COMMUTATIVITY AND PARALLEL PROGRAM DESIGN

Josyula R. Rao

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-36 November 1991

Commutativity and Parallel Program Design

Josyula R. Rao*
Department of Computer Sciences
The University of Texas at Austin

November 23, 1991

Abstract

One of the fundamental problems in the theory of parallelism is the development of a simple programming methodology that aids a programmer in implementing a given specification in a modular and compositional fashion. Existing techniques are helpful if the specification consists of safety properties alone and the actions of the different processes of the program satisfy a commutativity condition introduced by Richard Lipton. A different commutativity condition proposed recently by Jayadev Misra goes further in allowing progress properties in the specification provided the process implementing the progress property does so in a *wait-free* manner.

In this paper, we begin by showing that Lipton and Misra commutativity are incomparable notions and capture fundamental though complementary aspects of asynchrony. We propose two new definitions of commutativity that combine essential elements of Lipton and Misra commutativity. Our first definition is expressed in terms of partial functions and is derived by operational considerations. Our second definition is expressed in terms of predicate transformers and is derived by purely formal considerations. Our definitions are related in the sense that the second definition is weaker than the first.

Our definitions have three important consequences. Firstly, they form the basis of a simple compositional methodology for parallel program design. This methodology is better than Lipton's, in that *both* safety and progress properties are permitted in the specification of a parallel program. It improves on the methodology proposed by Misra by removing the constraint of *wait-freedom* on processes implementing progress properties. Secondly, from the perspective of a programming language designer, these definitions suggest constraints on the variables that can be shared between programs. Thirdly, a set of processes that satisfy our definitions also satisfy Misra's definition of commutativity: this means that the set of processes is loosely-coupled and can be implemented on a shared memory multiprocessor with an *asynchronous cache coherence scheme* thus enabling the scaling up of shared memory architectures.

Keywords: distributed computing, distributed systems, program correctness, program specification

*This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658-065 and by grant ONR 26-0679-4200 from the Office of Naval Research.

0 Introduction

0.1 Background and Motivation

In [Lip75], Richard Lipton introduced the idea of *reduction* to enable the verification of a special class of safety properties of parallel programs, namely partial correctness and deadlock–freedom. The notion of *reduction* was defined in terms of a commutativity condition[0] on the actions of the processes comprising the parallel program. In later papers, Doeppner [Doe77] and Lamport and Schneider [LS89] used Lipton commutativity to generalize and extend Lipton's results to cover *all* safety properties. However, no headway was made in the task of proving progress properties of parallel programs.

In [Mis91], Jayadev Misra introduced the idea of *loose–coupling*. The aim of the work was to facilitate the design of parallel programs intended for execution on a shared–memory architecture. The key idea was to break the *tight–coupling* of shared–memory parallelism by imposing the constraints of a distributed, message–passing architecture on the programmer. These constraints were phrased in terms of a commutativity condition[1] on the actions of the processes comprising the parallel program. The importance of loose–coupling is borne out by two consequences. Firstly, one can derive a parallel programming methodology that treats *both* safety and progress properties. Specifically, if a progress property is implemented in a *wait–free* manner by a component process then the property holds in any loosely–coupled collection of processes which contain the component process and satisfy a simple stability property. Secondly, one can propose an asynchronous scheme to the problem of cache–coherence which enables the scaling up of shared–memory multiprocessors.

0.2 Contributions

In this paper, we integrate these two streams of research to develop a compositional theory for parallelism. Our investigations reveal that Lipton commutativity and Misra commutativity are incomparable notions, in the sense, that there exist actions that commute with respect to one definition and not the other. Moreover, each definition captures a fundamental aspect of asynchrony that is different from the other. While Lipton commutativity captures the fact that the environment of a process cannot *enable* the actions of a process, Misra commutativity imposes the requirement that the environment cannot *disable* the actions of a process.

Using these definitions of commutativity as a basis, we propose a new and operational definition of commutativity of actions. Our definition is stronger than both Lipton's and Misra's definition and combines the essential elements of both. An important consequence of our definition is a compositional programming methodology for the design of parallel programs. Specifically, we use an operational argument to show that if two processes satisfy our notion of commutativity, then the progress properties of each are maintained when the two processes are composed and executed in parallel. It is important to note that unlike [Lip75, Doe77, LS89], we allow *both* safety and progress properties in specifications. Furthermore, unlike [Mis91], we do not require any additional assumptions (in particular, *wait–freedom*) in the preservation of progress properties.

The approach adopted in the research outlined above is an *operational* one. In studying a phenomenon as fundamental as asynchrony, we chose to be operational because we did not wish to restrict the scope of our study by committing ourselves to a programming model. We were willing to tolerate a limited proliferation of detail rather than overlook some basic theorems in the process. The complexity of our operational argument suggested two things to us: firstly, that the time had come to abandon the operational approach and secondly, that a concept of fundamental importance was the notion of the

[0] Henceforth referred to as *Lipton commutativity*.

[1] Henceforth referred to as *Misra commutativity*.

closure of a program [Lam87, Coh91]. The *closure* of a program F, with respect to a predicate P is the strongest predicate Q, implied by P, that is stable in F. Much of the complexity of our operational approach could be eliminated by postulating this at the very outset.

We define the closure of a program by means of a predicate transformer. We use this definition along with the theory of UNITY [CM88] to *derive* a second definition of commutativity. Our definition is in terms of predicate transformers and is *weaker* than the operational definition of commutativity proposed above. Inspite of being weaker, we show that the definition is sufficient to ensure a compositional methodology for the design of parallel programs. Specifically, the definition of commutativity is sufficient to ensure that the closures of the safety and progress properties of two programs that commute are not violated when the programs are executed in parallel.

In addition to providing a compositional programming methodology, our definitions have three important ramifications. Firstly, from the perspective of the programming language designer, they suggest constraints on the variables that can be shared between programs. For example, it is easy to see that the commutativity conditions are satisfied if all shared variables are restricted to be *logic variables* [CT90]. Secondly, our definitions imply Misra's definition of commutativity. This means that any two processes that satisfy our notions of commutativity are loosely–coupled. It has been demonstrated in [Mis91] that the cache coherence problem vanishes for a set of loosely–coupled processes executing on a shared memory multiprocessor. In particular, one can use an asynchronous cache–coherence scheme (such a scheme is proposed in [Mis91]) and this enables the scaling up of shared memory multiprocessors. Thirdly, from the perspective of methodology, they enable us to derive a set of rules of composition that can be used to implement a composite specification by several component programs [Coh91].

1 Programming Formalism and Logic

Our computational model is based on UNITY [CM88]. In our approach, a program is a syntactic object that consists of two sections: a **declare** section consisting of variable declarations and an **assign** section consisting of a finite and non–empty set of statements.

Operationally, program execution can begin in *any* state and proceeds by repeatedly selecting a statement from the **assign** section and executing it with the *fairness* constraint that in every infinite execution, each statement is selected and executed infinitely often. In this model, parallelism is captured by a fair, interleaved execution of the statements of the program with each statement being executed atomically.

We permit our programs to have *conditional* statements of the form

$$S :: x := e \textbf{ if } b$$

where x is a list of variables, e is a list of expressions and b is a boolean condition. We say that a statement S is *defined* in a state iff condition b holds in that state. As would be expected, if S is selected for execution in a state in which it is defined, executing S results in a new state in which the list of variables x is assigned the values of the corresponding expressions in list e. But, what if S is selected in a state in which condition b is false ? Technically speaking, such a selection is allowed by our notion of fairness. While S is undefined in such a state, our execution should record the fact that S was selected for execution. We resolve this dilemma by postulating that if an *undefined* statement is selected for execution, then its execution has the same effect as a **skip** statement: that is, the resulting new state is the same as the old one.

We are interested in two classes of properties of programs, namely *safety* and *progress* properties.

- *Safety properties:* We express safety using the **unless** operator of UNITY. For state predicates, X and Y, the operational interpretation of X **unless** Y is as follows: in any execution, once X becomes true it remains true as long as Y is false. A special case is X **unless** $false$ (also written **stable** X) that captures the idea that once X is true, it remains true.

  Formally, for a program F,

$$X \textbf{ unless } Y \textbf{ in } F \;\equiv\; \langle \forall s : s \in F : [X \wedge \neg Y \Rightarrow \textbf{wp}.s.(X \vee Y)] \rangle.$$

- *Progress properties:* We express progress in terms of the $\mapsto$ (read, *leads–to*) operator of UNITY. For state predicates X and Y, we say $X \mapsto Y$ if in any *fair* execution, once X is true, Y is or will become true. Formally,

$$X \textbf{ ensures } Y \;\equiv\; (X \textbf{ unless } Y) \wedge \langle \exists t :: [X \wedge \neg Y \Rightarrow \textbf{wp}.t.Y] \rangle.$$

$$\begin{array}{rcl} X \textbf{ ensures } Y & \Rightarrow & X \mapsto Y \\ X \mapsto Y, Y \mapsto Z & \Rightarrow & X \mapsto Z \\ \langle \forall X : X \in W : X \mapsto Y \rangle & \Rightarrow & (\langle \exists X : X \in W : X \rangle \mapsto Y) \end{array}$$

Our goal in this paper is the development of a *compositional* programming methodology for the parallel program design. A first step in such an exercise is to define a notion of program composition.

- *Program composition:* As in UNITY, we compose two programs F and G by taking the union of their corresponding **assign** sections. The composed program is denoted by $F \;[\!]\; G$.

A programming methodology is said to be *compositional* if one is able to infer the safety and progress properties of a composite program from the corresponding properties of its components. In the framework of UNITY, the following result on composing safety properties is known.

$$X \textbf{ unless } Y \textbf{ in } F \;[\!]\; G \;\equiv\; X \textbf{ unless } Y \textbf{ in } F \;\wedge\; X \textbf{ unless } Y \textbf{ in } G.$$

For a long time no such result was known for composing progress properties. In a recent paper [Mis91], Jayadev Misra has suggested a theorem for composing progress properties. The theorem assumes that the process that implements the progress property does so in a *wait–free* manner. We believe such an assumption is not required.

In the remainder of the paper, we demonstrate two approaches — an operational approach and a predicate transformer based approach — to developing a compositional methodology of parallel programs that deals with *both* safety and progress properties.

2 An Operational Approach

2.1 Preliminaries

We will concentrate on three kinds of objects: *states*, *partial functions* and *sequences of partial functions*.

Assume that we are given a *set of states*, Σ, and a set of *partial functions*. Both the domain and the range of a partial function will be Σ. We denote a representative state of Σ by the letter s. We use the letters f and g to represent partial functions.

Informally, a *state* is intended to capture the notion of a *program state*. In the same vein, a *partial function* is intended to capture the notion of a statement. Such an abstraction allows us to assert that a statement is defined in a state if and only if the corresponding partial function is defined at the corresponding state of Σ.

The third category of interest will be *finite sequences of partial functions*[2]. The functions in the sequence will be applied to their argument in the order read, that is, from left to right. The empty sequence will be denoted by "$<>$". We will use the letters A, B, M, N, X, Y to represent arbitrary sequences.

We now introduce notation for expressing properties about these objects and present rules that will be useful in manipulating such properties.

Definition 0 inductively defines the application of a sequence of functions to a state. The base case states that the empty sequence applied to a state returns the same state. The inductive step is as expected.

Definition 0 (function application, ".")

1. $<> .s \equiv s$
2. $fA.s \equiv A.(f.s)$

Definition 1 inductively defines the concept of *definedness* of sequences of functions. The base case states that the empty sequence is defined at every state. The inductive step is as expected.

Definition 1 (definedness, "[]")

1. $[s]<>$.
2. $[s]fA \equiv [s]f \wedge [f.s]A$.

Equality of two sequences is given by definition 2.

Definition 2 (equality, "$\overset{s}{\sim}$")

$$(A \overset{s}{\sim} B) \equiv [s]A \wedge [s]B \wedge (A.s = B.s).$$

Theorem 0 *The relation $\overset{s}{\sim}$ is an equivalence relation on the set of sequences $\{A : [s]A\}$.*

Theorem 1 (Substitution of equals)

$$(A \overset{s}{\sim} B) \wedge [s]AX \Rightarrow (AX \overset{s}{\sim} BX).$$

While Theorem 1 states the most general form of substitution for $\overset{s}{\sim}$, in practice we use the following corollary frequently.

Corollary 0 (Leibniz)

$$(A \overset{s}{\sim} B) \wedge (AX \overset{s}{\sim} Y) \Rightarrow (BX \overset{s}{\sim} Y)$$

$$(A \overset{X.s}{\sim} B) \wedge (XA \overset{s}{\sim} Y) \Rightarrow (XB \overset{s}{\sim} Y).$$

[2]In the sequel, we will use the abbreviation *function* to refer to a *partial function* and the abbreviation *sequence* to refer to *a finite sequence of partial functions.*

2.2 Lipton's definition

In an early paper [Lip75], Richard Lipton introduced the idea of *left movers* and *right movers* to simplify the task of proving partial correctness properties of parallel programs. Motivated by his ideas, we introduce the operators *left-commutes* ($\mathbf{lco}_l$), *right-commutes* ($\mathbf{rco}_l$) and *commutes* ($\mathbf{co}_l$).

Definition 3 (Lipton) *For any two functions f and g, we have*

$$\begin{array}{lcl}(f\ \mathbf{lco}_l\ g) & \equiv & \langle \forall s : [s]gf : fg \overset{s}{\sim} gf\rangle. \\ (f\ \mathbf{rco}_l\ g) & \equiv & (g\ \mathbf{lco}_l\ f). \\ (f\ \mathbf{co}_l\ g) & \equiv & (f\ \mathbf{lco}_l\ g) \wedge (f\ \mathbf{rco}_l\ g).\end{array}$$

The three operators, $\mathbf{lco}_l$, $\mathbf{rco}_l$ and $\mathbf{co}_l$ can be naturally extended to pairs of sequences as follows.

Definition 4 *For any two sequences A and B,*

$$\begin{array}{lcl}(A\ \mathbf{lco}_l\ B) & \equiv & \langle \forall f, g : f \in A \wedge g \in B : f\ \mathbf{lco}_l\ g\rangle. \\ (A\ \mathbf{rco}_l\ B) & \equiv & (B\ \mathbf{lco}_l\ A). \\ (A\ \mathbf{co}_l\ B) & \equiv & (A\ \mathbf{lco}_l\ B) \wedge (A\ \mathbf{rco}_l\ B).\end{array}$$

Lemma 0 *Let a function f and a sequence M be given such that $(f\ \mathbf{lco}_l\ M)$. Then,*

$$\langle \forall s : [s]Mf : Mf \overset{s}{\sim} fM\rangle.$$

We now present an important theorem: given two sequences A and B such that $(A\ \mathbf{lco}_l\ B)$, if any interleaving of A and B is defined in a state then a serial execution of A followed by B is defined in that same state. Furthermore, both the interleaving and the serial execution yield the same state when applied to the same state.

Theorem 2 *Assume two sequences A and B are given such that $A\ \mathbf{lco}_l\ B$. Let X be any interleaving of A and B. Then,*

$$\langle \forall s : [s]X : X \overset{s}{\sim} AB\rangle.$$

Remark: One can generalize Theorem 2 to obtain the following stronger theorem.

Theorem 3 *Assume two sequences A and B are given such that $A\ \mathbf{co}_l\ B$. Let X and Y be any two interleavings of A and B. Then,*

$$\langle \forall s : [s]X : X \overset{s}{\sim} Y\rangle.$$

That is, given two sequences A and B such that $(A\ \mathbf{co}_l\ B)$, if any interleaving of A and B is defined, then *all* interleavings of A and B are defined and each interleaving yields the *same* state. In particular, if a serial execution of A followed by B is defined than *all* parallel executions (that is, interleavings) are defined and yield the same state. That is, $(A\ \mathbf{co}_l\ B)$ is a sufficient condition to interleave the executions of A and B. A similiar result was the basis of the work in [Lip75]. (End of Remark)

2.3 Misra's definition

In a recent paper [Mis91], Jayadev Misra introduced the idea of *loose-coupling.* An integral part of his work was a commutativity relation between functions of different processes comprising the program. Motivated by his ideas, we introduce the operator *commutes* ($\mathbf{co}_m$).

Definition 5 (Misra) *For any two functions f and g, we have*

$$(f \,\mathbf{co}_m\, g) \;\equiv\; \langle \forall s : [s]f \wedge [s]g : fg \overset{s}{\sim} gf \rangle.$$

The operator, $\mathbf{co}_m$ can be naturally extended to pairs of sequences as follows.

Definition 6 *For any two sequences, A and B,*

$$(A \,\mathbf{co}_m\, B) \;\equiv\; \langle \forall f, g : f \in A \wedge g \in B : f \,\mathbf{co}_m\, g \rangle.$$

Lemma 1 *Let a function f and a sequence M be given such that $(f \,\mathbf{co}_m\, M)$. Then,*

$$\langle \forall s : [s]f \wedge [s]Mf : Mf \overset{s}{\sim} fM \rangle.$$

Theorem 4 *Assume that two sequences A and B are given such that $(A \,\mathbf{co}_m\, B)$. Let X be an interleaving of sequences A and B. Then,*

$$\langle \forall s : [s]X \wedge [s]A : X \overset{s}{\sim} AB \rangle.$$

This result appears as Theorem 3 and Corollary 1 in [Mis91]. Using theorem 4, Misra derives the following important corollary.

Corollary 1 *Given are two sequences Ag and B such that $Ag \,\mathbf{co}_m\, B$. Let X be an interleaving of A and B. Then,*

$$\langle \forall s : [s]Ag \wedge [s]X : [s]Xg \rangle.$$

Remark : By constructing a proof very similiar to that of Theorem 4 one can prove the following stronger theorem.

Theorem 5 *Assume that two sequences A and B are given such that $(A \,\mathbf{co}_m\, B)$. Let X and Y be any two interleavings of A and B. Then,*

$$\langle \forall s : [s]X \wedge [s]Y : X \overset{s}{\sim} Y \rangle.$$

We summarize this important result. Given two sequences A and B such that $(A \,\mathbf{co}_m\, B)$, any two defined interleavings of A and B yield the *same* state. In particular, if a serial execution of A followed by B is defined, then any *defined* interleaving will have the same value as the defined serial execution. Notice that not all interleavings may be defined. This result appears as Theorem 2 in [Mis91]. (End of Remark)

2.4 Incomparability of Lipton and Misra Commutativity

In the previous sections, we introduced two notions of commutativity due to Lipton [Lip75] and Misra [Mis91] and have explored some of the consequences of these definitions. While it might appear that Lipton commutativity ($\mathbf{co}_l$) is stronger than Misra commutativity ($\mathbf{co}_m$), the two notions of commutativity are incomparable.

Theorem 6 *There exist functions f and g such that,*

$$(f \,\mathbf{co}_l\, g) \wedge \neg(f \,\mathbf{co}_m\, g)$$

and

$$\neg(f \,\mathbf{co}_l\, g) \wedge (f \,\mathbf{co}_m\, g).$$

Proof (of 6): The first assertion follows by choosing f and g to be P operations on a semaphore. The second assertion follows by choosing f to be a P operation and g to be a V operation on a a semaphore. (End of Proof)

2.5 A New Definition of Commutativity

Based on Lipton's and Misra's definitions of commutativity, we propose the following definition.

Definition 7 *For any two functions f and g, we have*

$$\begin{array}{lcl}(f\,\mathbf{lco}\,g) & \equiv & (f\,\mathbf{lco}_l\,g)\wedge(f\,\mathbf{co}_m\,g).\\ (f\,\mathbf{rco}\,g) & \equiv & (g\,\mathbf{lco}\,f).\\ (f\,\mathbf{co}\,g) & \equiv & (f\,\mathbf{lco}\,g)\wedge(f\,\mathbf{rco}\,g).\end{array}$$

As before, the three operators, **lco**, **rco** and **co** can be naturally extended to pairs of sequences. However, keeping in mind that our programs are *sets of functions*, we choose to extend them to pairs of sets as follows.

Definition 8 *For two sets F and G,*

$$\begin{array}{lcl}(F\,\mathbf{lco}\,G) & \equiv & \langle\forall f,g : f\in F\wedge g\in G : f\,\mathbf{lco}\,g\rangle.\\ (F\,\mathbf{rco}\,G) & \equiv & (G\,\mathbf{lco}\,F).\\ (F\,\mathbf{co}\,G) & \equiv & (F\,\mathbf{lco}\,G)\wedge(F\,\mathbf{rco}\,G).\end{array}$$

2.6 A Compositional Theorem for Progress

We are now ready to present a theorem that allows the composition of progress properties.

Theorem 7 *Assume that two programs F and G are given such that $(F\,\mathbf{lco}\,G)$. For state predicates P and Q, we have*

$$\frac{P\mapsto Q\,\mathbf{in}\,F \qquad \mathbf{stable}\,Q\,\mathbf{in}\,G}{P\mapsto Q\,\mathbf{in}\,F\ [\!]\ G.}$$

3 A Predicate Transformer Approach

3.1 Preliminaries

We assume familiarity with the usual boolean and arithmetic operators. The operators we use are summarized below, ordered by increasing binding power.

$$\begin{array}{c}\equiv,\not\equiv\\ \Leftarrow,\Rightarrow\\ \rightsquigarrow\\ \wedge,\vee\\ \neg\\ =,\neq,\leq,<,\geq,>\\ +,-\\ \text{“.” (function application)}\end{array}$$

We use X, Y, Z to denote predicates on program states, f, g, h to denote predicate transformers and $\mathbf{R}$,$\mathbf{S}$ and $\rightsquigarrow$ to denote binary relations on predicates.

Universal quantification over all the program variables [DS90] is denoted by the square brackets ([], read *everywhere*). This operator takes a predicate as an argument and returns a boolean value. It enjoys all the properties of universal quantification over a non–empty domain.

3.2 Extremal Solutions of Equations

For a set of equations E in the unknown x, we write $x : E$ to explicate the dependence on the unknown x. Given an ordering $\Rightarrow$ on the solutions of E, y is the *strongest solution* (*or the least fixpoint of* E) if and only if

1. y solves E
2. $\langle \forall z : z \text{ solves } E : y \Rightarrow z \rangle$

The *weakest solution* (*or the greatest fixpoint*) of E can be defined in a similiar manner.

We extend the notation for quantification, introduced earlier, to cover extremal solutions as follows. We allow expressions of the form

$$\langle \underline{Q} x :: E \rangle$$

where $\underline{Q}$ can be one of μ and ν to denote the strongest and weakest solution of $x : E$ respectively.

Finally, we will use the following form of the Theorem of Knaster–Tarski [DS90], to prove properties of extremal solutions.

Theorem 8 (Knaster–Tarski) *For monotonic f, the equation*

$$Y : [f.Y \equiv Y]$$

has a strongest and a weakest solution. Furthermore, it has the same strongest solution as

$$Y : [f.Y \Rightarrow Y]$$

and the same weakest solution as

$$Y : [f.Y \Leftarrow Y].$$

Theorem 9 *For monotonic f, let g.X be the strongest solution and h.X the weakest solution of*

$$Y : [f.X.Y \equiv Y].$$

Then both g and h are monotonic.

3.3 The Closure of a Program

Recall the model of computation introduced in Section 1. A program is essentially a finite and non–empty set of statements. We assume that the statements in our programs are *terminating*. In terms of the *predicate transformer* semantics [DS90], this means that for a statement s and a state predicate X,

$$[\mathbf{wlp}\,.s.X \equiv \mathbf{wp}\,.s.X] \tag{S0}$$

Given this, we are allowed to assume that the predicate transformers **wp** and **sp** are converses, that is,

$$[X \Rightarrow \mathbf{wp}\,.s.Y] \equiv [\mathbf{sp}\,.s.X \Rightarrow Y] \tag{S1}$$

We also assume that the $\mathbf{wp}\,.s$ predicate transformer is monotonic, that is

$$[X \Rightarrow Y] \;\Rightarrow\; [\mathbf{wp}\,.s.X \Rightarrow \mathbf{wp}\,.s.Y] \tag{S2}$$

An execution of a program begins in *any* state and proceeds by repeatedly selecting a statement from this set and executing it with the *fairness* constraint that in every infinite execution each statement is selected and executed infinitely often.

Given this model of computation, it makes sense to ask the following question: given that program execution is begun in a state satisfying state predicate P, what is the predicate that characterizes the *smallest* set of states that the program execution can lead to ? For a given program G, we capture this notion by means of a predicate transformer g that takes the predicate P as an argument. We call $g.P$ the *closure* of program G with respect to predicate P.

Formally, given a program G and a state predicate P, consider the set L of equations L0–L1 in the unknown predicate Z:

(L0) $$[P \Rightarrow Z]$$
(L1) $$(Z \textbf{ unless } false \textbf{ in } G)$$

Condition L0 states that Z is weaker than P. Condition L1 states that Z is **stable** in program G.

Definition 9 (closure) *For a given program G, the strongest solution of equations L will be called the* closure *of G. It will be denoted by $g.P$.*

The following properties of the closure follow from its definition.

(G0) $$[P \Rightarrow g.P]$$
(G1) $$(g.P \textbf{ unless } false \textbf{ in } G)$$
(G2) $$[P \Rightarrow Z] \wedge (Z \textbf{ unless } false \textbf{ in } G) \Rightarrow [g.P \Rightarrow Z]$$

The closure of a program can also be written as the least fixpoint of a predicate calculus expression as the following theorem shows.

Theorem 10 *For a program G and a state predicate P,*

$$[g.P \equiv \langle \mu Z :: P \vee \langle \exists s : s \in G : \mathbf{sp}.s.Z \rangle \rangle]$$

This alternate characterization of the closure of a program, as the least fixpoint of a predicate calculus expression, gives us the following properties.

(G3) $$[g.P \equiv P \vee \langle \exists s : s \in G : \mathbf{sp}.s.(g.P) \rangle]$$
(G4) $$[Z \Leftarrow P \vee \langle \exists s : s \in G : \mathbf{sp}.s.Z \rangle] \Rightarrow [g.P \Rightarrow Z]$$
(G5) $$[g.(g.P) \equiv g.P]$$
(G6) $$[g.\langle \exists X : X \in W : X \rangle \equiv \langle \exists X : X \in W : g.X \rangle]$$

We will make use of the following lemma in our theorems.

Lemma 2 *For a program G and a state predicate Z,*

$$\textbf{stable } Z \textbf{ in } G \equiv [g.Z \equiv Z]$$

3.4 A Predicate Transformer Based Definition of Commutativity

The concept of the closure of a program allows us to formulate a new definition for the commutativity of two programs. Accordingly, we define *left-commutes* ($\triangleleft$), *right-commutes* ($\triangleright$) and *commutes* ($\bowtie$) as follows.

Definition 10 (commutes, "$\bowtie$") *Given are two programs F and G. Assume that the variable s ranges over the statements of programs F. Then,*

$$\begin{array}{lcl} (F \triangleleft G) & \equiv & \langle \forall Z : \textbf{stable } Z \textbf{ in } G : \langle \forall s : s \in F : \textbf{stable wp}.s.Z \textbf{ in } G \rangle \rangle \\ (F \triangleright G) & \equiv & (G \triangleleft F) \\ (F \bowtie G) & \equiv & (F \triangleleft G) \wedge (F \triangleright G) \end{array}$$

3.5 Preservation of Properties

To reiterate, our goal is to derive a compositional methodology for the development of parallel programs. This means that if we have proved a property of a program F, we would like the same (or similiar) property to hold of the composite program $F \parallel G$ obtained by composing F with some other program G.

Motivating Example : Consider the example of a *producer* and a *consumer* who share an infinite buffer. Let N denote the number of items in the buffer. It is clear that the property $(N = 1) \mapsto (N = 0)$ holds for the *consumer*. What corresponding progress property can we expect when the *producer* and *consumer* are executed in parallel ? Suppose the execution of the composite program is begun in a state satisfying predicate $(N = 1)$. It is possible that the *producer* executes faster than the *consumer* filling the buffer with items. Now, from the definition of closure, the worst that the *producer* can do is to take the composite program to a state satisfying the closure of $(N = 1)$, that is, $(N \geq 1)$. Eventually, by our requirement of fairness, the *consumer* will be scheduled for execution. While this will decrease N by the number of items consumed, it is easy to see that the composite program may never reach a state $(N = 0)$. However, a little thinking will show that the program will reach a state satisfying the closure of $(N = 0)$, that is $(N \geq 0)$. And this is exactly what the paradigm of loose–coupling intends to capture !

Formally, given programs F and G and a property $P \mapsto Q$ of F, we would like to derive conditions under which property $g.P \mapsto g.Q$ holds of $F \parallel G$, where g is the closure of program G. This way of defining loose–coupling was first reported by Ernie Cohen [Coh91]. (End of Motivating Example)

Lemma 3 (Safety, "unless") *Assume that programs F and G are given such that $F \triangleleft G$. For state predicates P and Q, let $g.P$ and $g.Q$ be the closures of G with respect to P and Q respectively. Then,*

$$P \textbf{ unless } Q \textbf{ in } F \;\Rightarrow\; g.P \textbf{ unless } g.Q \textbf{ in } F \parallel G.$$

Lemma 4 (Basic Progress, "ensures") *Assume that programs F and G are given such that $F \triangleleft G$. For state predicates P and Q, let $g.P$ and $g.Q$ be the closures of G with respect to P and Q respectively. Then,*

$$P \textbf{ ensures } Q \textbf{ in } F \;\Rightarrow\; g.P \textbf{ ensures } g.Q \textbf{ in } F \parallel G.$$

Lemma 5 (Progress, "$\mapsto$") *Assume that programs F and G are given such that $F \triangleleft G$. For state predicates P and Q, let $g.P$ and $g.Q$ be the closures of G with respect to P and Q respectively. Then,*

$$P \mapsto Q \textbf{ in } F \;\Rightarrow\; g.P \mapsto g.Q \textbf{ in } F \parallel G.$$

Theorem 11 *Given are programs F and G such that $F \triangleleft G$. For state predicates P and Q, we have*

$$\frac{P \mapsto Q \textbf{ in } F \qquad \textbf{stable } Q \textbf{ in } G}{P \mapsto Q \textbf{ in } F \parallel G.}$$

4 Conclusions

We have introduced two new definitions of commutativity, namely **co** and $\bowtie$. As would be expected the two notions are related by the following theorem.

Theorem 12 *For any two programs F and G,*

$$(F \textbf{ lco } G) \;\Rightarrow\; (F \triangleleft G).$$

5 Acknowledgements

I am grateful to Professor Jayadev Misra for many stimulating discussions. I would also like to thank Ernie Cohen for sharing many valuable insights with me. Thanks are also due to the Austin Tuesday Afternoon Club and the UNITY group at the University of Texas at Austin for their comments.

References

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Coh91] Ernie Cohen. Ph.d. dissertation (in preparation). Department of Computer Sciences, The University of Texas at Austin, Austin, Texas, 1991.

[CT90] K. Mani Chandy and Stephen Taylor. A primer for program composition notation. Technical Report, Department of Computer Sciences, California Institute of Technology, Pasadena, CA, 1990.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[Doe77] Thomas W. Doeppner Jr. Parallel program correctness through refinement. In *Proceedings of the 4th Annual ACM Symposium on the Principles of Programming Languages*, January 1977.

[Lam87] Leslie Lamport. **win** and **sin**: Predicate transformers for concurrency. Technical Report 17, DEC Systems Research Center, May 1987.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[LS89] Leslie Lamport and Fred B. Schnieder. Pretending atomicity. Technical Report 44, DEC Systems Research Center, May 1989.

[Mis91] Jayadev Misra. Loosely coupled processes. In *Proceedings of PARLE '91, Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, LNCS 506*, pages 1–26, June 1991.

6 Appendix: Proofs of Theorems

Section 2 :

Theorem 1 (Substitution of equals)

$$(A \overset{s}{\sim} B) \wedge [s]AX \;\Rightarrow\; (AX \overset{s}{\sim} BX).$$

Proof (of 1):

$$\begin{array}{ll} & (AX \overset{s}{\sim} BX) \\ = & \{\text{definition 2}\} \\ & [s]AX \wedge [s]BX \wedge (AX.s = BX.s) \\ = & \{\text{definition 1 for } [s]AX \text{ and } [s]BX\ \} \\ & [s]AX \wedge [s]A \wedge [s]B \wedge [B.s]X \wedge (AX.s = BX.s) \\ \Leftarrow & \{\text{Leibniz for } = \text{ twice}\} \\ & [s]AX \wedge [s]A \wedge [A.s]X \wedge [s]B \wedge (A.s = B.s) \\ \Leftarrow & \{\text{definition 1 for } [s]AX\} \\ & [s]AX \wedge [s]A \wedge [s]B \wedge (A.s = B.s) \\ = & \{\text{definition 2}\} \\ & [s]AX \wedge (A \overset{s}{\sim} B) \end{array}$$

(End of Proof)

Lemma 0 *Let a function f and a sequence M be given such that $(f \,\mathbf{lco}_l\, M)$. Then,*

$$\langle \forall s : [s]Mf : Mf \overset{s}{\sim} fM \rangle.$$

Proof (of 0): The proof is by induction on M.

Base Case : M is the empty string $<>$.

$$\begin{array}{ll} & (Mf \overset{s}{\sim} fM) \\ = & \{M = <>\} \\ & (f \overset{s}{\sim} f) \\ = & \{\text{reflexivity of } \overset{s}{\sim}\} \\ & [s]f \\ = & \{M = <>\} \\ & [s]Mf \end{array}$$

Induction Step : Assume that M is of the form $M'g$.

$$
\begin{array}{ll}
 & (Mf \overset{s}{\sim} fM) \\
= & \{M = M'g\} \\
 & (M'gf \overset{s}{\sim} fM'g) \\
\Leftarrow & \{\text{Leibniz}\} \\
 & (M'gf \overset{s}{\sim} M'fg) \wedge (fM' \overset{s}{\sim} M'f) \\
\Leftarrow & \{\text{Leibniz}\} \\
 & (M'fg \overset{s}{\sim} M'fg) \wedge (fg \overset{M'.s}{\sim} gf) \wedge (fM' \overset{s}{\sim} M'f) \\
= & \{\text{definition 2}\} \\
 & [s]M'fg \wedge (fg \overset{M'.s}{\sim} gf) \wedge (M'f \overset{s}{\sim} fM') \\
\Leftarrow & \{(f \,\mathbf{lco}_l\, M') \text{ and induction hypothesis}\} \\
 & [s]M'fg \wedge [s]M'f \wedge (fg \overset{M'.s}{\sim} gf) \\
= & \{\text{definition 1}\} \\
 & [s]M'fg \wedge (fg \overset{M'.s}{\sim} gf) \\
\Leftarrow & \{\text{Leibniz}\} \\
 & [s]M'gf \wedge (fg \overset{M'.s}{\sim} gf) \\
\Leftarrow & \{f \,\mathbf{lco}_l\, g\} \\
 & [s]M'gf \wedge [M'.s]gf \\
= & \{\text{definition 1}\} \\
 & [s]M'gf \\
= & \{\text{definition of } M\} \\
 & [s]Mf
\end{array}
$$

(End of Proof)

Theorem 2 *Assume two sequences A and B are given such that $A \,\mathbf{lco}_l\, B$. Let X be any interleaving of A and B. Then,*

$$\langle \forall s : [s]X : X \overset{s}{\sim} AB \rangle.$$

Proof (of 2): The proof is by induction on the length of A.

Base Case : Let A be the empty sequence $<>$. This means that X is the sequence B.

$$
\begin{array}{ll}
 & (X \overset{s}{\sim} AB) \\
= & \{X = B \text{ and } A = <>\} \\
 & (B \overset{s}{\sim} B) \\
= & \{\text{reflexivity of } \overset{s}{\sim} \text{ and } [s]B\} \\
 & true
\end{array}
$$

Induction Step : Let A be the sequence fA'. From the definition of an interleaving this implies that f occurs in X as well. That is, X is of the form MfN, for some sequences M and N. Since f is from A, the sequence M is a prefix of B. From our assumption of $(A \,\mathbf{lco}_l\, B)$, this means that $(f \,\mathbf{lco}_l\, M)$ which by Lemma 0 yields $\langle \forall s : [s]Mf : Mf \overset{s}{\sim} fM \rangle$. Using these facts, we observe

$$
\begin{array}{ll}
 & (X \overset{s}{\sim} AB) \\
= & \{\text{definition of } X \text{ and } A\} \\
 & (MfN \overset{s}{\sim} fA'B) \\
\Leftarrow & \{\text{Leibniz}\} \\
 & (fMN \overset{s}{\sim} fA'B), (Mf \overset{s}{\sim} fM) \\
= & \{\text{definition 2}\} \\
 & [s]f, (MN \overset{f.s}{\sim} A'B), (Mf \overset{s}{\sim} fM) \\
= & \{\text{induction hypothesis}\} \\
 & [s]f, [f.s]MN, [f.s]A', (Mf \overset{s}{\sim} fM) \\
= & \{\text{definition 1 twice}\} \\
 & [s]fMN, [s]fA', (Mf \overset{s}{\sim} fM) \\
\Leftarrow & \{\text{Theorem 1}\} \\
 & [s]MfN, [s]fA', (Mf \overset{s}{\sim} fM) \\
\Leftarrow & \{\text{condition 1}\} \\
 & [s]MfN, [s]fA', [s]f, [s]Mf \\
= & \{\text{definition 1 twice}\} \\
 & [s]MfN, [s]fA' \\
= & \{\text{definition of } X \text{ and } A\} \\
 & [s]X \wedge [s]A
\end{array}
$$

(End of Proof)

Theorem 6 *There exist functions f and g such that,*

$$(f \,\mathbf{co}_l\, g) \wedge \neg(f \,\mathbf{co}_m\, g)$$

and

$$\neg(f \,\mathbf{co}_l\, g) \wedge (f \,\mathbf{co}_m\, g).$$

Proof (of 6): We will prove the theorem by demonstrating functions f and g which satisfy the two conditions. Our examples will be drawn from the familiar operations P and V on the semaphore x.

0. We will demonstrate $(f \,\mathbf{co}_l\, g) \wedge \neg(f \,\mathbf{co}_m\, g)$. Let both f and g be P operations. The following derivation proves $(f \,\mathbf{co}_l\, g)$.

$$
\begin{array}{ll}
 & (f \,\mathbf{co}_l\, g) \\
= & \{\text{definition of } \mathbf{co}_l\} \\
 & (f \,\mathbf{lco}_l\, g) \wedge (f \,\mathbf{rco}_l\, g) \\
= & \{\text{instantiating } f \text{ and } g\} \\
 & (P \,\mathbf{lco}_l\, P) \wedge (P \,\mathbf{rco}_l\, P) \\
= & \{\text{symmetry and definition of } \mathbf{lco}_l\} \\
 & \langle \forall s : [s]PP : PP \overset{s}{\sim} PP \rangle \\
= & \{\text{reflexivity of } \overset{s}{\sim}\} \\
 & true
\end{array}
$$

The following derivation demonstrates $\neg(f \,\mathbf{co}_m\, g)$.

$(f\ \mathbf{co}_m\ g)$
$=$ {instantiating f and g}
$(P\ \mathbf{co}_m\ P)$
$=$ {definition of $\mathbf{co}_m$}
$\langle \forall s : [s]P \wedge [s]P : PP \overset{s}{\sim} PP \rangle$
$=$ {semaphore $x = 1$ implies P is defined and PP isn't}
$false$

1. We will demonstrate $\neg(f\ \mathbf{co}_l\ g) \wedge (f\ \mathbf{co}_m\ g)$. Let f be a P operation and g be V operations. The following derivation proves $\neg(f\ \mathbf{co}_l\ g)$.

$(f\ \mathbf{co}_l\ g)$
$=$ {definition of $\mathbf{co}_l$}
$(f\ \mathbf{lco}_l\ g) \wedge (f\ \mathbf{rco}_l\ g)$
$=$ {instantiating f and g}
$(P\ \mathbf{lco}_l\ V) \wedge (P\ \mathbf{rco}_l\ V)$
$=$ {definitions of $\mathbf{lco}_l$ and $\mathbf{rco}_l$}
$\langle \forall s : [s]VP : PV \overset{s}{\sim} VP \rangle \wedge$
$\langle \forall s : [s]PV : PV \overset{s}{\sim} VP \rangle$
$=$ {semaphore $x = 0$ implies VP is defined but PV isn't}
$false$

The following derivation demonstrates $(f\ \mathbf{co}_m\ g)$.

$(f\ \mathbf{co}_m\ g)$
$=$ {instantiating f and g}
$(P\ \mathbf{co}_m\ V)$
$=$ {definition of $\mathbf{co}_m$}
$\langle \forall s : [s]P \wedge [s]V : PV \overset{s}{\sim} VP \rangle$
$=$ {semaphore definition}
$true$

(End of Proof)

Theorem 7 *Assume that two programs F and G are given such that $(F\ \mathbf{lco}\ G)$. For state predicates P and Q, we have*

$$\frac{P \mapsto Q\ \mathbf{in}\ F \qquad \mathbf{stable}\ Q\ \mathbf{in}\ G}{P \mapsto Q\ \mathbf{in}\ F\ [\!]\ G.}$$

Proof (of 7): Our argument will be an operational one. By their very nature operational arguments tend to be involved and ours is no exception. To make the proof readable and understandable, we develop a running example along with the proof.

1. Consider an execution, σ, of the UNITY program $F\ [\!]\ G$ beginning in a state, s. We know four facts about σ. Firstly, σ is infinite. Secondly, the transitions of the execution σ can be labelled with the statements of $F\ [\!]\ G$. Thirdly, it is *fair* with respect to the statements of the composite program $F\ [\!]\ G$. Fourthly, recall from section 1, that some of these transitions may have caused no state change as the statement selected to execute the transition may have been undefined.

Example : Let σ be of the form

$$s \xrightarrow{G0} s1 \xrightarrow{F0} s2 \xrightarrow{F1} s3 \xrightarrow{G1} s4 \xrightarrow{F2} s5 \xrightarrow{G2} s6 \xrightarrow{F3} s7 \ldots$$

where $F0$, $F1$, $F2$, $F3$, ... are transitions corresponding to statements of program F and $G0$, $G1$, $G2$, ... correspond to statements of program G.

2. Assume that predicate P is true in s. Our proof obligation requires us to show that there exists a state, t, in σ such that predicate Q is true in t.

3. From σ, project the sequence of transitions that belong to program F. Call this sequence σ_F. We know three facts about σ_F. Firstly, σ_F is NOT an execution sequence. It is a sequence of statements. Secondly, it is infinite. Thirdly, σ_F is *fair* with respect to the statements of F, in the sense that every statement of F occurs infinitely often in it.

   Example (contd.) : With σ as given above, σ_F is the sequence $F0$, $F1$, $F2$, $F3$, ...

4. Suppose the statements of the sequence σ_F are applied, in order, to the state s. Now predicate P is true in s. Since σ_F is *fair* with respect to program F, and since our first hypothesis is that $(P \mapsto Q \textbf{ in } F)$, we eventually reach a state, u, satisfying predicate Q after a finite prefix, α, of σ_F has been applied to s. We know two facts about α. Firstly, α is finite. Secondly, it is possible that α is undefined in state s (that is, $\neg[s]\alpha$ holds) as some of the statements of α were undefined and were treated as **skip** statements.

   Example (contd.) : The sequence σ_F was given by $F0$, $F1$, $F2$, $F3$, ... Applying this sequence to the state s yields the following execution sequence.

$$s \xrightarrow{F0} t1 \xrightarrow{F1} t2 \xrightarrow{F2} u \xrightarrow{F3} t4 \ldots$$

   Thus α is the finite prefix $F0$, $F1$, $F2$ of σ_F.

5. Since α is a prefix of σ_F and since σ_F is obtained, by projection, from σ, there exists a prefix τ of σ such that, α is obtained by projecting the statements of F from τ. We know two facts about τ. Firstly, τ is finite. Secondly, τ may be undefined in state s.

   Example (contd.) : Given that α is the finite sequence $F0$, $F1$, $F2$ the sequence τ will be $G0$, $F0$, $F1$, $G1$, $F2$.

6. Clearly, τ is an interleaving of two sequences : sequence α consisting of transitions from F and sequence β consisting of transitions from G. The question is: how can one transform τ so that it becomes defined in state s ? The answer is: by dropping those transitions of τ that were undefined and equivalent to **skip** actions. Note that the dropped transitions may belong to either α or β. We consider these two cases separately.

   Example (contd.) : The sequence τ is an interleaving of sequence α ($F0$, $F1$, $F2$) and β ($G0$, $G1$). Consider the execution sequence σ again. Then the execution sequence corresponding to τ is

$$s \xrightarrow{G0} s1 \xrightarrow{F0} s2 \xrightarrow{F1} s3 \xrightarrow{G1} s4 \xrightarrow{F2} s5$$

   Now it may be the case that the transition $F1$ is undefined in state $s2$ and $G1$ is undefined in state $s3$. That is, the states $s2$, $s3$ and $s4$ are exactly the same states. So to ensure that τ is defined in state s, we have to drop transitions $F1$ and $G1$.

7. Dropping actions of β does not pose a problem. Dropping transitions of α is problematic for the following reason. Suppose a transition f (belonging to α) was undefined in τ and was dropped. The danger that arises is that f may have been *defined* when α is applied to state s in step 4 above. In fact, it may have been instrumental in reaching state u which satisfies Q. We can drop f safely if we have a theorem that says that f is undefined in α as well.

   Example (contd.) : Dropping $G1$ from τ reduces the sequence β from $G0$, $G1$ to $G0$. Dropping $F1$ from τ may be problematic if $F1$ is defined in the execution sequence corresponding to α in step 4 above. Going from state $t1$ to $t2$ may be instrumental in reaching state u in which Q holds. We need a theorem that states that if $F1$ is undefined in τ then it is undefined in α as well.

8. Suppose, X is a prefix of τ that is defined in state s. Note that X is an interleaving of sequences A and B, where A is a prefix of α and B is a prefix β. Since $F\,\mathbf{lco}\,G$, we have $A\,\mathbf{lco}\,B$. From the definition of **lco**, this means that $A\,\mathbf{lco}_l\,B$. From $A\,\mathbf{lco}_l\,B$, $[s]X$ and Theorem 2 we get $[s]AB$ and infer $[s]A$.

   Now let Xf be an undefined prefix of τ. We can drop f from τ if we have the following theorem.

$$[s]X \wedge \neg[s]Xf \wedge [s]A \;\Rightarrow\; \neg[s]Af.$$

   An equivalent and simpler statement of the same theorem is,

$$[s]X \wedge [s]Af \;\Rightarrow\; [s]Xf.$$

   But this is the precise statement of Corollary 1 ! And in our analysis, we may assume this result provided $A\,\mathbf{co}_m\,B$ and this is exactly what $A\,\mathbf{lco}\,B$ gives us. A simple induction allows us to conclude that any transition of program F that is undefined in τ is undefined in α as well.

9. Let τ' be the sequence that results after dropping the undefined transitions of τ. We know three facts about τ'. Firstly, τ' is defined in state s. Secondly, τ' is an interleaving of sequences α' (obtained by dropping the undefined F–transitions of τ from α) and β' (obtained by dropping the undefined G–transitions of τ from β). Thirdly, since the undefined transitions that were dropped from τ were **skip** transitions, both τ and τ' yield the same state t when applied to state s. Our goal is to show that predicate Q is true in state t.

   From the first and second facts about τ', $F\,\mathbf{lco}\,G$ and Theorem 2, we can conclude that the sequence $\alpha'\beta'$ is defined in state s and results in the same state t when applied to s. Further, since only undefined F–transitions were dropped from α to obtain α', applying α' to s yields the same state u as α did (in step 4). Recall that Q is true in u. By our second hypothesis, transitions of G do not falsify Q. Thus Q is maintained by β'. Hence Q is true in state t. That is, Q is true in the state that results when a finite prefix τ of σ is applied to state s. And this concludes our argument.

   Example (contd.) : After dropping $F1$ and $G1$ from τ, the sequence τ' that results will be $G0$, $F0$, $F2$. The execution sequence corresponding to τ' is

$$s \xrightarrow{G0} s1 \xrightarrow{F0} s4 \xrightarrow{F2} s5$$

   Note that same state $s5$ results when either τ or τ' is applied to s. Our goal is to show that Q holds in $s5$.

   From Theorem 2, we have $\tau' \overset{s}{\sim} F0, F2, G0$. That is, both sequences yield the same state ($s5$) when applied to s. But from step 4, applying $F0$, $F2$ to state s leads to state u in which Q holds. Further, from our second hypothesis, Q is **stable** in G. Thus Q holds after the application of $G0$. That is Q holds in state $s5$. And this concludes our example.

(End of Proof)

Section 3:

Theorem 10 *For a program G and a state predicate P,*

$$[g.P \;\equiv\; \langle \mu Z :: P \vee \langle \exists s : s \in G : \mathbf{sp}.s.Z \rangle \rangle]$$

Proof (of 10):

$$\begin{array}{ll} & g.P \\ = & \{\text{definition of } g\} \\ & \langle \mu Z :: [P \Rightarrow Z] \wedge (Z \textbf{ unless } false \textbf{ in } G) \rangle \\ = & \{\text{definition of } \textbf{unless}\} \\ & \langle \mu Z :: [P \Rightarrow Z] \wedge \langle \forall s : s \in G : [Z \Rightarrow \mathbf{wp}.s.Z] \rangle \rangle \\ = & \{\text{property S1}\} \\ & \langle \mu Z :: [P \Rightarrow Z] \wedge \langle \forall s : s \in G : [\mathbf{sp}.s.Z \Rightarrow Z] \rangle \rangle \\ = & \{\text{interchange quantification}\} \\ & \langle \mu Z :: [P \Rightarrow Z] \wedge [\langle \forall s : s \in G : \mathbf{sp}.s.Z \Rightarrow Z \rangle] \rangle \\ = & \{\text{predicate calculus}\} \\ & \langle \mu Z :: [P \Rightarrow Z] \wedge [\langle \exists s : s \in G : \mathbf{sp}.s.Z \rangle \Rightarrow Z] \rangle \\ = & \{\text{predicate calculus}\} \\ & \langle \mu Z :: [P \vee \langle \exists s : s \in G : \mathbf{sp}.s.Z \rangle \Rightarrow Z] \rangle \\ = & \{\text{Theorem 9}\} \\ & \langle \mu Z :: [P \vee \langle \exists s : s \in G : \mathbf{sp}.s.Z \rangle \equiv Z] \rangle \end{array}$$

(End of Proof)

Lemma 2 *For a program G and a state predicate Z,*

$$\textbf{stable } Z \textbf{ in } G \;\equiv\; [g.Z \equiv Z]$$

Proof (of 2): The proof will be by mutual implication.

($\Rightarrow$)

$$\begin{array}{ll} & \textbf{stable } Z \textbf{ in } G \\ = & \{\text{predicate calculus}\} \\ & [Z \Rightarrow Z] \wedge \textbf{stable } Z \textbf{ in } G \\ \Rightarrow & \{\text{property G2}\} \\ & [g.Z \Rightarrow Z] \\ = & \{\text{property G0}\} \\ & [g.Z \equiv Z] \end{array}$$

($\Leftarrow$)

$$
\begin{array}{ll}
 & \mathit{true} \\
= & \{\text{property G1}\} \\
 & \textbf{stable}\ g.Z\ \textbf{in}\ G \\
= & \{\text{hypothesis } [g.Z \equiv Z]\} \\
 & \textbf{stable}\ Z\ \textbf{in}\ G
\end{array}
$$

(End of Proof)

Lemma 3 (Safety, "unless") *Assume that programs F and G are given such that $F \triangleleft G$. For state predicates P and Q, let $g.P$ and $g.Q$ be the closures of G with respect to P and Q respectively. Then,*

$$P\ \textbf{unless}\ Q\ \textbf{in}\ F \ \Rightarrow\ g.P\ \textbf{unless}\ g.Q\ \textbf{in}\ F\ [\!]\ G.$$

Proof (of 3):

$$
\begin{array}{ll}
 & g.P\ \textbf{unless}\ g.Q\ \textbf{in}\ F\ [\!]\ G \\
= & \{\text{union theorem for safety}\} \\
 & g.P\ \textbf{unless}\ g.Q\ \textbf{in}\ F\ \wedge\ g.P\ \textbf{unless}\ g.Q\ \textbf{in}\ G \\
\Leftarrow & \{\text{consequence weakening for } \textbf{unless}\} \\
 & g.P\ \textbf{unless}\ g.Q\ \textbf{in}\ F\ \wedge\ g.P\ \textbf{unless}\ \mathit{false}\ \textbf{in}\ G \\
= & \{\text{property G1}\} \\
 & g.P\ \textbf{unless}\ g.Q\ \textbf{in}\ F \\
= & \{\text{definition of } \textbf{unless}\} \\
 & \langle \forall s : s \in F : [g.P \wedge \neg g.Q \Rightarrow \textbf{wp}\,.s.(g.P \vee g.Q)]\rangle \\
= & \{\text{predicate calculus; omitting range of } F\} \\
 & \langle \forall s :: [g.P \Rightarrow g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q)]\rangle \\
\Leftarrow & \{\text{property G2}\} \\
 & \langle \forall s :: [P \Rightarrow g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q)] \wedge \\
 & (g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))\ \textbf{unless}\ \mathit{false}\ \textbf{in}\ G\rangle \\
\Leftarrow & \{\text{property G0}\} \\
 & \langle \forall s :: [P \Rightarrow Q \vee \textbf{wp}\,.s.(g.P \vee g.Q)] \wedge \\
 & (g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))\ \textbf{unless}\ \mathit{false}\ \textbf{in}\ G\rangle \\
\Leftarrow & \{\text{property G0 and property S2}\} \\
 & \langle \forall s :: [P \Rightarrow Q \vee \textbf{wp}\,.s.(P \vee Q)] \wedge \\
 & (g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))\ \textbf{unless}\ \mathit{false}\ \textbf{in}\ G\rangle \\
= & \{\text{predicate calculus}\} \\
 & \langle \forall s :: [P \wedge \neg Q \Rightarrow \textbf{wp}\,.s.(P \vee Q)] \wedge \\
 & (g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))\ \textbf{unless}\ \mathit{false}\ \textbf{in}\ G\rangle \\
= & \{\text{assumption and definition of } P\ \textbf{unless}\ Q\} \\
 & \langle \forall s :: (g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))\ \textbf{unless}\ \mathit{false}\ \textbf{in}\ G\rangle \\
= & \{\text{definition of } \textbf{unless}\} \\
 & \langle \forall s :: \langle \forall t : t \in G : [(g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q)) \ \Rightarrow\ \textbf{wp}\,.t.(g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))]\rangle\rangle \\
= & \{\text{unnesting, omitting range, predicate calculus}\} \\
 & \langle \forall s,t :: [g.Q \ \Rightarrow\ \textbf{wp}\,.t.(g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))]\rangle \wedge \\
 & \langle \forall s,t :: [\textbf{wp}\,.s.(g.P \vee g.Q) \ \Rightarrow\ \textbf{wp}\,.t.(g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))]\rangle \\
= & \{\text{property G1}\} \\
 & \langle \forall s,t :: [\textbf{wp}\,.s.(g.P \vee g.Q) \ \Rightarrow\ \textbf{wp}\,.t.(g.Q \vee \textbf{wp}\,.s.(g.P \vee g.Q))]\rangle
\end{array}
$$

$\Leftarrow$ {predicate calculus}
$\langle \forall s, t :: [\mathbf{wp}.s.(g.P \vee g.Q) \;\Rightarrow\; \mathbf{wp}.t.(\mathbf{wp}.s.(g.P \vee g.Q))]\rangle$
$=$ {property G6}
$\langle \forall s, t :: [\mathbf{wp}.s.(g.(P \vee Q)) \;\Rightarrow\; \mathbf{wp}.t.(\mathbf{wp}.s.(g.(P \vee Q)))]\rangle$
$=$ {definition of **stable**}
$\langle \forall s :: \mathbf{stable}\ \mathbf{wp}.s.(g.(P \vee Q))\ \mathbf{in}\ G\rangle$
$\Leftarrow$ {$F \triangleleft G$}
$\mathbf{stable}\ g.(P \vee Q)\ \mathbf{in}\ G$
$=$ {property G1}
$true$

(End of Proof)

Lemma 4 (Basic Progress, "ensures") *Assume that programs F and G are given such that $F \triangleleft G$. For state predicates P and Q, let $g.P$ and $g.Q$ be the closures of G with respect to P and Q respectively. Then,*

$$P\ \mathbf{ensures}\ Q\ \mathbf{in}\ F \;\Rightarrow\; g.P\ \mathbf{ensures}\ g.Q\ \mathbf{in}\ F \;[\!]\; G.$$

Proof (of 4):

$g.P\ \mathbf{ensures}\ g.Q\ \mathbf{in}\ F \;[\!]\; G$
$\Leftarrow$ {union theorem}
$(g.P\ \mathbf{ensures}\ g.Q\ \mathbf{in}\ F) \wedge (g.P\ \mathbf{unless}\ g.Q\ \mathbf{in}\ G)$
$\Leftarrow$ {consequence weakening}
$(g.P\ \mathbf{ensures}\ g.Q\ \mathbf{in}\ F) \wedge (g.P\ \mathbf{unless}\ false\ \mathbf{in}\ G)$
$=$ {property G1}
$g.P\ \mathbf{ensures}\ g.Q\ \mathbf{in}\ F$
$=$ {definition of **ensures**}
$(g.P\ \mathbf{unless}\ g.Q\ \mathbf{in}\ F) \wedge \langle \exists s : s \in F : [g.P \wedge \neg g.Q \Rightarrow \mathbf{wp}.s.(g.Q)]\rangle$
$\Leftarrow$ {Lemma 3, P **ensures** Q **in** F, $F \triangleleft G$}
$\langle \exists s : s \in F : [g.P \wedge \neg g.Q \Rightarrow \mathbf{wp}.s.(g.Q)]\rangle$
$=$ {omitting range, predicate calculus}
$\langle \exists s :: [g.P \Rightarrow g.Q \vee \mathbf{wp}.s.(g.Q)]\rangle$
$\Leftarrow$ {property G2}
$\langle \exists s :: [P \Rightarrow g.Q \vee \mathbf{wp}.s.(g.Q)] \wedge$
$(g.Q \vee \mathbf{wp}.s.(g.Q))\ \mathbf{unless}\ false\ \mathbf{in}\ G\rangle$
$\Leftarrow$ {property G0}
$\langle \exists s :: [P \Rightarrow Q \vee \mathbf{wp}.s.(g.Q)] \wedge$
$(g.Q \vee \mathbf{wp}.s.(g.Q))\ \mathbf{unless}\ false\ \mathbf{in}\ G\rangle$
$\Leftarrow$ {property G0 and property S2}
$\langle \exists s :: [P \Rightarrow Q \vee \mathbf{wp}.s.Q] \wedge$
$(g.Q \vee \mathbf{wp}.s.(g.Q))\ \mathbf{unless}\ false\ \mathbf{in}\ G\rangle$
$=$ {predicate calculus}
$\langle \exists s :: [P \wedge \neg Q \Rightarrow \mathbf{wp}.s.Q] \wedge$
$(g.Q \vee \mathbf{wp}.s.(g.Q))\ \mathbf{unless}\ false\ \mathbf{in}\ G\rangle$
$\Leftarrow$ {predicate calculus}
$\langle \exists s :: [P \wedge \neg Q \Rightarrow \mathbf{wp}.s.Q] \wedge$
$\langle \forall u : u \in F : (g.Q \vee \mathbf{wp}.u.(g.Q))\ \mathbf{unless}\ false\ \mathbf{in}\ G\rangle\rangle$
$=$ {predicate calculus}
$\langle \exists s :: [P \wedge \neg Q \Rightarrow \mathbf{wp}.s.Q]\rangle \wedge$
$\langle \forall u : u \in F : (g.Q \vee \mathbf{wp}.u.(g.Q))\ \mathbf{unless}\ false\ \mathbf{in}\ G\rangle$
$\Leftarrow$ {assumption and definition P **ensures** Q}

$\langle \forall u : u \in F : (g.Q \vee \mathbf{wp}.u.(g.Q)) \textbf{ unless } false \textbf{ in } G \rangle$

$=$ {definition of **unless**, omitting range}

$\langle \forall u :: [\langle \forall t :: (g.Q \vee \mathbf{wp}.u.(g.Q)) \Rightarrow \mathbf{wp}.t.(g.Q \vee \mathbf{wp}.u.(g.Q))\rangle] \rangle$

$=$ {predicate calculus}

$\langle \forall u, t :: [(g.Q \vee \mathbf{wp}.u.(g.Q)) \Rightarrow \mathbf{wp}.t.(g.Q \vee \mathbf{wp}.u.(g.Q))] \rangle$

$=$ {predicate calculus}

$\langle \forall u, t :: [g.Q \Rightarrow \mathbf{wp}.t.(g.Q \vee \mathbf{wp}.u.(g.Q))] \rangle \wedge$
$\langle \forall u, t :: [\mathbf{wp}.u.(g.Q) \Rightarrow \mathbf{wp}.t.(g.Q \vee \mathbf{wp}.u.(g.Q))] \rangle$

$=$ {property G1}

$\langle \forall u, t :: [\mathbf{wp}.u.(g.Q) \Rightarrow \mathbf{wp}.t.(g.Q \vee \mathbf{wp}.u.(g.Q))] \rangle$

$\Leftarrow$ {predicate calculus}

$\langle \forall u, t :: [\mathbf{wp}.u.(g.Q) \;\Rightarrow\; \mathbf{wp}.t.(\mathbf{wp}.u.(g.Q))] \rangle$

$=$ {definition of **stable**}

$\langle \forall u :: \textbf{stable } \mathbf{wp}.u.(g.Q) \textbf{ in } G \rangle$

$\Leftarrow$ {$F \triangleleft G$}

stable $g.Q$ **in** G

$=$ {property G1}

$true$

(End of Proof)

Lemma 5 (Progress, "$\mapsto$") *Assume that programs F and G are given such that $F \triangleleft G$. For state predicates P and Q, let $g.P$ and $g.Q$ be the closures of G with respect to P and Q respectively. Then,*

$$P \mapsto Q \textbf{ in } F \;\Rightarrow\; g.P \mapsto g.Q \textbf{ in } F \;[\!]\; G.$$

Proof (of 5): The proof will be by induction on the structure of the proof of $P \mapsto Q$ **in** F.

Base Case : Assume that P **ensures** Q **in** F. Then,

P **ensures** Q **in** F

$\Rightarrow$ {Lemma 4, $F \triangleleft G$}

$g.P$ **ensures** $g.Q$ **in** $F \;[\!]\; G$

$\Rightarrow$ {definition of $\mapsto$}

$g.P \mapsto g.Q$ **in** $F \;[\!]\; G$

Induction Step (transitivity) : Assume that $P \mapsto Q$ **in** F and $Q \mapsto R$ **in** F. Then,

$(P \mapsto Q \textbf{ in } F) \wedge (Q \mapsto R \textbf{ in } F)$

$\Rightarrow$ {induction hypothesis, twice}

$(g.P \mapsto g.Q \textbf{ in } F \;[\!]\; G) \wedge (g.Q \mapsto g.R \textbf{ in } F \;[\!]\; G)$

$\Rightarrow$ {transitivity of $\mapsto$}

$(g.P \mapsto g.R \textbf{ in } F \;[\!]\; G)$

Induction Step (disjunctivity) : Assume that $\langle \forall P : P \in W : P \mapsto Q \textbf{ in } F \rangle$.

$\langle \forall P : P \in W : P \mapsto Q \textbf{ in } F \rangle$

$\Rightarrow$ {induction hypothesis}

$\langle \forall P : P \in W : g.P \mapsto g.Q \textbf{ in } F \;[\!]\; G \rangle$

$\Rightarrow$ {disjunctivity of $\mapsto$}

$\langle \exists P : P \in W : g.P \rangle \mapsto g.Q \textbf{ in } F \;[\!]\; G$

$=$ {property G6}

$g.\langle \exists P : P \in W : P \rangle \mapsto g.Q \textbf{ in } F \;[\!]\; G$

(End of Proof)

Theorem 11 *Given are programs F and G such that $F \triangleleft G$. For state predicates P and Q, we have*

$$\frac{P \mapsto Q \textbf{ in } F \qquad \textbf{stable } Q \textbf{ in } G}{P \mapsto Q \textbf{ in } F \ [\!] \ G.}$$

Proof (of 11):

$$\begin{array}{ll} & P \mapsto Q \textbf{ in } F \\ \Rightarrow & \{F \triangleleft G, \text{ Lemma 5}\} \\ & g.P \mapsto g.Q \textbf{ in } F \ [\!] \ G \\ \Rightarrow & \{\text{property G0 and } \mapsto \text{ transitivity}\} \\ & P \mapsto g.Q \textbf{ in } F \ [\!] \ G \\ = & \{ \textbf{ stable } Q \textbf{ in } G \text{ and Lemma 2}\} \\ & P \mapsto Q \textbf{ in } F \ [\!] \ G \end{array}$$

(End of Proof)