangle \to^* \texttt{tt}}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \to \langle C_1, s\rangle}$$

$$\frac{\langle B, s\rangle \to^* \texttt{ff}}{\langle \textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \to \langle C_2, s\rangle}$$

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

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

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

$$\frac{\langle B, s\rangle \to^* \texttt{tt} \quad \langle C, s\rangle \to^* \langle C', s'\rangle \textit{term}}{\langle \textbf{await } B \textbf{ then } C, s\rangle \to \langle C', s'\rangle}$$

$$\frac{\langle B, s\rangle \to^* \texttt{ff}}{\langle \textbf{await } B \textbf{ then } C, s\rangle \to \langle \textbf{await } B \textbf{ then } C, s\rangle}$$

FIG. 1. Transition rules for commands.

A configuration like $\langle \textbf{await } B \textbf{ then } C, s\rangle$ is said to be "blocked" if B is false in s; we model blocking operationally by means of a "busy-wait" or "idle" transition, as expressed in the final rule of Fig. 1.[3] More generally a parallel command is said to be blocked in a given state if each of its component commands is blocked in this manner; thus C is blocked in state s iff the only possible transition of $\langle C, s\rangle$ is an "idle" transition back to itself.

A *computation* of a command C from state s is a maximal sequence of consecutive transitions starting from $\langle C, s\rangle$. A computation is either finite, ending in a successfully terminated configuration, or infinite.

4. PROGRAM BEHAVIOR

We will focus initially on two closely related notions of behavior, each determined by the finite computations of a program.

[3] An alternative interpretation of blocking is obtained if we omit this transition rule: a blocked configuration would then have no enabled transition. We would then need to distinguish between two kinds of finite computation (successful and blocking). This would greatly complicate the presentation of later definitions and results, without gaining any generality.

DEFINITION 4.1. The *partial correctness* behavior function $\mathscr{M}$: $\mathbf{Com} \to \mathscr{P}(S \times S)$ is defined by

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

A Hoare-style partial correctness assertion $\{P\}\ C\{Q\}$, where P and Q are boolean-valued expressions involving identifiers, is valid if every terminating computation of C from a state satisfying P ends in a state satisfying Q. As usual, P and Q are "conditions" drawn from some specification language. Assuming that the condition language contains at least expressions of the form

$$x_1 = n_1 \,\&\, \cdots \,\&\, x_k = n_k,$$

where the x_i are identifiers and the n_i are numerals, it is easy to see that two programs have the same partial correctness behavior if and only if they satisfy the same Hoare-style assertions.

The partial correctness behavior $\mathscr{M}$ ignores all intermediate states that arise during a computation, focussing only on the first and last states of successful computations. We now present a simple generalization. A (finite) *state trace* of C is a sequence of states occurring during a successful computation of C. Let S^+ denote the set of non-empty finite sequences of states.

DEFINITION 4.2. The (finite) *state trace* behavior function $\mathscr{S}$: $\mathbf{Com} \to \mathscr{P}(S^+)$ is given by

$$\mathscr{S}[\![C]\!] = \{s_0 s_1 \ldots s_k \mid \langle C, s_0\rangle \to^* \langle C_1, s_1\rangle \to^* \cdots \to^* \langle C_k, s_k\rangle \ term\}.$$

Partial correctness behavior induces a preorder $\sqsubseteq_{\mathscr{M}}$ and an equivalence relation $\equiv_{\mathscr{M}}$ on commands,

$$C \sqsubseteq_{\mathscr{M}} C' \Leftrightarrow \forall s.(\text{free}[\![C]\!] \cup \text{free}][\![C']\!] \subseteq \text{dom}(s) \Rightarrow \mathscr{M}[\![C]\!]s \subseteq \mathscr{M}[\![C']\!]s)$$

$$C \equiv_{\mathscr{M}} C' \Leftrightarrow C \sqsubseteq_{\mathscr{M}} C' \,\&\, C' \sqsubseteq_{\mathscr{M}} C,$$

where we write $\mathscr{M}[\![C]\!]s$ for $\{s' \mid (s, s') \in \mathscr{M}[\![C]\!]\}$.

Similarly we define the state trace preorder and equivalence relation on commands:

$$C \sqsubseteq_{\mathscr{S}} C' \Leftrightarrow \forall s.(\text{free}[\![C]\!] \cup \text{free}][\![C']\!] \subseteq \text{dom}(s) \Rightarrow \mathscr{S}[\![C]\!]s \subseteq \mathscr{S}[\![C']\!]s)$$

$$C \equiv_{\mathscr{S}} C' \Leftrightarrow C \sqsubseteq_{\mathscr{S}} C' \,\&\, C' \sqsubseteq_{\mathscr{S}} C,$$

where $\mathscr{S}[\![C]\!]s = \{s_1 \ldots s_k \mid ss_1 \ldots s_k \in \mathscr{S}[\![C]\!]\}$.

None of these relations is *substitutive*, since we have:

$$x := 1; x := x + 1 \equiv_{\mathscr{M}} x := 2$$

$$(x := 1; x := x + 1) \parallel x := 2 \not\equiv_{\mathscr{M}} x := 2 \parallel x := 2$$

$$x := 1; x := x + 1 \equiv_{\mathscr{S}} x := 1; x := 2$$

$$(x := 1; x := x + 1) \parallel x := 2 \not\equiv_{\mathscr{S}} (x := 1; x := 2) \parallel x := 2.$$

We therefore define the substitutive preorders $\leqslant_{\mathscr{M}}$ and $\leqslant_{\mathscr{S}}$, and the substitutive equivalence relations $=_{\mathscr{M}}$ and $=_{\mathscr{S}}$ by

$$C \leqslant_{\mathscr{M}} C' \Leftrightarrow \forall P[-].(P[C] \sqsubseteq_{\mathscr{M}} P[C'])$$

$$C =_{\mathscr{M}} C' \Leftrightarrow C \leqslant_{\mathscr{M}} C' \,\&\, C' \leqslant_{\mathscr{M}} C$$

$$C \leqslant_{\mathscr{S}} C' \Leftrightarrow \forall P[-].(P(C] \sqsubseteq_{\mathscr{S}} P[C'])$$

$$C =_{\mathscr{S}} C' \Leftrightarrow C \leqslant_{\mathscr{S}} C' \,\&\, C' \leqslant_{\mathscr{S}} C,$$

where $P[-]$ ranges over program contexts, that is, programs with a hole (denoted $[-]$) into which a command may be substituted; and $P[C]$ denotes the program obtained by substituting C into the hole. Thus $C =_{\mathscr{M}} C'$ if and only if C and C' are interchangeable in all program contexts without affecting partial correctness. By definition, $\leqslant_{\mathscr{M}}$ and $=_{\mathscr{M}}$ are *congruences*, since they are preserved by all programming language constructs.

Since $\mathscr{M}[\![C]\!] = \{(s, s') \mid ss' \in \mathscr{S}[\![C]\!]\}$ it is easy to see that $C \sqsubseteq_{\mathscr{S}} C' \Rightarrow C \sqsubseteq_{\mathscr{M}} C'$, and similarly state trace equivalence implies partial correctness equivalence. The converse fails, as shown by

$$x := 1; x := x + 1 \not\equiv_{\mathscr{S}} x := 2$$

$$x := 1; x := x + 1 \equiv_{\mathscr{M}} x := 2.$$

Nevertheless the substitutive preorders $\leqslant_{\mathscr{M}}$ and $\leqslant_{\mathscr{S}}$ coincide, so that $=_{\mathscr{M}}$ and $=_{\mathscr{S}}$ also coincide; two commands induce the same state traces in every context if and only if they induce the same partial correctness behavior in every context.

PROPOSITION 4.3. *For all commands C and C', $C \leqslant_{\mathscr{M}} C'$ if and only if $C \leqslant_{\mathscr{S}} C'$.*

Proof. Since $C \sqsubseteq_{\mathscr{S}} C'$ implies $C \sqsubseteq_{\mathscr{M}} C'$ it follows that $C \leqslant_{\mathscr{S}} C'$ implies $C \leqslant_{\mathscr{M}} C'$. For the converse we argue as follows.

Assume $C \not\leqslant_{\mathscr{S}} C'$. Then there is a context $P[-]$ and a state trace $\sigma = s_0 \ldots s_k$ such that $\sigma \in \mathscr{S}[\![P[C]]\!]$ but $\sigma \notin \mathscr{S}[\![P[C']]\!]$. Since states are finite, for each state s there is a boolean expression IS_s such that for all states s' such that $\text{dom}(s') \supseteq \text{dom}(s)$, $\langle \text{IS}_s, s'\rangle \to^* \texttt{tt}$ if s' and s agree on

dom(s), and $\langle \mathrm{IS}_s, s' \rangle \to^*$ ff otherwise. Then the program context

$$Q[-] = P[-] \parallel (\textbf{await } \mathrm{IS}_{s_0} \textbf{ then skip}; \ldots; \textbf{await } \mathrm{IS}_{s_k} \textbf{ then skip})$$

has the property that $(s_0, s_k) \in \mathscr{M}[\![Q[C]]\!]$ but $(s_0, s_k) \notin \mathscr{M}[\![Q[C']]\!]$. Thus $C \not\leq_{\mathscr{M}} C'$. ∎

As a corollary, a semantics will be fully abstract with respect to state traces if and only if it is fully abstract with respect to partial correctness.

5. RESUMPTION SEMANTICS

Hennessy and Plotkin [9] gave a denotational semantics based on a domain R of "resumptions," defined recursively by the domain equation

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

where $\mathscr{P}$ is a suitable powerdomain constructor, $+$ denotes the separated sum and $\times$ denotes the cartesian product of domains. The resumption semantic function $\mathscr{R}$: **Com** $\to R$ is characterized operationally by the following property:

$$\mathscr{R}[\![C]\!]s = \{s \mid \langle C, s \rangle \; term\} \cup \{(\mathscr{R}[\![C']\!], s') \mid \langle C, s \rangle \to \langle C', s' \rangle\}.$$

This shows clearly how resumptions model intermediate state changes. The resumption semantic function can also be given a denotational definition, since $\mathscr{R}[\![C_1 \parallel C_2]\!]$ can be determined from $\mathscr{R}[\![C_1]\!]$ and $\mathscr{R}[\![C_2]\!]$. Moreover, $\mathscr{M}[\![C]\!]$ can be be extracted from $\mathscr{R}[\![C]\!]$ by an "unravelling" operation, as described in [9].

However, the resumption 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. To attain full abstraction Hennessy and Plotkin added a form of "coroutine" composition C_1 **co** C_2 to the syntax of the programming language, together with a nondeterministic 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. The transition rules for C_1 **co** C_2 are

$$\frac{\langle C_1, s \rangle \to \langle C_1', s' \rangle}{\langle C_{11} \textbf{ co } C_2, s \rangle \to \langle C_2 \textbf{ co } C_1', s' \rangle}$$

$$\frac{\langle C_1, s \rangle \; term}{\langle C_1 \textbf{ co } C_2, s \rangle \; term}.$$

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 difficult to motivate. The full abstraction problem for the original language remained open.

6. 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 (finite) *transition trace* of a command C is defined to be a sequence of pairs of states $(s_0, s_0')(s_1, s_1') \cdots (s_k, s_k')$ such that it is possible for C to perform a computation from s_0 to s_k' if execution is interrupted k times, the ith 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, so that the trace corresponds to a computation in which the state is never changed by the command's environment. 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 $\mathscr{T}[\![C]\!]$ for the set of transition traces of C, characterized operationally by

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

Clearly $\mathscr{M}$ and $\mathscr{S}$ can be extracted from $\mathscr{T}$, in the following sense: for all commands C,

$$\mathscr{M}[\![C]\!] = \{(s, s') \mid (s, s') \in \mathscr{T}[\![C]\!]\}$$

$$\mathscr{S}[\![C]\!] = \{s_0 s_1 \ldots s_k \mid (s_0, s_1)(s_1, s_2) \ldots (s_{k-1}, s_k) \in \mathscr{T}[\![C]\!]\}.$$

The above operational characterization of $\mathscr{T}$ has obvious but important consequences.

PROPOSITION 6.1. *The set of transition traces of a command C is closed under "stuttering" and "mumbling": for all $\alpha, \beta \in (S \times S)^*$ and all $s, s', s'' \in S$,*

$$\alpha\beta \in \mathscr{T}[\![C]\!] \Rightarrow \alpha(s, s)\,\beta \in \mathscr{T}[\![C]\!]$$

$$\alpha(s, s')(s', s'')\beta \in \mathscr{T}[\![C]\!] \Rightarrow \alpha(s, s'')\beta \in \mathscr{T}[\![C]\!].$$

The first closure condition corresponds to reflexivity of $\rightarrow^*$, the second to transitivity. Since a stutter introduces a repetition and a mumble absorbs an intermediate state, the terminology is closely related to everyday usage.[4] 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 $T = T^\dagger$. Proposition 6.1 thus states that $\mathcal{T}[\![C]\!]$ 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. We will show that the trace semantic function $\mathcal{T}$: **Com** $\rightarrow$ $\mathcal{P}^\dagger(\Sigma^+)$, characterized operationally above, can be defined compositionally.

Since **skip** causes no state change, no matter how many times it is interrupted, it is easy to see that the traces of **skip** are finite sequences of stuttering steps:

$$\begin{aligned}\mathcal{T}[\![\textbf{skip}]\!] &= \{(s_0, s_0)\ldots(s_k, s_k) \mid k \geqslant 0 \;\&\\ &\qquad \forall i.(0 \leqslant i \leqslant k \Rightarrow s_i \in S)\}\\ &= \{(s, s) \mid s \in \mathcal{S}\}^\dagger.\end{aligned}$$

Similarly, since we assume that assignments and conditional critical regions are atomic,

$$\mathcal{T}[\![I := E]\!] = \{(s, [s \mid I = n]) \mid (s, n) \in \mathcal{E}[\![E]\!]\}^\dagger$$

$$\begin{aligned}&\mathcal{T}[\![\textbf{await } B \textbf{ then } C]\!]\\ &\quad = \{(s, s') \mid (s, \mathtt{tt}) \in \mathcal{B}[\![B]\!] \;\&\; (s, s') \in \mathcal{T}[\![C]\!]\}^\dagger.\end{aligned}$$

The traces of $C_1; C_2$ are built by concatenation. When T_1 and T_2 are closed sets of traces we define

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

denoting the smallest closed set of traces containing all relevant concatenations; the closure operator accounts for the possibility of mumbling between the end of a trace in T_1 and the start of a trace in T_2. It then follows that $\mathcal{T}[\![C_1; C_2]\!] = \mathcal{T}[\![C_1]\!]; \mathcal{T}[\![C_2]\!]$.

[4] We should note, however, that our use of the term "stuttering" differs from Lamport [15]. In Lamport's setting a program denotes a set of sequences of states, and a set $X \subseteq \mathcal{P}(S^+)$ is said to be closed under stuttering if for all ρ, ρ' in S^* and all $s \in S$,

$$\rho s s \rho' \in X \Leftrightarrow \rho s s s \rho' \in X.$$

The implication from left to right corresponds to reflexivity of $\rightarrow^*$, and the reverse implication to transitivity. Thus in Lamport's terminology "stuttering" combines repetition of a state and absorption of an intermediate state.

The traces of a while-loop **while** B **do** C are obtained by closure from traces of form

$$\beta_1\gamma_1\beta_2\gamma_2\ldots\beta_k\gamma_k\alpha,$$

where $k \geqslant 0$, each β_i corresponds to an evaluation of B to $\mathtt{tt}$, each γ_i is a trace of C, and α corresponds to an evaluation of B to $\mathtt{ff}$. We therefore 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,[5] closed under stuttering, mumbling and concatenation. It then follows that

$$\mathcal{T}[\![\textbf{while } B \textbf{ do } C]\!] = (\mathcal{T}[\![B]\!]; \mathcal{T}[\![C]\!])^*; \mathcal{T}[\![\neg B]\!],$$

where we write $\mathcal{T}[\![B]\!] = \{(s, s) \mid (s, \mathtt{tt}) \in \mathcal{B}[\![B]\!]\}^\dagger$. Similarly we have

$$\begin{aligned}&\mathcal{T}[\![\textbf{if } B \textbf{ then } C_1 \textbf{ then } C_2]\!]\\ &\quad = \mathcal{T}[\![B]\!]; \mathcal{T}[\![C_1]\!] \cup \mathcal{T}[\![\neg B]\!]; \mathcal{T}[\![C_2]\!],\end{aligned}$$

writing $\mathcal{T}[\![\neg B]\!]$ for $\{(s, s) \mid (s, \mathtt{ff}) \in \mathcal{B}[\![B]\!]\}^\dagger$.

The transition traces of $C_1 \parallel C_2$ are built by interleaving. For α and β in Σ^* let $\alpha \parallel \beta$ be the set of all transition traces built by interleaving α with β. This may be defined inductively as follows:

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

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

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

It then follows that $\mathcal{T}[\![C_1 \parallel C_2]\!] = \mathcal{T}[\![C_1]\!] \parallel \mathcal{T}[\![C_2]\!]$.

We summarize these results in the following denotational description of $\mathcal{T}$.

PROPOSITION 6.2. *The (finite) transition traces semantic function* $\mathcal{T}$: **Com** $\rightarrow \mathcal{P}^\dagger(\Sigma^+)$ *is characterized uniquely by the following clauses*:

$$\begin{aligned}&\mathcal{T}[\![\textbf{skip}]\!] = \{(s, s) \mid s \in S\}^\dagger\\ &\mathcal{T}[\![I := E]\!] = \{(s, [s \mid I = n]) \mid (s, n) \in \mathcal{E}[\![E]\!]\}^\dagger\\ &\mathcal{T}[\![C_1; C_2]\!] = \mathcal{T}[\![C_1]\!]; \mathcal{T}[\![C_2]\!]\end{aligned}$$

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

$$\mathscr{T}[\![\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2]\!]$$
$$= \mathscr{T}[\![B]\!]; \mathscr{T}[\![C_1]\!] \cup \mathscr{T}[\![\neg B]\!]; \mathscr{T}[\![C_2]\!]$$
$$\mathscr{T}[\![\textbf{while } B \textbf{ do } C]\!] = (\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!])^*; \mathscr{T}[\![\neg B]\!]$$
$$\mathscr{T}[\![C_1 \parallel C_2]\!] = \mathscr{T}[\![C_1]\!] \parallel \mathscr{T}[\![C_2]\!]$$
$$\mathscr{T}[\![\textbf{await } B \textbf{ then } C]\!] = \{(s, s') \in \mathscr{T}[\![C]\!] \mid (s, s) \in \mathscr{T}[\![B]\!]\}^\dagger.$$

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. For each boolean expression B and command C, the function F on $\mathscr{P}^\dagger(\Sigma^+)$ given by $F(T) = (\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!]; T \cup \mathscr{T}[\![\neg B]\!])$ is continuous. Its least fixed point is therefore

$$\mu T.F(T) = \bigcup_{n=0}^{\infty} F^n\{\,\}$$
$$= (\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!])^*; \mathscr{T}[\![\neg B]\!]$$
$$= \mathscr{T}[\![\textbf{while } B \textbf{ do } C]\!].$$

It is also worth noting that the loop semantics can be obtained by taking the least fixed point of the analogous operation on (not necessarily closed) sets of traces and then taking the closure. This can be made precise as follows. For (arbitrary) trace sets T_1 and T_2 let $T_1 \cdot T_2 = \{\alpha\beta \mid \alpha \in T_1 \,\&\, \beta \in T_2\}$, so that $T_1; T_2 = (T_1 \cdot T_2)^\dagger$. Clearly the collection of (arbitrary) trace sets, ordered by inclusion, also forms a complete lattice. The function $F'(T) = (\mathscr{T}[\![B]\!] \cdot \mathscr{T}[\![C]\!] \cdot T \cup \mathscr{T}[\![\neg B]\!])$ is continuous, with least fixed point given by

$$\mu T.F'(T) = (\mathscr{T}[\![B]\!] \cdot \mathscr{T}[\![C]\!])^* \cdot \mathscr{T}[\![\neg B]\!],$$

and the closure of this set is $(\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!])^*; \mathscr{T}[\![\neg B]\!] = \mathscr{T}[\![\textbf{while } B \textbf{ do } C]\!]$.

7. 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, $\mathscr{E}$ is fully abstract for the expression sub-language, and $\mathscr{B}$ is fully abstract for the boolean expression sub-language.

We now show that the transition trace semantics for commands is fully abstract. We define $\mathscr{T}[\![C]\!]s = \{s'\alpha \mid (s, s')\alpha \in \mathscr{T}[\![C]\!]\}$ and

$$C \sqsubseteq_{\mathscr{T}} C' \Leftrightarrow \forall s.(\text{free}[\![C]\!] \cup \text{free}[\![C']\!] \subseteq \text{dom}(s) \Rightarrow \mathscr{T}[\![C]\!]s \subseteq \mathscr{T}[\![C']\!]s)$$
$$C \equiv_{\mathscr{T}} C' \Leftrightarrow C \sqsubseteq_{\mathscr{T}} C' \,\&\, C' \sqsubseteq_{\mathscr{T}} C.$$

PROPOSITION 7.1. *The transition traces semantics $\mathscr{T}$ is inequationally fully abstract: for all commands C and C', $C \sqsubseteq_{\mathscr{T}} C' \Leftrightarrow C \leqslant_{\mathscr{M}} C'$.*

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

$$\mathscr{M}[\![P[C]]\!]s = \{s' \mid (s, s') \in \mathscr{T}[\![P[C]]\!]\}$$
$$\subseteq \{s' \mid (s, s') \in \mathscr{T}[\![P[C']]\!]\}$$
$$= \mathscr{M}[\![P[C']]\!]s.$$

This shows that $C \sqsubseteq_{\mathscr{T}} C' \Leftrightarrow C \leqslant_{\mathscr{M}} C'$.

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

$$\langle \text{MAKE}_s, s' \rangle \to^* \langle \textbf{skip}, s \rangle$$

for all states such that $\text{dom}(s') = \text{dom}(s)$.

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

$$\textbf{await } \text{IS}_{s_0'} \textbf{ then } \text{MAKE}_{s_1};$$
$$\textbf{await } \text{IS}_{s_0'} \textbf{ then } \text{MAKE}_{s_2};$$
$$\ldots$$
$$\textbf{await } \text{IS}_{s_{k-1}'} \textbf{ then } \text{MAKE}_{s_k}.$$

The traces of DO_α are obtained by closure from the trace $\bar{\alpha} = (s_0', s_1)(s_1', s_2)\ldots(s_{k-1}', s_k)$. Let $P_\alpha[\,-\,]$ be the program context $[\,-\,] \parallel \text{DO}_\alpha$. Each trace of $P_\alpha[C]$ is obtained by closure from the result of interleaving a trace of C with $\bar{\alpha}$. In particular, since $\alpha \in \mathscr{T}[\![C]\!]$ it follows that the interference-free trace

$$(s_0, s_0')(s_0', s_1)(s_1, s_1')(s_1', s_2)\ldots(s_{k-1}', s_k)(s_k, s_k')$$

is possible for $P_\alpha[C]$. Since trace sets are closed under mumbling, it follows that $(s_0, s_k') \in \mathscr{M}[\![P[C]]\!]$.

However, (s_0, s'_k) cannot belong to the set $\mathscr{M}[\![P[C']\!]$, since this would require C' to have a trace α' such that interleaving α' with $\bar{\alpha}$ can yield an interference-free trace of form

$$(s_0, t_1)(t_1, t_2)(t_2, t_3)\ldots(t_{r-1}, t_r)(t_r, s'_k).$$

But it is easy to see that this is possible only if α itself is derivable from α' by stuttering and mumbling. It would then follow that $\alpha \in \mathscr{T}[\![C']\!]$, contradicting our assumption.

Thus, $C \not\sqsubseteq_{\mathscr{T}} C'$ implies $C \not\leqslant_{\mathscr{M}} C'$. 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 behavior but different transition traces: clearly

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

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

$$[\,-\,] \parallel \textbf{await } x = 1 \textbf{ then } x := 0$$

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

$$\mathscr{T}[\![x := 0]\!] \subseteq \mathscr{T}[\![x := 0;\ x := 0]\!],$$

and this inclusion is proper. The 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 **await** $x = 0$ **then** $x := 1$.

8. LAWS OF PARALLEL PROGRAMMING

We can use trace semantics to prove equations and inequations between programs, with the guarantee that these laws may be used safely for reasoning about partial correctness, or state traces, in any program context. The following Proposition summarizes some laws, writing $\equiv$ for $\equiv_{\mathscr{T}}$ (or, equivalently, for $=_{\mathscr{M}}$ or $=_{\mathscr{S}}$) and $\sqsubseteq$ for $\sqsubseteq_{\mathscr{T}}$ (or, equivalently, for $\leqslant_{\mathscr{M}}$ or $\leqslant_{\mathscr{S}}$).

PROPOSITION 8.1. *The following laws are valid*:

$$\textbf{skip};\ C \equiv C \equiv C;\ \textbf{skip}$$

$$(C_1;\ C_2);\ C_3 \equiv C_1;\ (C_2;\ C_3)$$

$$C \parallel \textbf{skip} \equiv C$$

$$C_1 \parallel C_2 \equiv C_2 \parallel C_1$$

$$(C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$

$$(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2);\ C \equiv \textbf{if } B \textbf{ then } C_1;\ C \textbf{ else } C_2;\ C$$

$$\textbf{if } (B_1 \,\&\, B_2) \textbf{ then } C_1 \textbf{ else } C_2 \sqsubseteq \textbf{if } B_1 \textbf{ then } (\textbf{if } B_2 \textbf{ then } C_1 \textbf{ else } C_2) \textbf{ else } C_2$$

$$\textbf{while } B \textbf{ do } C \equiv \textbf{if } B \textbf{ then } C;\ \textbf{while } B \textbf{ do } C \textbf{ else skip}.$$

Proof. The laws may be easily validated, taking advantage of natural algebraic identities involving $T_1;\ T_2$, $T_1 \parallel T_2$, and T^*. For example, **skip** is a unit for sequential and parallel composition because trace sets are closed under stuttering and mumbling. It is obvious from the definitions that for all closed sets T_1, T_2, and T_3, we have

$$\begin{aligned}(T_1;\ T_2);\ T_3 &= T_1;\ (T_2;\ T_3)\\ &= \{\alpha\beta\gamma \mid \alpha \in T_1 \,\&\, \beta \in T_2 \,\&\, \gamma \in T_3\}^\dagger\\ (T_1 \parallel T_2) \parallel T_3 &= T_1 \parallel (T_2 \parallel T_3)\\ &= \bigcup \{\alpha \parallel \beta \parallel \gamma \mid \alpha \in T_1 \,\&\, \beta \in T_2 \,\&\, \gamma \in T_3\}^\dagger\\ T_1 \parallel T_2 &= T_2 \parallel T_1,\end{aligned}$$

since concatenation and interleaving operations on transition traces are clearly associative, and interleaving of transition traces is obviously a symmetric operation. The rules for conditionals and loops are straightforward, since $(T_1 \cup T_2);\ T = (T_1;\ T) \cup (T_2;\ T)$ and $\mathscr{T}[\![B_1 \,\&\, B_2]\!] \subseteq \mathscr{T}[\![B_1]\!];\ \mathscr{T}[\![B_2]\!]$. ∎

In addition, since for all $\alpha, \beta, \gamma \in \Sigma^*$ we have $\alpha(\beta \parallel \gamma) \subseteq (\alpha\beta) \parallel \gamma$, we obtain the following law:

$$C_1;\ (C_2 \parallel C_3) \sqsubseteq (C_1;\ C_2) \parallel C_3,$$

from this law, we may derive the inequation $C_1;\ C_2 \sqsubseteq C_1 \parallel C_2$.

Since the semantics is tailored to correspond to partial correctness, and we model blocking as busy waiting, the semantics also validates the following laws:

$$\textbf{while true do skip} \sqsubseteq C$$

$$\textbf{while true do skip} \equiv \textbf{await false then } C.$$

This is reasonable, since a diverging program satisfies only vacuous partial correctness properties, and induces nontermination in every program context.

Since assignment is atomic, this semantics validates the law $I := I \equiv \textbf{skip}$. Since boolean expression evaluation is atomic, we can also show that

$$\mathscr{T}[\![B]\!];\ \mathscr{T}[\![B]\!] = \mathscr{T}[\![B]\!]$$

$$\mathscr{T}[\![B]\!];\ \mathscr{T}[\![\neg B]\!] \subseteq \mathscr{T}[\![B]\!] \cap \mathscr{T}[\![\neg B]\!],$$

and the semantics validates the following laws:

while B **do** C

$\equiv$ **if** B **then while** B **do** C **else skip**

if B **then** C_1 **else** C_2

$\equiv$ **if** B **then** (**if** B **then** C_1 **else** C_2) **else** C_2.

If the expression language is deterministic, so that for all E and s the set $\mathscr{E}[\![E]\!]s$ 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.

By way of contrast, we discuss briefly some results of de Bakker [6] concerning the equivalence in all *sequential* contexts of sequences of simple assignments, and we compare what happens in the parallel case. A simple assignment has form $I := I'$, where I and I' are (not necessarily distinct) identifiers. The following list of axioms is given by de Bakker:

$$x := y; y := x \equiv x := y$$

$$x := y; x := z \equiv x := z \quad (x \neq z)$$

$$x := y; z := x \equiv x := y; z := y$$

$$x := y; z := y \equiv z := y; x := y$$

Together with some simple rules, such as

$$\frac{C_1; x := x' \equiv C_2; x := x' \quad C_1; y := y' \equiv C_2; y := y' \quad x \neq y}{C_1 \equiv C_2}$$

this gives a complete axiomatization of equivalence in all sequential contexts. That is, two sequences of simple assignments induce the same behavior in all sequential contexts if and only if their equivalence can be proven from these axioms and rules.

However, each of these axioms fails for parallel contexts:

- The first of de Bakker's axioms is invalid in the context $[- \parallel x := 1]$.
- The second is invalid in the context $[- \parallel w := x]$.
- The third is invalid in the context $[- \parallel x := 0]$.
- The fourth is invalid in the context $[- \parallel y := 0]$.

Instead, the following inequational versions of the first two axioms hold in all parallel contexts:

$$x := y \sqsubseteq x := y; y := x$$

$$x := z \sqsubseteq x := y; x := z.$$

The second of these is itself a special case of the law given above for $I := E_1; I := E_2$. The third and fourth de Bakker axioms cannot be weakened to an inequality in either direction. For instance,

$$x := y; z := x \sqsubseteq x := y; z := y$$

is invalid in the context $[-] \parallel y := y + 1$ and the reverse inequality fails in the context $[-] \parallel x := x + 1$.

9. 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:

$$B ::= \mathbf{true} \mid \mathbf{false} \mid \neg B \mid B_1 \,\&\, B_2 \mid E_1 \leqslant E_2$$

$$E ::= 0 \mid 1 \mid I \mid E_1 + E_2 \mid \mathbf{if}\ B\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2.$$

To adapt the operational semantics we introduce the set **BExp′** of extended boolean expressions, defined by adding the clauses $B ::= v\ (v \in V)$ to the grammar for **BExp**, and the set **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 evaluated is E_2; a configuration of form $n \in N$ represents the final result of evaluation.

A fine-grained transition system for expressions is described in Figs. 2 and 3. 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 and subsequent semantic definitions may easily be modified to model different evaluation strategies.

Now that expression evaluation is no longer atomic, the semantic functions $\mathscr{E}$ and $\mathscr{B}$ are not fully abstract. Instead

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

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

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

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

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

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

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

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

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

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

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

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

FIG. 2. Fine-grained transition rules for boolean expressions.

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:[6]

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

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

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

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

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

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

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

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

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

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

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

FIG. 3. Fine-grained transition rules for integer expresions.

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

Thus a trace $((s_0, s_0)(s_1, s_1)\ldots(s_k, s_k), v) \in \mathscr{T}[\![B]\!]$ 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 ith interruption changing the state to s_i. In particular allowing no interruptions corresponds to the definition of $\mathscr{B}$, and $\mathscr{B}[\![B]\!] = \{(s, n) \mid ((s, s), n) \in \mathscr{T}[\![B]\!]\}$. 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 9.1. *For all boolean expressions B, all states s, all $\alpha, \beta \in \Sigma^*$, and all truth values v,*

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

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

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

PROPOSITION 9.2. *The fine-grained trace semantics* $\mathscr{T}$: **BExp** $\to \mathscr{P}^{\dagger}(\Sigma^{+} \times V)$ *is uniquely characterized by the following clauses*:

$$\mathscr{T}[\![\mathbf{true}]\!] = \{((s,s), \mathtt{tt}) \mid s \in S\}^{\dagger}$$

$$\mathscr{T}[\![\mathbf{true}]\!] = \{((s,s), \mathtt{ff}) \mid s \in S\}^{\dagger}$$

$$\mathscr{T}[\![\neg B]\!] = \{(\alpha, \neg v) \mid (\alpha, v) \in \mathscr{T}[\![B]\!]\},$$
$$\textit{where } \neg\mathtt{tt} = \mathtt{ff},\ \neg\mathtt{ff} = \mathtt{tt}$$

$$\mathscr{T}[\![B_1 \,\&\, B_2]\!] = \{(\alpha, \mathtt{ff}) \mid (\alpha, \mathtt{ff}) \in \mathscr{T}[\![B_1]\!]\}$$
$$\cup \{(\alpha\beta, v) \mid (\alpha, \mathtt{tt}) \in \mathscr{T}[\![B_1]\!] \,\&\, (\beta, v) \in \mathscr{T}[\![B_2]\!]\}^{\dagger}$$

$$\mathscr{T}[\![E_1 \leqslant E_2]\!] = \{(\gamma, m \leqslant n) \mid (\alpha, m) \in \mathscr{T}[\![E_1]\!] \,\&$$
$$(\beta, n) \in \mathscr{T}[\![E_2]\!] \,\&\, \gamma \in \alpha \parallel \beta\}^{\dagger}.$$

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 Fig. 4:

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

PROPOSITION 9.3. *The fine-grained trace semantics of commands is uniquely characterized by the following clauses*:

$$\mathscr{T}[\![\mathbf{skip}]\!] = \{(s,s) \mid s \in S\}^{\dagger}$$

$$\mathscr{T}[\![I := E]\!] = \{\alpha(s, [s \mid I = n]) \mid (\alpha, n) \in \mathscr{T}[\![E]\!]\}^{\dagger}$$

$$\mathscr{T}[\![C_1; C_2]\!] = \mathscr{T}[\![C_1]\!]; \mathscr{T}[\![C_2]\!]$$

$$\mathscr{T}[\![\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2]\!]$$
$$= \mathscr{T}[\![B]\!]; \mathscr{T}[\![C_1]\!] \cup \mathscr{T}[\![\neg B]\!]; \mathscr{T}[\![C_2]\!]$$

$$\mathscr{T}[\![\mathbf{while}\ B\ \mathbf{do}\ C]\!] = (\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!])^{*}; \mathscr{T}[\![\neg B]\!]$$

$$\mathscr{T}[\![C_1 \parallel C_2]\!] = \mathscr{T}[\![C_1]\!] \parallel \mathscr{T}[\![C_2]\!]$$

$$\mathscr{T}[\![\mathbf{await}\ B\ \mathbf{then}\ C]\!] = \{(s,s') \in \mathscr{T}[\![C]\!] \mid (s,s) \in \mathscr{T}[\![B]\!]\}^{\dagger}.$$

Here we write $\mathscr{T}[\![B]\!]$ *for the set* $\{\alpha \mid (\alpha, \mathtt{tt}) \in \mathscr{T}[\![B]\!]$ *and* $\mathscr{T}[\![\neg B]\!]$ *for* $\{\alpha \mid (\alpha, \mathtt{ff}) \in \mathscr{T}[\![B]\!]\}$.

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

Of course, since the operational semantics of commands is now fine-grained, we are now interested in notions of behavior defined as before but based instead on the fine-grained transition relation of Fig. 4.

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

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

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

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

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

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

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

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

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

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

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

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

FIG. 4. Fine-grained transition rules for commands.

PROPOSITION 9.4. *The fine-grained semantics is fully abstract: for all terms* t *and* t' *of the same syntactic type,* $t \sqsubseteq_{\mathscr{T}} t' \Leftrightarrow t \leqslant_{\mathscr{M}} t'$.

Proof. For commands the proof is similar to the proof of Proposition 7.1.

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

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

For example, the boolean expressions $x \leqslant x$ and **true** are not semantically equivalent: $(([x=0],[x=0])([x=1],[x=1]), \mathtt{ff}) \in \mathscr{T}[\![x \leqslant x]\!] - \mathscr{T}[\![\mathbf{true}]\!]$. As a result they may induce different behavior in contexts such as

$$x := 1 \parallel \mathbf{if}\ [-]\ \mathbf{then}\ z := 0\ \mathbf{else}\ z := 1.$$

The relationships given in Proposition 8.1 continue to hold for the fine-grained semantics. However, the identity $I := I \equiv \mathbf{skip}$ fails because assignment is not atomic. For example,

$$x := 0;\ (x := x \parallel x := 1) \not\equiv_{\mathscr{M}} x := 0;\ (\mathbf{skip} \parallel x := 1),$$

because $([x=0],[x=0])([x=1],[x=0])$ is a trace of $x := x$ but not of **skip**. Instead we get the inequality $\mathbf{skip} \sqsubset I := I$. Similarly, we have only an inclusion $\mathscr{T}[\![B]\!]; \mathscr{T}[\![B]\!] \supseteq \mathscr{T}[\![B]\!]$, the converse direction failing in general; and we obtain the following inequalities

$$\mathbf{while}\ B\ \mathbf{do}\ C \sqsubseteq \mathbf{if}\ B\ \mathbf{then}\ \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{else}\ \mathbf{skip}$$

$$\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \sqsubseteq \mathbf{if}\ B\ \mathbf{then}\ (\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2)\ \mathbf{else}\ C_2.$$

10. FAIRNESS

So far we have ignored the possibility of infinite computation and non-termination. This was appropriate for reasoning about partial correctness, and in general for any property determined entirely by the finite traces of a program. 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.

When reasoning about parallel programs it is often natural to make a *fairness* assumption [19]: when running commands in parallel, no individual command is forever denied its turn for execution. This kind of assumption permits us to abstract away from scheduling details, such as the relative speeds of parallel processes, and to deduce program properties that hold in any "realistic" implementation of the programming language. At least for the form of fairness described informally above, it is easy to design schedulers that guarantee fair execution, so that the fairness assumption provides a convenient and reasonable abstraction. In this section we extend our semantics to incorporate finite and infinite traces corresponding to fair executions. We provide an operational characterization of the set of fair traces of a program, followed by a denotational characterization. Since we model blocking as busy waiting, there is no need to distinguish between "weak" and "strong" fairness: a non-terminated command is *always* enabled, since even a command stuck at an **await** with its test false can perform an idle step.

In order to characterize the fair infinite computations of a command operationally, care must be taken to keep track of which parallel component performs each atomic action (see, for example, [8]). The crucial case is for a command of form $C_1 \parallel C_2$. An infinite transition sequence of $C_1 \parallel C_2$ is fair if and only if for each $i = 1, 2$ the sub-sequence of steps involving C_i is either finite and ends in a successfully terminated configuration, or is infinite.

For example, consider the program

$$C = (x := 0 \parallel \mathbf{while}\ x = 1\ \mathbf{do}\ y := y + 1)$$

started in initial state $[x=1, y=1]$. This program has infinitely many terminating computations, in which the loop body is executed a finite number of times and then the assignment to x occurs, causing the loop to terminate. For each positive integer k there is such a computation ending in the state $[x=0, y=k]$. There is also an infinite computation of C in which the loop body is repeated forever. However, this computation is *unfair*, since it can only occur if the assignment to x is forever denied a chance to execute. Under the fairness assumption this program is guaranteed to terminate. Since the final value of y is an arbitrary positive integer this program exhibits unbounded nondeterminism.

For another example, the program

$$x := 0;\ y := 0;\ (\mathbf{await}\ x = 1\ \mathbf{then}\ y := 1 \parallel \mathbf{while}\ y = 0\ \mathbf{do}\ x := 1 - x)$$

has a non-terminating weakly fair computation, in which the await command only gets to move after an even number of loop iterations, when x is equal to 0, so that it never changes the value of y. This shows that even under fairness assumptions it can be possible for blocking to persist.

We write

$$\langle C_0, s_0\rangle \to \langle C_1, s_1\rangle \to \cdots \langle C_n, s_n\rangle \to \cdots \text{fair}$$

to indicate a fair infinite computation. Similarly, we write

$$\langle C_0, s_0\rangle \to^* \langle C_1, s_1\rangle \to^* \cdots \langle C_n, s_n\rangle \to^* \cdots \text{fair}$$

to indicate an infinite sub-sequence of a fair infinite computation; in particular, this indicates that infinitely many of the transition sequences $\langle C_i, s_i\rangle \to^* \langle C_{i+1}, s_{i+1}\rangle$ involve a non-zero number of steps. Finally, we generalize further to non-consecutive transition sequences, writing

$$\langle C_0, s_0\rangle \to^* \langle C_1, s_0'\rangle\ \&$$
$$\langle C_1, s_1\rangle \to^* \langle C_2, s_1'\rangle\ \& \cdots \&$$
$$\langle C_n, s_n\rangle \to^* \langle C_{n+1}, s_n'\rangle \cdots \text{fair}$$

when there is a fair computation of C_0 from initial state s_0 during which execution is interrupted infinitely often, the ith interruption changing the state from s_i' to s_{i+1} (for each $i \geqslant 0$). 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.

This provides a natural generalization of transition traces to encompass fairness. We therefore define $\Sigma^\infty = \Sigma^+ \cup \Sigma^\omega$, the set of finite or infinite transition traces, and we characterize the fair trace semantics operationally as follows:

$$\begin{aligned}\mathscr{T}[\![C]\!] = \{&(s_0, s_0')(s_1, s_1')\ldots(s_k, s_k') \mid \\ &\langle C, s_0\rangle \to^* \langle C_1, s_0'\rangle \,\&\cdots\& \\ &\langle C_k, s_k\rangle \to^* \langle C', s_k'\rangle \ term\} \cup \\ \{&(s_0, s_0')\ldots(s_n, s_n')\ldots \mid \\ &\langle C, s_0\rangle \to^* \langle C_1, s_0'\rangle \,\&\cdots\& \\ &\langle C_n, s_n\rangle \to^* \langle C_{n+1}, s_n'\rangle \,\&\cdots \text{fair}\}.\end{aligned}$$

Note that C has an infinite interference-free trace beginning in state s iff $\langle C, s\rangle$ has a fair infinite computation from s.

For obvious reasons only finitely many interruptions can occur between successive atomic actions by C; consequently, $\mathscr{T}[\![C]\!]$ is again closed under stuttering and mumbling, where we allow finitely many stutters or mumbles between successive stages in a trace, and we allow use of a closure operation at infinitely many positions in an infinite trace. Formally, we extend the definition of closure to subsets of Σ^∞ as follows. A set T of finite and infinite traces is closed if and only if T satisfies the conditions of Proposition 6.1 and also

$$\begin{aligned}&\alpha_0\alpha_1\ldots\alpha_n\ldots \in T \\ &\quad\Rightarrow \alpha_0(s_0, s_0)\,\alpha_1(s_1, s_1)\ldots\alpha_n(s_n, s_n)\ldots \in T \\ &\alpha_0(s_0, s_0')(s_0', s_0'')\,\alpha_1(s_1, s_1')(s_1', s_1'')\,\alpha_2\ldots \in T \\ &\quad\Rightarrow \alpha_0(s_0, s_0'')\,\alpha_1(s_1, s_1'')\,\alpha_2\ldots \in T.\end{aligned}$$

We let $\mathscr{P}^\dagger(\Sigma^\infty)$ denote the set of closed sets of finite or infinite traces. This again forms a complete lattice under set inclusion.

Again we show that $\mathscr{T}$ can be defined compositionally. The clauses defining the traces of **skip** and assignment are unchanged.

We extend concatenation to infinite traces in the obvious way: if α is finite we define $\alpha\beta$ as before to be the result of concatenating β on the end of α; if α is infinite we define $\alpha\beta$ to be α. Then we define $T_1; T_2 = \{\alpha\beta \mid \alpha \in T_1, \beta \in T_2\}^\dagger$ and again we have $\mathscr{T}[\![C_1; C_2]\!] = \mathscr{T}[\![C_1]\!]; \mathscr{T}[\![C_2]\!]$.

We define T^* on closed sets of finite or infinite traces as before but using the extended notion of concatenation. We also define

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

We then extend the trace set of a loop command to include infinite traces corresponding to divergence:

$$\begin{aligned}&\mathscr{T}[\![\textbf{while } B \textbf{ do } C]\!] \\ &\quad = (\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!])^*; \mathscr{T}[\![\neg B]\!] \cup (\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!])^\omega.\end{aligned}$$

Similarly we extend the trace set of an await command to include infinite traces corresponding to blocking:

$$\begin{aligned}&\mathscr{T}[\![\textbf{await } B \textbf{ then } C]\!] \\ &= \{(s, s') \mid (s, s) \in \mathscr{T}[\![B]\!] \,\&\, (s, s') \in \mathscr{T}[\![C]\!]\}^\dagger \cup (\mathscr{T}[\![\neg B]\!])^\omega.\end{aligned}$$

For α and β in Σ^∞ let $\alpha \parallel \beta$ be the set of all traces built by fairly interleaving α with β. Perhaps the simplest way to define $\alpha \parallel \beta$ formally, following Park [19], is

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

where we extend concatenation to work on sets and on triples of traces in the obvious way. To be precise, for $X, Y \subseteq \Sigma^\infty$ let $XY = \{\alpha\beta \mid \alpha \in X, \beta \in Y\}$; for triples let $(\alpha_1, \alpha_2, \alpha_3)(\beta_1, \beta_2\beta_3) = (\alpha_1\beta_1, \alpha_2\beta_2, \alpha_3\beta_3)$. Intuitively, when $(\alpha, \beta, \gamma) \in (L^*RR^*L)^\omega$ each of α and β is infinite and γ is an interleaving of all of α with all of β. When $(\alpha, \beta, \gamma) \in (L \cup R)^*\, A$ at least one of α and β is finite, and γ is again an interleaving of all of α with all of β; as soon as all of one trace has been consumed there is no longer any fairness requirement.

Then we define a fair interleaving operator on closed sets of traces by

$$T_1 \parallel T_2 = \bigcup \{\alpha_1 \parallel \alpha_2 \mid \alpha_1 \in T_1 \,\&\, \alpha_2 \in T_2\}^\dagger,$$

with the result that $\mathscr{T}[\![C_1 \parallel C_2]\!] = \mathscr{T}[\![C_1]\!] \parallel \mathscr{T}[\![C_2]\!]$. With these definitions in hand, we can define $\mathscr{T}$ denotationally. We give details for the coarse-grained case; the corresponding fine-grained version is obtainable similarly.

DEFINITION 10.1. The fair transition traces semantic function $\mathscr{T}$: **Com** $\to \mathscr{P}^{\dagger}(\Sigma^{\infty})$ is defined by the following clauses:

$$\mathscr{T}[\![\textbf{skip}]\!] = \{(s, s) \mid s \in S\}^{\dagger}$$

$$\mathscr{T}[\![I := E]\!] = \{(s, [s \mid I = n]) \mid (s, n) \in \mathscr{E}[\![E]\!]\}^{\dagger}$$

$$\mathscr{T}[\![C_1; C_2]\!] = \mathscr{T}[\![C_1]\!]; \mathscr{T}[\![C_2]\!]$$

$$\mathscr{T}[\![\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2]\!]$$
$$= \mathscr{T}[\![B]\!]; \mathscr{T}[\![C_1]\!] \cup \mathscr{T}[\![\neg B]\!]; \mathscr{T}[\![C_2]\!]$$

$$\mathscr{T}[\![\textbf{while } B \textbf{ do } C]\!]$$
$$= (\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!])^{*}; \mathscr{T}[\![\neg B]\!] \cup (\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!])^{\omega}$$

$$\mathscr{T}[\![C_1 \parallel C_2]\!] = \mathscr{T}[\![C_1]\!] \parallel \mathscr{T}[\![C_2]\!]$$

$$\mathscr{T}[\![\textbf{await } B \textbf{ then } C]\!]$$
$$= \{(s, s') \in \mathscr{T}[\![C]\!] \mid (s, s) \in \mathscr{T}[\![B]\!]\}^{\dagger} \cup (\mathscr{T}[\![\neg B]\!])^{\omega}.$$

Yet again all operations on trace sets used in this semantics are continuous with respect to set inclusion. The semantics of while-loops again has an equivalent formulation using fixed-points, but now we must use a *greatest* fixed-point. As before, the function $F'(T) = (\mathscr{T}[\![B]\!] \cdot \mathscr{T}[\![C]\!] \cdot T \cup \mathscr{T}[\![\neg B]\!])$ is monotone on the complete lattice of (not necessarily closed) sets of traces. Its greatest fixed point is

$$\nu T.F'(T) = (\mathscr{T}[\![B]\!] \cdot \mathscr{T}[\![C]\!])^{*} \cdot \mathscr{T}[\![\neg B]\!] \cup (\mathscr{T}[\![B]\!] \cdot \mathscr{T}[\![C]\!])^{\omega},$$

and the closure of this set is $\mathscr{T}[\![\textbf{while } B \textbf{ do } C]\!]$.[7]

To show how the fairmerge definition works, we now return to the example programs discussed above. First, it is easy to show that all interference-free traces of the program

$$x := 0 \parallel \textbf{while } x = 1 \textbf{ do } y := y + 1$$

are finite, corresponding to the fact that according to the fair operational semantics this program always terminates. Second, consider the program

$$\textbf{await } x = 1 \textbf{ then } y := 1 \parallel \textbf{while } y = 0 \textbf{ do } x := 1 - x.$$

[7] The reader can easily check that $\mathscr{T}[\![\textbf{while } B \textbf{ do } C]\!]$ does *not* coincide with the greatest fixed point of the corresponding operator on closed sets, $\lambda T.(\mathscr{T}[\![B]\!]; \mathscr{T}[\![C]\!]; T \cup \mathscr{T}[\![\neg B]\!])$. For instance, when $B = \textbf{true}$ and $C = \textbf{skip}$ this functional is simply $\lambda T.T$ and its greatest fixed point is $\mathscr{P}^{\dagger}(\Sigma^{\infty})$, the set of all traces; but $\mathscr{T}[\![\textbf{while true do skip}]\!] = \{(s, s) \mid s \in S\}^{\omega}$, consisting only of the infinite stuttering sequences.

Introduce the following abbreviations:

$$\sigma_{-} = ([x = 1, y = 0], [x = 0, y = 0])$$
$$\sigma_{+} = ([x = 0, y = 0], [x = 1, y = 0])$$
$$\tau = ([x = 0, y = 0], [x = 0, y = 0])$$

Then $(\sigma_{-}\sigma_{+})^{\omega}$ is a trace of the while-loop, and τ^{ω} is a trace of the await-command. And we have $((\sigma_{-}\sigma_{+})^{\omega}, \tau^{\omega}, (\sigma_{-}\tau\sigma_{+})^{\omega}) \in \mathit{fairmerge}$. The trace $(\sigma_{-}\tau\sigma_{+})^{\omega}$ represents a fair infinite execution of the parallel program in which the await command is continually blocked. The program also has traces corresponding to executions in which the await command is scheduled in a state satisfying its test, so that y gets set to 1.

10.1. *Full Abstraction*

We now generalize the notion of state trace behavior to include infinite state traces. Let $S^{\infty} = S^{+} \cup S^{\omega}$ be the set of non-empty finite and infinite sequences of states.

DEFINITION 10.2. The fair state trace behavior function $\mathscr{S}$: **Com** $\to \mathscr{P}(S^{\infty})$ is given by:

$$\mathscr{S}[\![C]\!] = \{s_0 s_1 \ldots s_k \mid \langle C, s_0\rangle \to^{*} \langle C_1, s_1\rangle \to^{*} \cdots \to^{*} \langle C_k, s_k\rangle \mathit{term}\} \cup \{s_0 \ldots s_n \ldots \mid \langle C, s_0\rangle \to^{*} \langle C_1, s_1\rangle \to^{*} \cdots \langle C_n, s_n\rangle \to^{*} \cdots \text{fair}\}.$$

By definition, as before, $\mathscr{S}[\![C]\!]$ is determined by the interference-free subset of $\mathscr{T}[\![C]\!]$.

PROPOSITION 10.3. *The fair transition trace semantics is fully abstract with respect to fair state traces: for all commands C and C', $C \sqsubseteq_{\mathscr{T}} C' \Leftrightarrow C \leqslant_{\mathscr{S}} C'$.*

Proof. As before the forward implication is straightforward. For the converse, suppose C has a trace α that is impossible for C'. If α is finite we may argue as before. Otherwise, suppose α is infinite, say

$$\alpha = (s_0, s_0')(s_1, s_1') \ldots (s_n, s_n') \ldots.$$

Let $x_1, \ldots, x_k$ be the (finitely many) identifiers occurring free in either C or C'. Without loss of generality we can assume that the domain of each state in α is $\{x_1, \ldots, x_k\}$. Let t, c, $y_1, \ldots, y_k$ and $z_1, \ldots, z_k$ be distinct fresh identifiers. Let $\bar{x} := \bar{z}$ stand for the command

$$x_1 := z_1; \ldots; x_k := z_k$$

and $\bar{y} := \bar{0}$ stand for $y_1 := 0; \ldots; y_k := 0$. Let $\bar{x} = \bar{y}$ be the boolean expression

$$(x_1 = y_1) \,\&\, \cdots \,\&\, (x_k = y_k).$$

Let CHOOSE($\bar{y}$) be the following command:

$$\begin{aligned}\bar{y} := \bar{0};\ t := 0;\ (\quad & t := 1 \\ \| \ & \textbf{while } t = 0 \textbf{ do } y_1 := y_1 + 1 \\ \| \ & \textbf{while } t = 0 \textbf{ do } y_2 := y_2 + 1 \\ \| \ & \ldots\ldots\ldots \\ \| \ & \textbf{while } t = 0 \textbf{ do } y_k := y_k + 1 \\ )\end{aligned}$$

with a similar form for CHOOSE($\bar{z}$). Intuitively, CHOOSE($\bar{y}$) "guesses" a state; fairness guarantees termination and that for each state s there is a computation of CHOOSE($\bar{y}$) that guesses s, by setting each y_i to $s(x_i)$ for $i = 1, \ldots, k$. We will write $\bar{y} = s(\bar{x})$ to indicate the state $[y_1 = s(x_1), \ldots, y_k = s(x_k)]$.

Let $P[-]$ be the context

$$\begin{aligned}[-] \| \textbf{while true do } (\ & \text{CHOOSE}(\bar{y}); \\ & \text{CHOOSE}(\bar{z}); \\ & c := c + 1; \\ & \textbf{await } \bar{x} = \bar{y} \textbf{ then } \bar{x} := \bar{z}\ ).\end{aligned}$$

Then $P[C]$ has an infinite state trace characteristic of α while $P[C']$ does not. Specifically, $P[C]$ has a fair execution in which at stage n the loop guesses the states s'_n and s_{n+1} then waits until C reaches s'_n before setting the state to s_{n+1}. This corresponds to a state trace of the form

$$\begin{aligned}&[\bar{x} = s_0(\bar{x}) \mid y = \bar{0} \mid \bar{z} = \bar{0} \mid c = 0] \\ &\quad [\bar{x} = s_0(\bar{x}) \mid \bar{y} = s'_0(\bar{x}) \mid \bar{z} = s_1(\bar{x}) \mid c = 1] \\ &\quad\quad [\bar{x} = s'_0(\bar{x}) \mid \bar{y} = s'_0(\bar{x}) \mid \bar{z} = s_1(\bar{x}) \mid c = 1] \\ &\quad\quad\quad [\bar{x} = s_1(\bar{x}) \mid \bar{y} = s'_1(\bar{x}) \mid \bar{z} = s_2(\bar{x}) \mid c = 2] \\ &\quad\quad\quad\quad [\bar{x} = s'_1(\bar{x}) \mid \bar{y} = s'_1(\bar{x}) \mid \bar{z} = s_2(\bar{x}) \mid c = 2] \\ &\quad\quad\quad\quad\quad [\bar{x} = s_2(\bar{x}) \mid \bar{y} = s'_2(\bar{x}) \mid \bar{z} = s_3(\bar{x}) \mid c = 3] \\ &\quad\quad\quad\quad\quad\quad \ldots\ldots\ldots\ldots,\end{aligned}$$

with alternating steps made by the context and by C. Since C' cannot manipulate the y_i, z_i or c, and does not have the trace α, $P[C']$ has no state trace of this form. ∎

The laws given in Proposition 8.1 continue to hold for the fair trace semantics. However, the inequation

$$C_1; (C_2 \| C_3) \sqsubseteq (C_1; C_2) \| C_3$$

and its corollary $C_1; C_2 \sqsubseteq C_1 \| C_2$ may fail when C_1 has infinite traces. Nevertheless, these inequations still hold if C_1 has only finite traces, for instance when C_1 contains no loops or awaits.

Note that the busy-wait treatment of blocking means that this semantics still identifies **await false then skip** with **while true do skip**, since they both denote the set of all infinite stuttering sequences. However, the inequation **while true do skip** $\sqsubseteq C$ now fails.

11. ROBUSTNESS

The full abstraction results given above relied only on certain general properties: monotonicity of the semantic definitions, compositionality of $\mathscr{T}$, 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.

11.1 *Non-deterministic Choice*

For instance, we may add a non-deterministic choice construct C_1 **or** C_2, with operational semantics given by

$$\langle C_1 \textbf{ or } C_2, s\rangle \to \langle C_1, s\rangle$$
$$\langle C_1 \textbf{ or } C_2, s\rangle \to \langle C_2, s\rangle.$$

Then $\mathscr{T}[\![C_1 \textbf{ or } C_2]\!] = \mathscr{T}[\![C_1]\!] \cup \mathscr{T}[\![C_2]\!]$, and all of the previous development goes through with minor modifications.[8] For the language extended with non-deterministic choice this semantics is again fully abstract, and the laws of programming given earlier continue to hold. In addition, **or** is idempotent, commutative and associative, **or** distributes through sequential and parallel composition, and $C \sqsubseteq C'$ if and only if $(C \textbf{ or } C') = C'$. The following laws of equivalence hold:

$$C \textbf{ or } C \equiv C$$
$$C_1 \textbf{ or } C_2 \equiv C_2 \textbf{ or } C_1$$
$$(C_1 \textbf{ or } C_2) \textbf{ or } C_3 \equiv C_1 \textbf{ or } (C_2 \textbf{ or } C_3)$$
$$(C_1 \textbf{ or } C_2); C \equiv (C_1; C) \textbf{ or } (C_2; C)$$
$$C; (C_1 \textbf{ or } C_2) \equiv (C; C_1) \textbf{ or } (C; C_2)$$
$$(C_1 \textbf{ or } C_2) \| C \equiv (C_1 \| C) \textbf{ or } (C_2 \| C)$$
$$\textbf{if } B \textbf{ then } (C \textbf{ or } C') \textbf{ else } C_2 \equiv (\textbf{if } B \textbf{ then } C \textbf{ else } C_2) \textbf{ or } (\textbf{if } B \textbf{ then } C' \textbf{ else } C_2).$$

[8] Of course, the coroutine construct C_1 **co** C_2 from Hennessy–Plotkin cannot be handled by our semantics, since $\mathscr{T}[\![C_1 \textbf{ co } C_2]\!]$ cannot be determined from $\mathscr{T}[\![C]\!]_1$ and $\mathscr{T}[\![C]\!]_2$.

The coarse-grained semantics satisfies the law

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

but this fails in the fine-grained case since $x := x+1 \parallel x := x+1$ has the trace $([x=0], [x=1])$, but this is not a trace of $x := x+1; x := x+1$. In the fine-grained semantics we obtain instead the following inequation:

$$I_1 := E_1 \parallel I_2 := E_2 \sqsupseteq (I_1 := E_1; I_2 := E_2) \textbf{ or } (I_2 := E_2; I_1 := E_1).$$

11.2. *Local Variables*

We can also expand the programming language to include *local variable declarations* by adding the syntactic clause

$$C ::= \textbf{new } I = E \textbf{ in } C'.$$

The intention is that I is a local variable initialized to the value of E and that C' may update and read I. Each occurrence of I in C' is bound by the declaration, so that

$$\text{free}[\![\textbf{new } I = E \textbf{ in } C']\!] = \text{free}[\![E]\!] \cup (\text{free}[\![C']\!] - \{I\}).$$

Since the scope of the declaration *only* includes C' no other process executing in parallel may read or write to the local variable. An advantage provided by this extension is that it permits a natural form of reasoning about the behavior of programs under "locality" assumptions, such as the assumption that no parallel command will alter the value of certain shared-variables. It is possible to incorporate local variable declarations into both coarse- and fine-grained semantic models. Again we give details for the coarse-grained case.

To define the operational behavior of this construct we first introduce generalized commands of the form **new** $[I=n]$ **in** C, representing a block with body C using local variable I with current value n. The transition rules are

$$\frac{\langle E, s\rangle \to^* n}{\langle \textbf{new } I = E \textbf{ in } C, s\rangle \to \langle \textbf{new } [I=n] \textbf{ in } C, s\rangle}$$

$$\frac{\langle C, [s \mid I=n]\rangle \to \langle C', [s' \mid I=n']\rangle}{\left(\begin{array}{l}\langle \textbf{new } [I=n] \textbf{ in } C, s\rangle \\ \quad \to \langle \textbf{new } [I=n'] \textbf{ in } C', [s' \mid I=s(I)]\rangle\end{array}\right)}$$

$$\frac{\langle C, [s \mid I=n]\rangle \; term}{\langle \textbf{new } [I=n] \textbf{ in } C, s\rangle \; term}$$

where we extend the update operation on states so that $[s' \mid I=s(I)]$ denotes $s'\backslash I$ when $I \notin \text{dom}(s)$, where $s'\backslash I$ is the restriction of s' to the set $\text{dom}(s') - \{I\}$. Note the careful treatment of I, distinguishing between local and global versions of the identifier; it is easy to see that every computation of **new** $I = E$ **in** C from a state s leaves the value of (the global) I unchanged.

To give a denotational description we first introduce some auxiliary definitions. For a trace α with nth step (s_n, s'_n) let $\alpha\backslash I$ be the trace with nth step $(s_n, [s'_n \mid I = s_n(I)])$. By convention, if $I \notin \text{dom}(s_n)$ this collapses to (s_n, s'_n). Clearly $\alpha\backslash I$ is a trace in which the value of I is never changed "internally." We also define $\langle I=k\rangle\alpha$ to be the trace beginning with the step $([s_0 \mid I=k], s'_0)$ and whose nth step for $n \geqslant 1$ is $([s_n \mid I = s'_{n-1}(I)], s'_n)$. This operation constructs a trace in which I is initialized to the value k and is never changed "externally." Then we define

$$\mathscr{T}[\![\textbf{new } I = E \textbf{ in } C']\!] = \{\alpha(\beta\backslash I) \mid (\alpha, k) \in \mathscr{T}[\![E]\!] \;\&\; \langle I=k\rangle\beta \in \mathscr{T}[\![C']\!]\}^{\dagger}.$$

The definition captures the idea that C' treats I as a local variable, and even though its environment may change the value of (a global variable named) I such a change does not affect the value of the local version of I; similarly, although C' may change the value of its local I it has no effect on the global version of I.

For example, the command

$$\textbf{new } x = 1 \textbf{ in } (y := x; x := x+1; y := x)$$

has each of the traces

$$([y=0], [y=1])([y=1], [y=2])$$
$$([x=9, y=0], [x=9, y=1])$$
$$([x=9, y=1], [x=9, y=2])$$
$$([x=9, y=0], [x=9, y=1])$$
$$([x=5, y=1], [x=5, y=2])$$
$$([y=0], [y=1])([y=8], [y=2])$$

but no traces of the forms

$$\alpha([x=9, y=0], [x=9, y=9])\beta$$
$$\alpha([x=0, y=0], [x=1, y=0])\beta$$

for any $\alpha, \beta \in \Sigma^{\infty}$. It is also easy to see that $\mathscr{T}[\![\textbf{new } x = 0 \textbf{ in } x := x+1]\!] = \mathscr{T}[\![\textbf{skip}]\!]$.

With this expansion the trace semantics is still fully abstract, and validates the following natural and intuitively valid equivalence:

$$\textbf{new } I = E \textbf{ in } C \equiv C \qquad \text{if} \quad I \notin \text{free}[\![C]\!].$$

The proof relies on the following property: if I does not occur free in C then every trace α of C satisfies $\alpha \backslash I = \alpha$.

Similarly we can show that when I is not free in C_1, and C_1 does not change any identifier occurring in E, we get

$$\textbf{new } I = E \textbf{ in } (C_1; C_2) \equiv C_1; \textbf{new } I = E \textbf{ in } C_2$$

$$\textbf{new } I = E \textbf{ in } (C_1 \parallel C_2) \equiv C_1 \parallel \textbf{new } I = E \textbf{ in } C_2.$$

Finally, if we define $[I'/I]\,C$ to be the command obtained by replacing all free occurrences of I in C by I', renaming bound variables when necessary to avoid capture, we can show the validity of the law

$$\textbf{new } I = E \textbf{ in } C \equiv \textbf{new } I' = E \textbf{ in } [I'/I]\,C,$$

provided I' does not occur free in C.

As an example illustrating the use of local variables, assume that f denotes a boolean-valued total function on the integers and consider the following commands:

$$C_0 = \textbf{while } \mathit{found} = 0 \textbf{ do } (\textbf{if } f(i) \textbf{ then } (\mathit{found} := 1; k := i) \textbf{ else } i := i + 2)$$

$$C_1 = \textbf{while } \mathit{found} = 0 \textbf{ do } (\textbf{if } f(j) \textbf{ then } (\mathit{found} := 1; k := j) \textbf{ else } j := j + 2).$$

Let SEARCH be the program

$$\textbf{new } \mathit{found} = 0 \textbf{ in } (\textbf{new } i = 0 \textbf{ in } C_0) \parallel (\textbf{new } j = 1 \textbf{ in } C_1).$$

Intuitively, SEARCH runs two loops in parallel, each looking for a "root" of f, and terminates if a root is found. Each loop uses a local counter variable, and the variables k and *found* are shared. Assuming that f has a root and that execution is fair, this program should always terminate and the final value of k should be a root of f. This is shown by the trace set:[9]

$$\mathscr{T}[\![\text{SEARCH}]\!] = \{(s, [s \mid k = n]) \mid n \in N \,\&\, (s, \texttt{tt}) \in \mathscr{B}[\![f(n)]\!]\}^{\dagger}.$$

[9] We tolerate a slight abuse of notation, using n to denote both an integer and the corresponding numeral.

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 two forms of program behavior, corresponding to partial and strong 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.

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 (sequences of communications) to build models of determinate or indeterminate dataflow networks, notably [11, 13, 21]. Indeed, others have also used sequences of pairs of states in imperative settings [2, 7, 10, 19].

In the earlier papers [2, 19] a pair of states was used to represent a *single* atomic action. Park's semantics [19] is obviously closely related to ours, since we adapt Park's definition of fairmerge; but Park's use of single-step traces causes his semantics to be too concrete, distinguishing between **skip** and **skip**; **skip** again. Abrahamson's semantics [2] is too concrete for the same reason, and ignores fairness. The key difference between this early work and ours is that we use a pair of states to represent an arbitrary *finite sequence* of atomic actions, and consequently develop a semantics based on *closed* sets of traces.

In the more recent papers [7, 10] a pair of states represents a single state-changing atomic action together with a finite sequence of idle steps. The semantics presented in [7, 10] models a program as a set of finite traces closed under a slightly weaker form of stuttering and mumbling than ours. In [7, 10] a set of traces T is said to be closed if and only if

$$\alpha\beta \in T \Rightarrow \alpha(s, s)\beta \in T$$

$$\alpha(s, s)(s, s')\beta \in T \Rightarrow \alpha(s, s')\beta \in T$$

$$\alpha(s, s')(s', s')\beta \in T \Rightarrow \alpha(s, s')\beta \in T,$$

so that mumbling is only allowed to absorb idle steps. The semantics given in [7] does not explicitly account for recursion (and therefore ignore issues of fairness). Their semantics achieves full abstraction with respect to a different notion of behavior, in which the observer of a program is assumed to be omniscient, able to observe *every* state-change occurring during a computation. For example the equivalence

$$(x := 1; x := x + 1) \textbf{ or } x := 2 \equiv x := 1; x := x + 1$$

fails in the semantics of [7], whereas it holds in our model. The treatment of local variables in [7] is essentially the same as ours.

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.

Jonsson and Back [12] have independently constructed a trace model closely related to ours, concentrating on total correctness and dealing with unfair as well as fair execution; they consider a simple imperative language of guarded commands (with branches executed as atomic actions), extended with an interleaving parallel composition. Their model for partial correctness is essentially the same as our finite trace model, involving closure under stuttering and mumbling; for total correctness they introduce closure under "fair unstoppability." In an earlier version of this paper [5], we mentioned that in a total correctness semantics trace sets would need to be closed under "chattering"; as Jonsson and Back show, this is not sufficient to achieve full abstraction. However, since the notion of termination considered in [12] is rather specialized, it is unclear if their ideas can be used to yield a total correctness semantics for our shared-variable language.

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

$$\langle I := ?, s\rangle \to \langle \mathbf{skip}, [s \mid I = n]\rangle \quad (n \in N)$$

and the following denotational semantics:

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

This would again yield a fully abstract semantics.

In contrast unbounded non-determinism causes severe problems in traditional powerdomain semantics [9, 4]. Apt and Plotkin [4] 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 a notion of behavior based on strong correctness, equivalently with respect to finite and infinite state traces. Our fair trace model provides a denotational continuous semantics for a parallel version of this language, is fully abstract with respect to the same notion of behavior, but is not a least fixed point semantics. The fair trace model can instead be characterized as the closure of a greatest fixed point semantics. We also showed that the corresponding least fixed point semantics (the finite traces model) is fully abstract with respect to finite state trace behavior, equivalently with respect to partial correctness behavior. For the sequential language discussed by Apt and Plotkin there is no need to use traces to achieve full abstraction, since the relevant behavior functions can already be defined compositionally. When our semantic definitions are simplified and adapted to the sequential setting they yield two fully abstract semantics for the Apt–Plotkin language, with respect to partial correctness and strong correctness, respectively.

ACKNOWLEDGMENTS

Several people have made helpful comments and suggestions that led to improvements in the presentation of this paper, including Albert Meyer, Prakash Panangaden, Vaughan Pratt, John Reynolds, and the anonymous referees. Susan Older suggested using non-deterministic "guessing" to simplify the full abstraction proof for the fair trace model. 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 7597. Support also came from the National Science Foundation under Grant 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.

Received January 1, 1994; final manuscript received February 1, 1996

REFERENCES

1. Abadi, M., and Plotkin, G. D. (1993), A logical view of composition, *Theoret. Comput. Sci.* **114**(1), 3–30.
2. Abrahamson, K. (1979), Modal logic of concurrent nondeterministic programs, *in* "Semantics of Concurrent Computation" (G. Kahn, Ed.), Lecture Notes in Computer Science, Vol. 70, pp. 21–33, Springer-Verlag, Berlin/New York.
3. Andrews, G. R., and Schneider, F. B. (1983), Concepts and notations for concurrent programming, *ACM Comput. Surveys* **15**(1), 3–43.
4. Apt, K. R., and Plotkin, G. D. (1986), Countable nondeterminism and random assignment, *J. Assoc. Comput. Mach.* **33** (4), 724–767.
5. Brookes, S. (1993), Full abstraction for a shared variable parallel language, *in* "Proc. 8th Annual IEEE Symposium on Logic in Computer Science," IEEE Comput. Soc. Press, Los Alamitos, CA.
6. de Bakker, J. W. (1971), Axiom systems for simple assignment statements, *in* "Symposium on Semantics of Algorithmic Languages" (E. Engeler, Ed.), Lecture Notes in Mathematics, Vol. 188, pp. 1–22, Springer-Verlag, Berlin/New York.
7. de Boer, F., Kok, J., Palamidessi, C., and Rutten, J. (1991), The failure of failures in a paradigm for asynchronous communication, *in* "Concur '91" (J. Baeten and J. Groote, Eds.), Lecture Notes in Computer Science, Vol. 527, pp. 111–126, Springer-Verlag, Berlin/New York.
8. Francez, N. (1986), "Fairness," Springer-Verlag, Berlin/New York.
9. Hennessy, M., and Plotkin, G. D. (1979), Full abstraction for a simple parallel programming language, *in* "Mathematical Foundations of Computer Science," Lecture Notes in Computer Science, Vol. 74, pp. 108–120, Springer-Verlag, Berlin/New York.
10. Horita, E., de Bakker, J., and Rutten, J. (1990), Fully Abstract Denotational Models for Nonuniform Concurrent Languages," Technical Report CS-R9027, Centre for Mathematics and Computer Science, Amsterdam.
11. Jonsson, B. (1989), A fully abstract trace semantics for dataflow networks, *in* "Sixteenth Annual ACM Symposium on Principles of Programming Languages," pp. 155–165, ACM Press, New York.

12. Jonsson, B., and Back, R. J. R. (1993), Fully abstract semantic orderings for shared-variable concurrent programs, draft paper.
13. Keller, R. M., and Panangaden, P. (1986), Semantics of digital networks containing indeterminate operators, *Distrib. Comput.* **1**(4), 235–245.
14. Lamport, L. (1977), Proving the correctness of multiprocess programs, *IEEE Trans. Software Eng.* **3**(2), 125–143.
15. Lamport, L. (1983), What good is temporal logic? *in* "Information Processing 83: Proceedings of the IFIP 9th World Congress" (R. E. A. Mason, Ed.), pp. 657–668, North-Holland, Amsterdam.
16. Lehmann, D., Pnueli, A., and Stavi, J. (1981), Impartiality, justice, and fairness: the ethics of concurrent termination, *in* "Proceedings of the Eighth International Conference on Automata, Languages and Programming," Lecture Notes in Computer Science, Vol. 115, pp. 264–277, Springer-Verlag, Berlin/New York.
17. Milner, R. (1977), Fully abstract models of typed lambda-calculi, *Theoret. Comput. Sci.* **4**, 1–22.
18. Owicki, S. S., and Gries, D. (1976), An axiomatic proof technique for parallel programs, *Acta Inform.* **6**, 319–340.
19. Park, D. (1979), On the semantics of fair parallelism, *in* "Abstract Software Specifications" (D. Bjøner, Ed.), Lecture Notes in Computer Science, Vol. 86, pp. 504–526, Springer-Verlag, Berlin/New York.
20. Plotkin, G. D. (1977), LCF considered as a programming language, *Theoret. Comput. Sci.* **5**(3), 223–255.
21. Russell, J. R. (1989). Full abstraction for nondeterministic dataflow networks, *in* "Proceedings of the 30th Annual Symposium on Foundations of Computer Science," pp. 170–177, IEEE Press, New York.
22. Stoughton, A. (1988), "Fully Abstract Models of Programming Languages," Research Notes in Theoretical Computer Science, Pitman, London.

Printed by Catherine Press, Ltd., Tempelhof 41, B-8000 Brugge, Belgium