Transfer Principles for Reasoning about Concurrent Programs

Stephen Brookes
Carnegie Mellon University

Submitted to MFPS 17

Abstract

In previous work we developed a *transition trace* semantic model, suitable for shared-memory parallel programs as well as networks of asynchronous communicating processes, abstract enough to support compositional reasoning about safety and liveness properties. We now use this framework to formalize and generalize some techniques used more or less informally in the literature to facilitate reasoning about the behavior of concurrent systems, typically without explicit attention to semantic foundations. Specifically, we identify a key *sequential-to-parallel transfer theorem* which, when applicable, allows us to replace a piece of a parallel program with another piece which is *sequentially* equivalent, with the guarantee that the safety and liveness properties of the overall program are unaffected. Two code fragments are said to be sequentially equivalent if they satisfy the same partial and total correctness properties. We also specify both coarse-grained and fine-grained version of trace semantics, assuming different degrees of atomicity, and we provide a *coarse-to-fine-grained transfer theorem* which, when applicable, allows replacement of a code fragment by another fragment which is *coarsely* equivalent, with the guarantee that the safety and liveness properties of the overall program are unaffected even if we assume fine-grained atomicity. Both of these results permit the use of a simpler, more abstract semantics, together with a notion of semantic equivalence which is easier to establish, to facilitate reasoning about the behavior of a parallel system.

1 Introduction

It is well known that reasoning about the behavior of parallel programs tends to be complicated by the combinatorial explosion caused by keeping track of the ways in which concurrent code fragments may interact dynamically. It is also well known that simple proof techniques based on state-transformation semantics do not adapt easily to the parallel setting. A more sophisticated semantic model is required, in which an accurate account can be given of interaction or interference between programs. Trace semantics provides a mathematical framework in which such reasoning may be carried out, and to some extent the combinatorial problems may be eased by the use of a number of laws of program equivalence, validated by trace semantics, which allow us (when applicable) to deduce properties of one program by analyzing instead a semantically equivalent program with simpler structure. The use of a succinct and compact notation for trace sets (based on extended regular expressions) can also help streamline program analysis.

Nevertheless, especially when reasoning about a parallel system which uses local variables in a disciplined manner, we would like to be able to adopt even simpler semantic models, and ideally we would like to be able to take advantage of forms of reasoning familiar from simpler settings. It is not generally safe to do so; for instance, it is well known that many laws of program equivalence that hold in the sequential setting cease to be valid in parallel languages. Yet local variables can only be accessed by processes occurring within a syntactically prescribed scope, and cannot be changed by any other processes running concurrently, and we ought to be able to take advantage of this property to simplify reasoning. Furthermore, when local variables are only ever used sequentially, in contexts which guarantee that no more than one process ever gains concurrent access, we should be able to employ styles of reasoning familiar from the sequential setting.

A number of more or less *ad hoc* techniques or methodologies have been proposed along these lines in the literature, usually without detailed consideration of semantic foundations, to facilitate concurrent program analysis by allowing replacement of a code fragment by another piece of code with "simpler" behavioral properties that permit an easier correctness proof.

In this paper we use our trace-theoretic framework to formalize and generalize some of these techniques. By paying careful attention to the underlying semantic framework we are able to recast these techniques in a more precise manner and we can be more explicit about the (syntactic and semantic) as-

sumptions upon which their validity rests. Since these techniques allow us to deduce program equivalence properties based on one semantic model by means of reasoning carried out on top of a different semantic model, we refer to our results as *transfer principles*. Our work can also be seen as further progress towards a theory of *context-sensitive development of parallel programs*, building on earlier work of Cliff Jones and spurred on by the recent Ph. D. thesis of Jüergen Dingel. We focus our attention in this extended abstract on some methodological ideas presented in Greg Andrews's book on concurrent programming. In the full version we intend to explore more fully the potential of our framework as a basis for further generalization and to extend our results to cover some of the contextual refinement ideas introduced by Dingel.

2 Syntax

Our parallel programming language is described by the following abstract grammar for commands c, in which b ranges over boolean-valued expressions, e over integer-valued expressions, x over identifiers, a over atomic commands (finite sequences of assignments), and d over declarations. The syntax for expressions is conventional and is assumed to include the usual primitives for arithmetic and boolean operations.

$$
\begin{array}{rcl}
c & ::= & \textbf{skip} \mid x{:=}e \mid c_1; c_2 \mid \\
 & & \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & \textbf{while } b \textbf{ do } c \mid \\
 & & \textbf{await } b \textbf{ then } a \mid \\
 & & c_1 \| c_2 \mid \\
 & & \textbf{local } d \textbf{ in } c \\
d & ::= & x = e \mid d_1; d_2 \\
a & ::= & \textbf{skip} \mid x{:=}e \mid a_1; a_2
\end{array}
$$

A command of form **await** b **then** a is a *conditional atomic action*, and causes the execution of a without interruption when executed in a state satisfying the test expression b; when executed in a state in which b is false the command idles. We will use the abbreviation $\langle a \rangle$ for **await true then** a.

Assume given the standard definitions of $\texttt{free}(c)$, the set of identifiers occurring free in c, and $\texttt{dec}(d)$, the set of identifiers declared by d.

A *parallel context* is a command which may contain a syntactic "hole" suitable for insertion of another command. Formally, the set of parallel contexts, ranged over by C, is described by the following abstract grammar, in which c_1, c_2 again range over parallel commands:

$$\begin{array}{rcl} C & ::= & [-] \mid \mathbf{skip} \mid x{:=}e \mid C; c_2 \mid c_1; C \\ & & \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ C \mid \\ & & \mathbf{while}\ b\ \mathbf{do}\ C \mid \\ & & \mathbf{await}\ b\ \mathbf{then}\ a \mid \\ & & C \| c_2 \mid c_1 \| C \mid \\ & & \mathbf{local}\ d\ \mathbf{in}\ C \end{array}$$

Note that our abstract grammar for contexts only allows at most one hole to appear in any particular context. It would be straightforward to adopt a more general notion of multi-holed context, but the technical details would become more involved and in any case there is no significant loss of generality.

We write $C[c]$ for the command obtained by inserting c into the hole of C. Note that the hole in a context may occur inside the scope of one or more (nested) declarations, and free occurrences of identifiers in c may become bound after insertion. To be precise about this possibility we let $\mathit{bound}(C)$ be the set of identifiers for which there is a binding declaration enclosing the hole in C, defined as follows:

$\mathit{bound}([-]) = \{\}$
$\mathit{bound}(x{:=}e) = \{\}$
$\mathit{bound}(C; c_2) = \mathit{bound}(c_1; C) = \mathit{bound}(C)$
$\mathit{bound}(\mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ c_2) = \mathit{bound}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ C) = \mathit{bound}(C)$
$\mathit{bound}(\mathbf{while}\ b\ \mathbf{do}\ C) = \mathbf{bound}(C)$
$\mathit{bound}(\mathbf{await}\ b\ \mathbf{then}\ a) = \{\}$
$\mathit{bound}(C \| c_2) = \mathit{bound}(c_1 \| C) = \mathit{bound}(C)$
$\mathit{bound}(\mathbf{local}\ d\ \mathbf{in}\ C) = \mathit{bound}(C) \cup \mathit{dec}(d)$

It is also possible in the same manner to introduce the notion of an *atomic context*, i.e. a command containing a hole into which an atomic command may be inserted; we will use A to range over atomic contexts, and write $A[a]$ for the command obtained by inserting a into the hole in A. Atomic contexts

conform to the following abstract grammar:

$$\begin{array}{rcl} A & ::= & [-] \mid \textbf{skip} \mid x{:=}e \mid A; c_2 \mid c_1; A \\ & & \textbf{if } b \textbf{ then } A \textbf{ else } c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } A \mid \\ & & \textbf{while } b \textbf{ do } A \mid \\ & & \textbf{await } b \textbf{ then } [-] \mid \\ & & A \| c_2 \mid c_1 \| A \mid \\ & & \textbf{local } d \textbf{ in } A \end{array}$$

A *sequential program* is just a command containing no **await** and no parallel composition. A *sequential context* is a limited form of context in which the hole never appears in parallel. We can characterize the set of sequential contexts, ranged over by S, as follows:

$$\begin{array}{rcl} S & ::= & [-] \mid \textbf{skip} \mid x{:=}e \mid S; c_2 \mid c_1; S \mid \\ & & \textbf{if } b \textbf{ then } S \textbf{ else } c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } S \\ & & \textbf{while } b \textbf{ do } S \mid \\ & & \textbf{await } b \textbf{ then } a \mid \\ & & c_1 \| c_2 \mid \\ & & \textbf{local } d \textbf{ in } S \end{array}$$

The important point in this definition is that $c_1 \| S$ is not regarded as a sequential context even when S is sequential, but we do allow "harmless" uses of parallelism inside sequential contexts. The key feature is that sequentiality of S ensures that when we fill the hole with a command we have the guarantee that the command will not be executed concurrently with any of the rest of the code in S.

3 Semantics

The operational behavior of parallel programs is described by the following transition system. Command configurations have the form $\langle c, s \rangle$, where c is a command and s is a state. A state s determines a (finite, partial) function $[\![s]\!]$ from identifiers to variables, and a variable corresponds to a state-dependent updateable integer value. We write $(s, x : v)$ for a state which is like s except that it associates the identifier x with a "fresh" variable with current value v; we write $[s \mid x : v']$ for the state which is like s except that the variable denoted by x in s has current value v'. Thus, for instance, the notation

$[(s, x : 0)|x : 1]$ denotes the same state as $(s, x : 1)$, and $[(s, x : 0)|y : 1]$ denotes the same state as $([s|y : 1], x : 0)$ when x and y are distinct identifiers.

$$\frac{}{\langle n, s\rangle \to n}$$

$$\langle x, s\rangle \to s(x)$$

$$\frac{\langle e_1, s\rangle \to \langle e_1', s\rangle}{\langle e_1 + e_2, s\rangle \to \langle e_1' + e_2, s\rangle}$$

$$\frac{\langle e_1, s\rangle \to n_1}{\langle e_1 + e_2, s\rangle \to \langle n_1 + e_2, s\rangle}$$

$$\frac{\langle e_2, s\rangle \to \langle e_2', s\rangle}{\langle n_1 + e_2, s\rangle \to \langle n_1 + e_2', s\rangle}$$

$$\frac{}{\langle \mathbf{true}, s\rangle \to \mathit{true}} \qquad \frac{}{\langle \mathbf{false}, s\rangle \to \mathit{false}}$$

$$\frac{\langle b_1, s\rangle \to \langle b_1', s\rangle}{\langle b_1 \wedge b_2, s\rangle \to \langle b_1' \wedge b_2, s\rangle}$$

$$\frac{\langle b_1, s\rangle \to \mathit{true}}{\langle b_1 \wedge b_2, s\rangle \to \langle b_2, s\rangle}$$

$$\frac{\langle b_1, s\rangle \to \mathit{false}}{\langle b_1 \wedge b_2, s\rangle \to \mathit{false}}$$

$$\frac{\langle e_1, s\rangle \to \langle e_1', s\rangle}{\langle e_1 = e_2, s\rangle \to \langle e_1' = e_2, s\rangle} \qquad \frac{\langle e_2, s\rangle \to \langle e_2', s\rangle}{\langle e_1 = e_2, s\rangle \to \langle e_1 = e_2', s\rangle}$$

$$\frac{}{\langle n_1 = n_2, s\rangle \to \mathit{true}} \quad (\text{if } n_1 = n_2)$$

$$\frac{}{\langle n_1 = n_2, s\rangle \to \mathit{false}} \quad (\text{if } n_1 \neq n_2)$$

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

$$\frac{\langle e, s\rangle \to \langle e', s\rangle}{\langle x{:=}e, s\rangle \to \langle x{:=}e', s\rangle} \qquad \frac{\langle e, s\rangle \to v}{\langle x{:=}e, s\rangle \to \langle x{:=}v, s\rangle}$$

$$\frac{}{\langle x{:=}v, s\rangle \to \langle \mathbf{skip}, [s \mid x : v]\rangle}$$

$$\frac{\langle c_1, s\rangle \to \langle c_1', s'\rangle}{\langle c_1; c_2, s\rangle \to \langle c_1'; c_2, s'\rangle} \qquad \frac{\langle c_1, s\rangle \texttt{term}}{\langle c_1; c_2, s\rangle \to \langle c_2, s\rangle}$$

$$\frac{\langle c_1, s\rangle \to \langle c_1', s'\rangle}{\langle c_1 \| c_2, s\rangle \to \langle c_1' \| c_2, s'\rangle} \qquad \frac{\langle c_2, s\rangle \to \langle c_2', s'\rangle}{\langle c_1 \| c_2, s\rangle \to \langle c_1 \| c_2', s'\rangle}$$

$$\frac{\langle c_1, s\rangle \texttt{term} \quad \langle c_2, s\rangle \texttt{term}}{\langle c_1 \| c_2, s\rangle \texttt{term}}$$

$$\frac{\langle b, s\rangle \to \langle b', s\rangle}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, s\rangle \to \langle \textbf{if } b' \textbf{ then } c_1 \textbf{ else } c_2, s\rangle}$$

$$\frac{\langle b, s\rangle \to t}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, s\rangle \to \langle \textbf{if } t \textbf{ then } c_1 \textbf{ else } c_2, s\rangle}$$

$$\frac{}{\langle \textbf{if } \mathit{true} \textbf{ then } c_1 \textbf{ else } c_2, s\rangle \to \langle c_1, s\rangle}$$

$$\frac{}{\langle \textbf{if } \mathit{false} \textbf{ then } c_1 \textbf{ else } c_2, s\rangle \to \langle c_2, s\rangle}$$

$$\frac{}{\langle \textbf{while } b \textbf{ do } c, s\rangle \to \langle \textbf{if } b \textbf{ then } c; \textbf{while } b \textbf{ do } c \textbf{ else skip}, \mathbf{s}\rangle}$$

$$\frac{\langle b, s\rangle \to^* \mathit{true} \quad \langle a, s\rangle \to^* \langle \textbf{skip}, s'\rangle}{\langle \textbf{await } b \textbf{ then } a, s\rangle \to \langle \textbf{skip}, s'\rangle}$$

$$\frac{\langle b, s\rangle \to^* \mathit{false}}{\langle \textbf{await } b \textbf{ then } a, s\rangle \to \langle \textbf{await } b \textbf{ then } a, s\rangle}$$

$$\frac{\langle e, s\rangle \to \langle e', s\rangle}{\langle \textbf{local } x = e \textbf{ in } c, s\rangle \to \langle \textbf{local } x = e' \textbf{ in } c, s\rangle}$$

$$\frac{\langle e, s\rangle \to v}{\langle \textbf{local } x = e \textbf{ in } c, s\rangle \to \langle \textbf{local } x = v \textbf{ in } c, s\rangle}$$

$$\frac{\langle c, (s, x : v)\rangle \to \langle c', (s', x : v')\rangle}{\langle \textbf{local } x = v \textbf{ in } c, s\rangle \to \langle \textbf{local } x = v' \textbf{ in } c, s'\rangle}$$

$$\frac{\langle c, (s, x : v)\rangle \texttt{term}}{\langle \textbf{local } x = v \textbf{ in } c, s\rangle \texttt{term}}$$

A *computation* of a command c is a finite sequence of transitions, ending in a terminal configuration, or an infinite sequence of transitions that is *fair* to all parallel component commands of c. We write $\langle c, s\rangle \to^* \langle c', s'\rangle$ to indicate a finite, possibly empty, sequence of transitions; and $\langle c, s\rangle \to^\omega$ to indicate the existence of a (weakly) fair infinite computation starting from a given

configuration. An *interactive computation* is a finite or infinite sequence of transitions in which the state may be changed between steps, representing the effect of other commands executing in parallel. There is an analogous notion of fairness for interactive computations. A computation is just an interference-free interactive computation, that is, an interactive computation in which no external changes occur.

Let $\mathcal{M}$ be a standard state-transformer semantics for programs, characterized operationally by:

$$\mathcal{M}(c) = \{(s, s') \mid \langle c, s\rangle \rightarrow^* \langle c', s'\rangle\mathtt{term}\} \ \cup \ \{(s, \bot) \mid \langle c, s\rangle \rightarrow^\omega\}.$$

We say that two programs c_1 and c_2 are *sequentially equivalent*, written $c_1 \equiv_{\mathcal{M}} c_2$, if and only if $\mathcal{M}(c_1) = \mathcal{M}(c_2)$.

Note that, as is well known, sequential equivalence is a congruence with respect to the sequential subset of our programming language. In fact, for all parallel programs c_1 and c_2, if $c_1 \equiv_{\mathcal{M}} c_2$ then $S[c_1] \equiv_{\mathcal{M}} S[c_2]$ also holds for all sequential contexts S. However, the analogous property fails to hold for parallel contexts, because in general $\mathcal{M}(c_1 \| c_2)$ cannot be determined solely on the basis of $\mathcal{M}(c_1)$ and $\mathcal{M}(c_2)$.

In previous work we developed a *transition trace* semantic model, suitable for shared-memory parallel programs as well as networks of asynchronous communicating processes, assuming weakly fair parallel execution. A transition trace of a program or process P is a finite or infinite sequence of steps, each step being a pair of states that represents the effect of a finite sequence of atomic actions performed by the process. A particular trace $(s_0, s'_0)(s_1, s'_1) \ldots (s_n, s'_n) \ldots$ of P represents a possible fair interactive computation of P in which the inter-step state changes (from s'_0 to s_1, and so on) are assumed to be caused by processes executing concurrently to P. A parallel program denotes a trace set closed under two natural conditions termed *stuttering* and *mumbling*, which correspond to our use of a step to represent finite sequences of actions: idle or stuttering steps of form (s, s) may be inserted into traces, and whenever two adjacent steps $(s, s')(s', s'')$ share the same intermediate state they can be combined to produce a mumbled trace which instead contains the step (s, s''). Traces are "complete", representing an entire interactive computation, rather than "partial" or "incomplete".

Trace semantics can be defined denotationally, and we note in particular that the traces of $P_1 \| P_2$ are obtained by forming fair merges of a trace of P_1 with as trace of P_2, and the traces of $P_1; P_2$ are obtained by concatenating

a trace of P_1 with a trace of P_2, closing up under stuttering and mumbling as required. The traces of **local** $x = v$ **in** P do not change the value of (the "global" version of) x, and are obtained by projection from traces of P in which the value of (the "local" version of) x is never altered between steps.

It is also possible to specify trace semantics based on different assumptions about the level of granularity of concurrent execution. It is generally regarded as realistic to assume *fine-grained* atomicity, i.e. that *reads* and *writes* to (simple) variables are executed atomically. It is often convenient to make the less realistic but simplifying assumption of *coarse-grained* atomicity, in which assignment commands and boolean expression evaluation are assumed to be executed indivisibly.

$$
\begin{array}{l}
\mathcal{T}(\textbf{skip}) = \{(s,s) \mid s \in \mathbf{S}\}^\dagger \\
\mathcal{T}(x{:=}e) = \{\alpha(s,[s \mid x : v]) \mid (\alpha, v) \in \mathcal{T}(e) \ \&\ s \in \mathbf{S}\}^\dagger \\
\mathcal{T}(c_1; c_2) = \{\alpha_1\alpha_2 \mid \alpha_1 \in \mathcal{T}(c_1) \ \&\ \alpha_2 \in \mathcal{T}(c_2)\}^\dagger \\
\mathcal{T}(c_1 \| c_2) = \{\alpha \mid \exists \alpha_1 \in \mathcal{T}(c_1), \alpha_2 \in \mathcal{T}(c_2).\ (\alpha_1, \alpha_2, \alpha) \in \mathit{fairmerge}\}^\dagger \\
\mathcal{T}(\textbf{await } b \textbf{ then } a) = \{(s,s') \in \mathcal{T}(a) \mid ((s,s), \mathit{true}) \in \mathcal{T}(b)\}^\dagger \\
\qquad\qquad\qquad\qquad \cup\ \{(s,s) \mid ((s,s), \mathit{false}) \in \mathcal{T}(b)\}^\omega \\
\mathcal{T}(\textbf{while } b \textbf{ do } c) = (\mathcal{T}(b)\mathcal{T}(c))^*\mathcal{T}(\neg b) \ \cup\ (\mathcal{T}(b)\mathcal{T}(c))^\omega \\
\mathcal{T}(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2) = \mathcal{T}(b)\mathcal{T}(c_1) \ \cup\ \mathcal{T}(\neg b)\mathcal{T}(c_2) \\
\mathcal{T}(\textbf{local } x = e \textbf{ in } c) = \{\alpha\beta) \mid (\alpha, v) \in \mathcal{T}(e) \ \&\ (\beta, x : v\rho) \in \mathcal{T}(c)\}
\end{array}
$$

We use the notation $(\beta, x{:}v_0v_1v_2\ldots)$ when β is a trace of form

$$(s_0, s_0')(s_1, s_1')(s_2, s_2')\ldots$$

and for each $i \geq 0$ we have $s_i(x) = s_i'(x)$, to stand for the trace

$$((s_0, x : v_0), (s_0', x : v_1))((s_1, x : v_1), (s_1', x : v_2))\ldots$$

In this "expanded" trace x is treated as a fresh variable and is never altered between steps. Thus the semantic clause for **local** $x = e$ **in** c captures the intention that e is evaluated to yield an initial value v for the fresh variable, then c is executed in the expanded state, but only the "global" part of the state is visible to other processes, since the local x is only in scope for c.

The closure properties ensure that trace semantics is *fully abstract* with respect to a notion of behavior which assumes that we can observe the state during execution. As a result trace semantics supports compositional reasoning about safety and liveness properties. Safety properties typically assert

that no "bad" state ever occurs when a process is executed, without interference, from an initial state satisfying some pre-condition. A liveness property typically asserts that some "good" state eventually occurs. When two processes have the same trace sets it follows that they satisfy identical sets of safety and liveness properties, in all parallel contexts.

Both coarse- and fine-grained trace semantics interpret conditional atomic actions **await** b **then** a as atomic, for obvious reasons. The coarse-grained trace semantics, which we will denote $\mathcal{T}_{coarse}$, assumes that assignment, and boolean expression evaluation, are atomic actions executed indivisibly. When using coarse-grained semantics one can safely assume (for instance) that algebraic laws of arithmetic can be employed to simplify reasoning about program behavior. For instance, in coarse-grained trace semantics the assignments $x{:=}x + x$ and $x{:=}2 \times x$ are equivalent. The fine-grained semantics, denoted $\mathcal{T}_{fine}$, assumes only that reads and writes to simple variables are atomic. This is closer in practice to conventional implementations of concurrent programs, but we are no longer permitted to assume with impunity that algebraic laws of expression equivalence remain valid. For instance, the assignments $x{:=}x + x$ and $x{:=}2 \times x$ are not equivalent in fine-grained trace semantics, and this reflects the fact that the former reads the value of x twice, so that if x is changed during execution (say from 0 to 1), the value assigned may be $0, 1$ or 2, whereas the latter assignment (under the same circumstances) would assign either 0 or 2.

Both coarse and fine versions of trace semantics can be defined denotationally, and each induces a notion of program equivalence, as illustrated above. Despite their characterizations as fine *vs.* coarse, these two trace semantic variants induce *incomparable* notions of semantic equivalence. For instance, we have already seen a pair of programs which are equivalent in coarse-grained semantics but not in fine-grained; and the programs $x{:=}x + 1$ and **local** $t = 0$ **in** $(t{:=}x; t{:=}t + 1; x{:=}t)$ are equivalent in fine-grained but not in coarse-grained semantics.

Each trace equivalence is a congruence for the entire parallel language, so that whenever $c_1 \equiv_{\mathcal{T}} c_2$ it follows that $C[c_1] \equiv_{\mathcal{T}} C[c_2]$ holds for all parallel contexts C. Moreover, $c_1 \equiv_{\mathcal{T}} c_2$ implies $c_1 \equiv_{\mathcal{M}} c_2$, but the converse implication is not generally valid.

4 Concurrent reads and writes

To prepare the ground, we first need to define for each parallel program c the *multiset* $\texttt{reads}(c)$ of identifier occurrences which appear free in non-atomic sub-expressions of c; and the *set* $\texttt{writes}(c)$ of identifier occurrences in c which occur free in c as targets of assignments. It is vital here, as suggested by the terminology, to keep track of how many references the program makes, and of what kinds of reference they are, to each identifier. The definition of $\texttt{reads}(c)$ and $\texttt{writes}(c)$ is a straightforward structural induction, using standard multiset operations.

For our purposes, we may think of a multiset as a set of identifiers equipped with a non-negative multiplicity count. In the empty multiset every identifier has multiplicity 0. When M_1 and M_2 are multisets, we let $M_1 \cup_+ M_2$ be the multiset union in which multiplicities are added, and $M_1 \cup_{max} M_2$ be the multiset union in which multiplicities are combined using *max*. Given a multiset M and a set X of identifiers, we define $M - X$ to be the multiset obtained from M by removing all occurrences of identifiers in X. We write $\{\!|x|\!\}$ for the singleton multiset containing a single occurrence of x.

$$
\begin{aligned}
&\mathit{reads}(n) = \{\!|\ |\!\} \\
&\mathit{reads}(x) = \{\!|x|\!\} \\
&\mathit{reads}(e_1 + e_2) = \mathit{reads}(e_1) \cup_+ \mathit{reads}(e_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reads}(x = e) = \mathit{reads}(e) \\
&\mathit{reads}(d_1; d_2) = \mathit{reads}(d_1) \cup_{max} (\mathit{reads}(d_2) - \mathit{dec}(d))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reads}(\textbf{skip}) = \{\!|\ |\!\} \\
&\mathit{reads}(x{:=}e) = \mathit{reads}(e) \\
&\mathit{reads}(c_1; c_2) = \mathit{reads}(c_1 \| c_2) = \mathit{reads}(c_1) \cup_{max} \mathit{reads}(c_2) \\
&\mathit{reads}(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2) = \mathit{reads}(b) \cup_{max} (\mathit{reads}(c_1) \cup_{max} \mathit{reads}(c_2)) \\
&\mathit{reads}(\textbf{while } b \textbf{ do } c) = \mathit{reads}(b) \cup_{max} \mathit{reads}(c) \\
&\mathit{reads}(\textbf{await } b \textbf{ then } a) = \{\!|\ |\!\} \\
&\mathit{reads}(\textbf{local } d \textbf{ in } c) = \mathit{reads}(d) \cup_{max} (\mathit{reads}(c) - \mathit{dec}(d))
\end{aligned}
$$

$writes(\textbf{skip}) = \{\}$
$writes(x{:=}e) = \{x\}$
$writes(c_1; c_2) = writes(c_1 \| c_2) = writes(c_1) \cup writes(c_2)$
$writes(\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2) = writes(c_1) \cup writes(c_2)$
$writes(\textbf{while } b \textbf{ do } c) = writes(c)$
$writes(\textbf{await } b \textbf{ then } a) = writes(a)$
$writes(\textbf{local } d \textbf{ in } c) = writes(c) - dec(d)$

We now state some fundamental properties of trace semantics, which formalize the sense in which the behavior of a parallel program depends only on the values of its free identifiers. We say that two states s and s' *agree* on a set X of identifiers if for all $x \in X$, $s(x) = s'(x)$.

Agreement Theorem

1. Let α be a trace of c and (s, s') be a step of α. Then s agrees with s' on all identifiers not in $writes(c)$.

2. Let $(s_0, s'_0)(s_1, s'_1) \ldots (s_n, s'_n) \ldots$ be a trace of c. Then for every sequence of states $t_0, t_1, \ldots, t_n, \ldots$ such that for all $i \geq 0$, t_i agrees with s_i on $X \supseteq reads(c)$, there is a trace

$$(t_0, t'_0)(t_1, t'_1) \ldots (t_n, t'_n) \ldots$$

of c such that for all $i \geq 0$, t'_i agrees with t_i on $X \cup writes(c)$.

Next we define, for each parallel context C, the pair $\texttt{crw}(C) = (R, W)$ where R is the *set* of identifiers which occur free in evaluation contexts concurrent to a hole of C, and W is the set of identifiers occurring free in assigning contexts concurrent to a hole. As usual the definition is inductive. (It suffices to work with sets here rather than multisets, since what matters for our present purposes is whether or not the context may change an identifier's value concurrently while whatever command occupies the hole is running, not how many times the context may do this; even once is bad enough.)

$crw([-]) = crw(\textbf{skip}) = crw(x{:=}e) = (\{\}, \{\})$
$crw(C; c_2) = crw(c_1; C) = crw(C)$
$crw(\textbf{if } b \textbf{ then } c_1 \textbf{ else } C) = crw(\textbf{if } b \textbf{ then } C \textbf{ else } c_2) = crw(C)$
$crw(\textbf{while } b \textbf{ do } C) = crw(C)$
$crw(\textbf{await } b \textbf{ then } a) = (\{\}, \{\})$
$crw(c \| C) = crw(C \| c) = (R \cup reads(c), W \cup writes(c))$,
where $(R, W) = crw(C)$
$crw(\textbf{local } d \textbf{ in } C) = crw(C)$

5 Transfer principles

Having set up the relevant background definitions we can now present the transfer principle to which we have been leading. The first one is almost too obvious to include:

Theorem 0

If A is an atomic context and $a_1 \equiv_{\mathcal{M}} a_2$, then $A[a_1] \equiv_{\mathcal{T}} A[a_2]$.

Proof The traces of **await** b **then** a depend only on the "atomic" traces of a, i.e. on the traces of a which represent uninterrupted complete executions; and (s, s') is an atomic trace of a iff $(s, s') \in \mathcal{M}(a)$.

The next transfer principle identifies conditions under which sequential equivalence of code fragments can safely be relied upon to establish trace equivalence of parallel programs.

Theorem 1

If $\texttt{free}(c_1) \cup \texttt{free}(c_2) \subseteq bound(C))$, and

$$|\texttt{reads}(c_i) \cap W| + |\texttt{writes}(c_i) \cap R| = 0, \ i = 1, 2$$

where $(R, W) = \texttt{crw}(C)$, then

$$c_1 \equiv_{\mathcal{M}} c_2 \Rightarrow C[c_1] \equiv_{\mathcal{T}} C[c_2].$$

Proof

It is worth noting here that the provisos built into this theorem are essential. If we omit the local declaration around the context the result becomes invalid, since the assumption that c_1 and c_2 are sequentially equivalent is not strong enough to imply that c_1 and c_2 are trace equivalent. And if we try to use the code fragments in a context with which it interacts non-trivially

again the result fails: when c_1 and c_2 are sequentially equivalent it does not follow that **local** d **in** $(c\|c_1)$ and **local** d **in** $(c\|c_2)$ are trace equivalent for all c, even if d declares all of the free identifiers of c_1 and c_2. A specific counterexample is obtained by considering the commands

$$\begin{array}{ll} c_1: & x{:=}x+1;\ x{:=}x+1 \\ c_2: & x{:=}x \end{array}$$

We have $\mathit{reads}(c_i) = \{\!|x|\!\}$, $\mathit{writes}(c_i) = \{x\}$. Let C be the context

$$\textbf{local } x = 0 \textbf{ in } (([-]\|x{:=}2);\ y{:=}x).$$

Then $\mathit{bound}(C) = \{x\}$ and $\mathit{crw}(C) = (\{\}, \{x\})$. Using the notation of the theorem, we have

$$|\mathit{reads}(c_i) \cap W| = 1, \quad |\mathit{writes}(c_i) \cap R| = 0$$

so that the assumption is violated. And it is easy to see that $c_1 \equiv_{\mathcal{M}} c_2$, but

$$\begin{array}{l} C[c_1] \equiv_{\mathcal{T}} y{:=}0 \textbf{ or } y{:=}1 \textbf{ or } y{:=}2 \\ C[c_2] \equiv_{\mathcal{T}} y{:=}0 \textbf{ or } y{:=}2 \end{array}$$

so that $C[c_1] \not\equiv_{\mathcal{T}} C[c_2]$.

Another counterexample shows that the other half of the assumption cannot be relaxed. Consider

$$\begin{array}{ll} c_1: & x{:=}1;\ \textbf{while true do skip} \\ c_2: & x{:=}2;\ \textbf{while true do skip} \end{array}$$

Let C be the context

$$\textbf{local } x = 0 \textbf{ in } ([-]\|y{:=}x).$$

Then $\mathit{bound}(C) = \{x\}$, $\mathit{free}(c_i) = \mathit{writes}(c_i) = \{x\}$, and $\mathit{reads}(c_i) = \{\!|\,|\!\}$. Moreover $c_1 \equiv_{\mathcal{M}} c_2$, since $\mathcal{M}(c_i) = \{(s, \bot) \mid s \in \mathbf{S}\}$ $(i = 1, 2)$. We have

$$|\mathit{reads}(c_i) \cap W| = 0, \quad |\mathit{writes}(c_i) \cap R| = 1$$

so that the assumption is violated again. And we also have

$$\begin{array}{l} C[c_1] \equiv_{\mathcal{T}} (y{:=}0 \textbf{ or } y{:=}1); \textbf{while true do skip} \\ C[c_2] \equiv_{\mathcal{T}} (y{:=}0 \textbf{ or } y{:=}2); \textbf{while true do skip} \end{array}$$

so that $C[c_1] \not\equiv_{\mathcal{T}} C[c_2]$.

The above theorem is always applicable in the special case where the context is sequential. We therefore state the following:

Corollary If S is a sequential context, and $\texttt{free}(c_1) \cup \texttt{free}(c_2) \subseteq \mathit{bound}(S)$, then

$$c_1 \equiv_{\mathcal{M}} c_2 \Rightarrow S[c_1] \equiv_{\mathcal{T}} S[c_2].$$

Proof When S is sequential we can show that $\mathit{crw}(S) = (\{\}, \{\})$.

Many simple laws of sequential equivalence are well known, including de Bakker's family of laws for sequential equivalence of sequences of assignment commands, but most of these laws fail to hold in the parallel setting. Our result shows the extent to which such laws may safely be used when reasoning about the safety and liveness properties of *parallel* programs, pointing out sufficient conditions under which sequential analysis of key code fragments is enough to ensure correctness of a parallel program. We illustrate the utility of this transfer theorem by considering some examples. Notably, this result can be used to simplify reasoning about a protocol in which concurrent processes interact periodically to ensure some desired global invariant.

Finally, we now consider what requirements must be satisfied in order to safely employ coarse-grained trace-based reasoning in establishing fine-grained equivalences. This may be beneficial, since for a given program (or program fragment) the coarse-grained trace set forms a (usually proper) subset of the fine-grained trace set and may therefore permit a streamlined analysis. This is especially important for code which may be executed concurrently, since it may help minimize the combinatorial analysis. Indeed, Andrews supplies a series of examples of protocols in which a "fine-grained" solution to a parallel programming problem (such as mutual exclusion) is derived by syntactic transformation from a "coarse-grained" solution whose correctness is viewed as easier to establish. Common to all of these examples is the desire to appeal to coarse-grained reasoning when trying to establish correctness in the fine-grained setting. Our requirements are based on a so-called "at-most-once" property that Andrews uses to facilitate analysis of a collection of mutual exclusion protocol designs.

The relevant definitions from Andrews, adapted to our setting, are as follows:

- An expression b (or e) has the at-most-once property if it refers to at most one identifier that might be changed by another process while the

expression is being evaluated, and it refers to this identifier at most once.

- An assignment $x{:=}e$ has the at-most-once property if either e has the at-most-once property and x is not read by another process, or if e does not refer to any identifier that may be changed by another process.

- A command c has the at-most-once property if every assignment and boolean test occurring non-atomically in c has the at-most-once property.

An occurrence is atomic if it is inside a subcommand of form **await** b **then** a, and all other occurrences are non-atomic.

Andrews's methodology is based on the idea that if a command has the at-most-once property then it suffices to assume coarse-grained execution when reasoning about its behavior, since there will be no discernible difference with fine-grained execution. We will couch our transfer principle in slightly more general terms.

Theorem 2
If $\texttt{free}(c_1) \cup \texttt{free}(c_2) \subseteq \texttt{bound}(C)$, and either

$$|\texttt{reads}(c_i) \cap W| = 0$$

or

$$|\texttt{reads}(c_i) \cap W| = 1 \ \& \ |\texttt{writes}(c_i) \cap (R \cup W)| = 0, \ i = 1, 2$$

where $(R, W) = \texttt{crw}(C)$, then

$$c_1 \equiv_{\mathit{coarse}} c_2 \Rightarrow C[c_1] \equiv_{\mathit{fine}} C[c_2].$$

Proof It is easy to see that if $c_1 \equiv_{\mathit{coarse}} c_2$ then $\mathit{reads}(c_1) = \mathit{reads}(c_2)$ and $\mathit{writes}(c_1) = \mathit{writes}(c_2)$. Thus, whichever of the two cases holds for c_1, the same case must hold for c_2, and vice versa. Suppose, first, that $c_1 \equiv_{\mathit{coarse}} c_2$, and $\mathit{free}(c_i) \subseteq \mathit{bound}(C)$, and $\texttt{reads}(c_i) \cap W = \{\}$, for $i = 1, 2$, where $(R, W) = \mathit{crw}(C)$. That is, neither c_1 nor c_2 refers to any identifier which may be changed concurrently. (insert details)

Now suppose that $c_1 \equiv_{\mathit{coarse}} c_2$ and that $\texttt{reads}(c_i) \cap W = \{x\}$ and $\texttt{writes}(c_i) \cap (R \cup W) = \{\}$. That is, both c_1 and c_2 refer to x exactly once, this is the only identifier used by c_1 and c_2 that is also subject to concurrent change by the rest of the context, and no assignments made by c_1 or c_2 affect any identifier used by the rest of the context. (insert details)

Again we remark that the built-in provisos imposing locality and the at-most-once property cannot be dropped. Firstly, every program has the at-most-once property, trivially, for the context $[-]$. But the assumption that $c_1 \equiv_{coarse} c_2$ is insufficient to ensure that $c_1 \equiv_{fine} c_2$. Thus the result becomes invalid if we omit the localization around the context. To illustrate the need for the at-most-once assumption, let the programs c_1 and c_2 be $y{:=}x + x$ and $y{:=}2 \times x$. These programs are clearly coarsely equivalent. Let C be the context

$$\mathbf{local}\ x = 0; y = 0\ \mathbf{in}\ (([-] \| x{:=}1);\ z{:=}y).$$

Of course c_1 refers twice to x, which is assigned to by the context concurrently; c_1 does not satisfy the at-most-once property for C. Moreover we can see that

$$\begin{array}{lll} \mathbf{local}\ x = 0; y = 0\ \mathbf{in}\ ((y{:=}x + x \| x{:=}1); z{:=}y) & \equiv_{fine} & z{:=}0\ \mathbf{or}\ z{:=}1\ \mathbf{or}\ z{:=}2 \\ \mathbf{local}\ x = 0; y = 0\ \mathbf{in}\ ((y{:=}2 \times x \| x{:=}1); z{:=}y) & \equiv_{fine} & z{:=}0\ \mathbf{or}\ z{:=}2 \end{array}$$

so that $C[c_1] \not\equiv_{fine} C[c_2]$. Although our programming language did not include a non-deterministic choice operator $c_1\ \mathbf{or}\ c_2$ it is convenient to use it as here, to specify a command that behaves like c_1 or like c_2; in terms of trace sets we have $\mathcal{T}(c_1\ \mathbf{or}\ c_2) = \mathcal{T}(c_1) \cup \mathcal{T}(c_2)$, a similar equation holiding in coarse- and in fine-grained versions.

Also note that the other way for the assumption to fail is when c_1 (say) both reads and writes to a concurrently accessed identifier. For instance, let c_1 be $x{:=}x$ and c_2 be $\langle x{:=}x \rangle$. Let C be the context

$$\mathbf{local}\ x = 0\ \mathbf{in}\ (([-] \| x{:=}1); y{:=}x)$$

Then we have $|reads(c_i) \cap W| = 1$ and $|writes(c_i) \cap (R \cup W)| > 0$. And $c_1 \equiv_{coarse} c_2$. But $C[c_1] \equiv_{fine} y{:=}0\ \mathbf{or}\ y{:=}1$, and $C[c_2] \equiv_{fine} y{:=}1$.

It is also worth remarking that the above principle cannot be strengthened by weakening the assumption that c_1 and c_2 are coarsely equivalent to the assumption that c_1 and c_2 are sequentially equivalent. For example, let c_1 and c_2 be

$$y{:=}1;\ \mathbf{while\ true\ do\ skip}$$

and

$$y{:=}2;\ \mathbf{while\ true\ do\ skip}.$$

Let C be the context **local** $y = 0$ **in** $([-]\|z{:=}y)$. Then we have $\mathit{reads}(c_i) = \{\}$, $\mathit{writes}(c_i) = \{y\}$, $\mathit{crw}(C) = (\{y\}, \{z\})$, $\mathit{bound}(C) = \{y\}$. Moreover, $c_1 \equiv_{\mathcal{M}} c_2$ holds, since $\mathcal{M}(c_i) = \{(s, \bot) \mid s \in \mathbf{S}\}$, $i = 1, 2$. However,

$$C[c_1] \equiv_{\mathit{fine}} (z{:=}0 \textbf{ or } z{:=}1)$$

and

$$C[c_2] \equiv_{\mathit{fine}} (z{:=}0 \textbf{ or } z{:=}2).$$

The coarse- to fine-grained transfer theorem given above generalizes some more *ad hoc* arguments based on occurrence-counting in Andrews's book, resulting in a single general principle in which the crucial underlying provisos are made explicit. To make the connection with Andrews's examples more precise, note the following special cases of our theorem, which appear in paraphrase in Andrews:

- If b refers at most once to identifiers written concurrently (by the context), then **await** b **then skip** can be replaced by **while** $\neg b$ **do skip** (throughout the program). This rule is used to justify replacement of a conditional atomic action with a (non-atomic) busy-wait loop.

- If $x{:=}e$ has the at-most-once property (for the context) then the assignment $x{:=}e$ can be replaced by its atomic version **await true then** $x{:=}e$ (throughout the program). This rule is used to simplify reasoning about the potential for interaction between processes.

6 Conclusions

Our transfer principles can be seen as supplying a semantic foundation for some of the ideas behind Andrews's protocol analysis, and a potential basis for further generalization and the systematic development of techniques to permit easier design and analysis of parallel programs.

These results permit the use of a simpler, more abstract semantics, together with a notion of semantic equivalence which is easier to establish, to facilitate reasoning about the behavior of a parallel system.

References

[1] Andrews, G., *Concurrent Programming: Principles and Practice*. Benjamin/Cummings (1991).

[2] Brookes, S., *Full abstraction for a shared-variable parallel language*. Information and Computation, **127**(2), 145–163 (June 1996).

[3] Brookes, S., *The essence of Parallel Algol*. Proc. 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 164–173 (1996). To appear in *Information and Computation*.

[4] Brookes, S., *Idealized CSP: Combining Procedures with Communicating Processes*. Mathematical Foundations of Programming Semantics, 13^{th} Conference, March 1997. Electronic Notes in Theoretical Computer Science 6, Elsevier Science (1997).
URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

[5] Brookes, S., *Communicating Parallel Processes*. In: *Millenium Perspectives in Computer Science*, Proceedings of the Oxford-Microsoft Symposium in honour of Professor Sir Antony Hoare, edited by Jim Davies, Bill Roscoe, and Jim Woodcock, Palgrave Publishers (2000).

[6] de Bakker, J., *Axiom systems for simple assignment statements*. In *Symposium on Semantics of Algorithmic Languages*, edited by E. Engeler. Springer-Verlag LNCS vol. 181, 1–22 (1971).

[7] Dingel, J., *Systematic parallel programming*. Ph. D. thesis, Carnegie Mellon University, Department of Computer Science (May 2000).

[8] Francez, N., *Fairness*. Springer-Verlag (1986).

[9] Jones, C. B., *Tentative steps towards a development method for interfering programs*, ACM Transactions on Programming Languages and Systems, 5(4):576–619 (1983).

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

[11] Park, D., *Concurrency and automata on infinite sequences*. Springer LNCS vol. 104 (1981).

7 Appendix

$\langle \textbf{skip} \rangle = \textbf{skip}$
$\langle x{:=}e \rangle = \textbf{await true then } x{:=}e$
$\langle c_1; c_2 \rangle = \langle c_1 \rangle; \langle c_2 \rangle$
$\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \rangle = \textbf{if } \langle b \rangle \textbf{ then } \langle c_1 \rangle \textbf{ else } \langle c_2 \rangle$
$\langle \textbf{while } b \textbf{ do } c \rangle = \textbf{while } \langle b \rangle \textbf{ do } \langle c \rangle$
$\langle \textbf{await } b \textbf{ then } a \rangle = \textbf{await } b \textbf{ then } a$
$\langle c_1 \| c_2 \rangle = \langle c_1 \rangle \| \langle c_2 \rangle$
$\langle \textbf{local } x = e \textbf{ in } c \rangle = \textbf{local } x = \langle e \rangle \textbf{ in } \langle c \rangle$

$$\langle c, s \rangle \rightarrow_{coarse} \langle c', s' \rangle \;\; \Leftrightarrow_{def} \;\; \langle \langle c \rangle, s \rangle \rightarrow \langle \langle c' \rangle, s' \rangle$$

$$\langle \langle e \rangle, s \rangle \rightarrow v \;\; \Leftrightarrow_{def} \;\; \langle e, s \rangle \rightarrow^* v$$

$$\langle e, s \rangle \rightarrow_{coarse} v \;\; \Leftrightarrow_{def} \;\; \langle e, s \rangle \rightarrow^* v$$