

# A New Roadmap for Linking Theories of Programming

He Jifeng<sup>(✉)</sup>

Shanghai Key Laboratory of Trustworthy Computing,  
International Research Center of Trustworthy Software,  
East China Normal University, Shanghai, China  
jifeng@sei.ecnu.edu.cn

**Abstract.** Formal methods advocate the crucial role played by the algebraic approach in specification and implementation of programs. Traditionally, a top-down approach (with denotational model as its origin) links the algebra of programs with the denotational representation by establishment of the *soundness* and *completeness* of the algebra against the given model, while a bottom-up approach (a journey started from operational model) introduces a variety of bisimulations to establish the equivalence relation among programs, and then presents a set of algebraic laws in support of program analysis and verification. This paper proposes a new roadmap for linking theories of programming. Our approach takes an algebra of programs as its foundation, and generates both denotational and operational representations from the algebraic refinement relation.

## 1 Introduction

Formal methods advocate the crucial role played by the algebra of programs in specification and implementation of programs. Study leads to the conclusion that both the top-down approach (with denotational model as its origin) and the bottom-up approach (a journey started from operational model) can meet in the middle:

- Top-down approach usually begins with construction of a specification-oriented model [1, 2, 4, 10, 15], then links the algebra of programs with the denotational framework by establishment of the *soundness* and *completeness* of the algebra [8, 13] against the given model.
- Bottom-up approach starts with an operational semantics [12] and introduces a rich variety of bisimulations [5, 11] to identify the equivalence relation among programs, and then presents a set of algebraic laws in support of program analysis and verification.

This paper proposes a new roadmap for linking theories of programming. Our framework takes an algebra of programs as its basis, and generates both denotational and operational representations from the algebraic refinement relation. This new strategy consists of the following steps:

**Step 1:** Within a given program algebra  $(\mathcal{P}, \sqsubseteq_A)$ , investigate the algebraic properties of the test operator  $\mathcal{T}$  which has test case  $tc$  and testing program  $P$  as its arguments

$$\mathcal{T}(tc, P)$$

In case of the Guarded Command Language [3],  $tc$  is represented by a total constant assignment  $x, y, \dots, z := a, b, \dots, c$  and the test operator  $\mathcal{T}$  composes  $tc$  and  $P$  in sequence:

$$\mathcal{T}(tc, P) =_{df} (tc; P)$$

For CSP [7, 14], a test case has the same alphabet as the testing process, and takes the form of a generalised prefix process  $s \rightarrow \Phi$  where  $s$  is a sequence of events in the alphabet of the process  $P$ , and  $\Phi$  a choice construct  $x : X \rightarrow Stop$  which is added to test the status of  $P$  after its engagement in the events of sequence  $s$ . The test  $\mathcal{T}(tc, P)$  behaves like the system composed of processes  $tc$  and  $P$  interacting in lock-step synchronisation

$$\mathcal{T}(tc, P) =_{df} (tc \parallel P)$$

**Step 2:** Explore the dependency between the test outcome with the test case in the following form

$$\mathcal{T}(tc, P) =_A \sqcap Obs$$

where  $Obs$  denotes the set of visible observations one can record during the execution of the test and  $\sqcap$  means the non-deterministic choices.

For the Guarded Command Language, an observation can be either a total constant assignment or the chaotic program  $\perp$  which represents the worst outcome. In case of CSP an observation has a very similar form as test case.

**Step 3:** Based on the algebra of test, identify a program  $P$  as a binary relation  $[P]$  which relates the test case with the final observation

$$[P] =_{df} \{(tc, obs) \mid \mathcal{T}(tc, P) \sqsubseteq_A obs\}$$

and select the set inclusion as the refinement relation  $\sqsubseteq_{rel}$

$$P \sqsubseteq_{rel} Q =_{df} ([P] \supseteq [Q])$$

Based on the algebra of programs, we can prove

$$\sqsubseteq_{rel} = \sqsubseteq_A$$

**Step 4:** Propose an algebraic definition of the *consistency* of step relation of the transition system of programs such that any consistent transition system  $(O, \sqsubseteq_O)$  satisfies

$$\sqsubseteq_O = \sqsubseteq_A$$

Furthermore, our approach shows how to *generates* the transition rules for CSP combinators directly from the closure properties of the *canonical processes* presented in the consistent criterion of the step relation.

The paper is organised in the following way:

Section 2 adopts this new roadmap to re-establish the semantical models of the Guarded Command Language, where

- Section 2.1 provides an algebraic representation of *machine state* and examines its properties.
- Section 2.2 introduces the notion of test cases.
- Section 2.3 presents a test-based model, where each program is identified as a binary relation between test case and visible observation recorded during the execution of the test. It is shown that the refinement relation  $\sqsubseteq_{rel}$  in the test model is equivalent to the algebraic refinement  $\sqsubseteq_A$ .
- Section 2.4 reconstructs the double predicate model with a simplified version of the refinement relation  $\sqsubseteq_{dp}$  satisfying  $\sqsubseteq_{dp} = \sqsubseteq_A$ .
- Section 2.5 revisits the predicate transformer model with the refinement relation  $\sqsubseteq_{wp}$ , and presents its link with the algebra of programs by showing  $\sqsubseteq_{wp} = \sqsubseteq_A$ .
- Section 2.6 links Hoare triple proof system with the test-based model of Sect. 2.3.

Section 3 proposes a formal definition for the consistency of step relation of transition system against the algebra of programs. Moreover, it provides a transition system for the Guarded Command Language, and establishes its correctness.

The paper ends with a short summary in Sect. 4. We leave the proof of some theorems in the appendix. We will extend the paper by adopting this new approach on CSP and probabilistic programming languages in the near future.

## 2 Guarded Command Language

This section investigates how to rediscover a variety of well-established semantical models from the program algebra presented in [9] for the Guarded Command Language:

$$\begin{aligned}
 P ::= & \perp \\
 & | \text{var} := \text{exp} \\
 & | P \triangleleft \text{bexp} \triangleright P \\
 & | P ; P \\
 & | P \sqcap P \\
 & | \mu X \bullet P(X)
 \end{aligned}$$

where the notation *bexp* stands for a Boolean expression.

Rather than following up an inductive approach to assign meaning to programs, this section develops a new mathematical framework where the behaviours of a program are described by those observations one can make during the testing of a program. To achieve this goal, we first introduce an algebra of tests and then deduce a simplified version of refinement relation in this algebra. Later we are going to derive a family of well-known denotational models [3, 4, 6, 9, 10] from the algebra of tests, and revalidate those familiar properties of programming operators.

## 2.1 Machine State

An operational approach usually defines the relationship between a program and its possible execution by machine. In an abstract way, a computation consists of a sequence of individual steps with the following features

- each step takes the machine from one state to a closely similar state
- each step is drawn from a very limited repertoire.

In a stored program computer, the machine states are represented as pairs

$$(s, P)$$

where

- (1)  $s$  is a text, defining the *data state* as an assignment of constants to all variables of the alphabet

$$x, y, \dots, z := a, b, \dots, c$$

- (2)  $P$  is a program text, representing the rest of the program that remains to be executed. When  $P$  becomes the empty text  $\epsilon$ , there is no more program to be executed. The machine state  $(t, \epsilon)$  is the last state of any execution sequence that contains it, and  $t$  presents the final value of the variables in the end of execution.

The following lemma indicates that data states are the best programs.

### Lemma 2.1

$$(s \sqsubseteq_A P) \text{ implies } (s =_A P).$$

Algebraic refinement relation on data state sets is the same as set inclusion.

### Lemma 2.2

$$\sqcap \{s_i \mid 1 \leq i \leq n\} \sqsubseteq_A t \text{ iff } t \in \{s_i \mid 1 \leq i \leq n\}$$

### Corollary

$$\sqcap \{s_i \mid 1 \leq i \leq m\} \sqsubseteq_A \sqcap \{t_j \mid 1 \leq j \leq n\} \text{ iff } \{s_i \mid 1 \leq i \leq m\} \supseteq \{t_j \mid 1 \leq j \leq n\}$$

## 2.2 Test

The execution of program  $(s; P)$  can be seen as a *test* on  $P$  with the test case  $s$ . The result of such a testing gives rise to a set of possible outcomes *obs*. We are then able to compare the behaviours of two programs based on testing. Formally, the test operator for the Guarded Command Language is defined by

$$\mathcal{T}(s, P) =_{df} (s; P)$$

When  $\perp$  is taken as the test case, we obtain

$$\mathcal{T}(\perp, P) =_A \perp$$

A test may end with delivery of a set of data states, or fail to produce any meaningful result.

**Theorem 2.1**

For any test  $\mathcal{T}(s, P)$ , either there exists a finite nonempty set  $\{t_i \mid 1 \leq i \leq n\}$  of total constant assignments such that

$$\mathcal{T}(s, P) =_A \sqcap \{t_i \mid 1 \leq i \leq n\}$$

or  $\mathcal{T}(s, P) =_A \perp$

**Proof.** Any finite program  $P$  can be converted into the finite normal form [9]

$$\perp \triangleleft b \triangleright Q$$

where  $Q$  is a nondeterministic choice on a finite nonempty set of total assignments

$$Q = \sqcap \{(v := e_i) \mid 1 \leq i \leq m\}$$

In this case, we conclude that

$$(v := c); P =_A \begin{cases} \perp & \text{if } b[c/v] = \text{true} \\ \sqcap \{(v := e_i[c/v]) \mid 1 \leq i \leq m\} & \text{if } b[c/v] = \text{false} \end{cases}$$

where  $b[c/v]$  stands for result of substituting all free occurrences of variables  $v$  in the Boolean expression  $b$  by constants  $c$ .

The behaviour of an infinite program can be represented as an infinite sequence of expressions [9]

$$S = \{S_n \mid n \in \text{Nat}\}$$

where each  $S_n$  is a finite normal form, and each  $S_{n+1}$  is stronger than its predecessor  $S_n$ :

$$(S_{n+1} \sqsupseteq_A S_n) \quad \text{for all } n \in \text{Nat}$$

This is called the *descending chain condition*. It allows the later members of the sequence to exclude more and more of impossible behaviours. The exact behaviour of the program is captured by the *least upper bound* of the whole sequence, written

$$\sqcup \{S_n \mid n \in \text{Nat}\}$$

In fact, the descending chain  $\{(\perp \triangleleft b_n \triangleright Q_n) \mid n \in \text{Nat}\}$  satisfies the following stronger order

$$(\perp \triangleleft b_n \triangleright Q_n) =_A (\perp \triangleleft b_n \triangleright Q_{n+k})$$

for all  $n, k$ . That is once  $n$  is high enough for  $b_n$  to be false, all assignments  $Q_m$  remain the same as  $Q_n$  for all  $m$  greater than  $n$ . The conclusion of the theorem follows from the continuity of sequential composition:

$$((v := c); \sqcup \{S_n \mid n \in \text{Nat}\}) =_A \begin{cases} \perp & \text{if } \forall n \in \text{Nat} \bullet (b_n[c/v] = \text{true}) \\ (v := c); Q_m & \text{if } b_m[c/v] = \text{false} \end{cases}$$

The following theorem reveals the compositionality of the testing process by demonstrating how to derive the test outcome of a composite program from those of its components.

**Theorem 2.2**

- (1)  $\mathcal{T}(s, (P \sqcap Q)) = \mathcal{T}(s, P) \sqcap \mathcal{T}(s, Q)$
- (2)  $\mathcal{T}(s, (P \triangleleft b \triangleright Q)) = \mathcal{T}(s, P) \triangleleft (s; b) \triangleright \mathcal{T}(s, Q)$  where  $(v := c); b =_{df} b[c/v]$
- (3)  $\mathcal{T}(s, (P; Q)) =_A \sqcap \{\mathcal{T}(t, Q) \mid \mathcal{T}(s, P) \sqsubseteq_A t\}$
- (4)  $\mathcal{T}(s, \mu X \bullet P(X)) =_A \sqcup \{\mathcal{T}(s, P^n(\perp)) \mid n \in \mathbb{N}at\}$  where

$$P^0(\perp) =_{df} \perp \text{ and } P^{n+1}(\perp) =_{df} P(P^n(\perp))$$

**Proof of (3).** From Theorem 2.1 we only need to consider two cases:

Case 1:  $\mathcal{T}(s, P) =_A \sqcap \{t_i \mid 1 \leq i \leq n\}$ .

$$\begin{aligned}
 & \mathcal{T}(s, (P; Q)) && \{\text{assumption}\} \\
 & =_A \sqcap \{t_i \mid 1 \leq i \leq n\}; Q && \{(U \sqcap V); W =_A (U; W) \sqcap (V; W)\} \\
 & =_A \sqcap \{\mathcal{T}(t_i, Q) \mid 1 \leq i \leq n\} && \{\text{Lemma 2.2}\} \\
 & =_A \sqcap \{\mathcal{T}(t, Q) \mid \mathcal{T}(s, P) \sqsubseteq_A t\}
 \end{aligned}$$

Case 2:  $\mathcal{T}(s, P) =_A \perp$ . The conclusion follows from the fact

$$\mathcal{T}(\perp, Q) =_A \perp$$

**Theorem 2.3.**  $P =_A Q$  iff for every data state  $s$ ,  $\mathcal{T}(s, P) =_A \mathcal{T}(s, Q)$

**2.3 A Test-Based Model**

As described in the previous section, the execution of test  $\mathcal{T}(s, P)$  may yield a finite nonempty set of outcomes. In the worst case, it may end with a chaotic state. In this sense, each testing program  $P$  can be treated as a binary relation on test cases and final observations. This section is going to construct a relation model from the test algebra.

**Definition 2.1**

A program  $P$  can be identified as a binary relation  $[P]$  between test case  $s$  with the final data state  $t$  it may enter in the end of testing.

$$[P] =_{df} \{(s, t) \mid \mathcal{T}(s, P) \sqsubseteq_A t\}$$

As usual we define the refinement relation  $\sqsubseteq_{rel}$  on the relational model by the set inclusion

$$P \sqsubseteq_{rel} Q =_{df} ([P] \supseteq [Q])$$

**Theorem 2.4**

$$\sqsubseteq_{rel} = \sqsubseteq_A$$

**Proof.** Assume that  $P \sqsubseteq_{rel} Q$ .

Case 1:  $\mathcal{T}(s, Q) =_A \perp$

$$\begin{aligned}
 & \mathcal{T}(s, Q) \sqsubseteq_A \perp && \{\text{Definition 2.1}\} \\
 \Rightarrow & \mathcal{T}(s, P) \sqsubseteq_A \perp && \{\text{assumption}\} \\
 \Rightarrow & \mathcal{T}(s, P) \sqsubseteq_A \mathcal{T}(s, Q)
 \end{aligned}$$

Case 2:  $\mathcal{T}(s, Q) =_A \sqcap \{t_i \mid 1 \leq i \leq n\}$ .

$$\mathcal{T}(s, Q) =_A \sqcap \{t_i \mid 1 \leq i \leq n\} \quad \{\text{Lemma 2.2}\}$$

$$\Rightarrow \forall i \in \{1, \dots, n\} \bullet (\mathcal{T}(s, Q) \sqsubseteq_A t_i) \quad \{\text{Definition 2.1}\}$$

$$\Rightarrow \forall i \in \{1, \dots, n\} \bullet (\mathcal{T}(s, P) \sqsubseteq_A t_i) \quad \{\text{monotonicity of } \sqcap\}$$

$$\Rightarrow \mathcal{T}(s, P) \sqsubseteq_A \sqcap \{t_i \mid 1 \leq i \leq n\} \quad \{\text{assumption}\}$$

$$\Rightarrow \mathcal{T}(s, P) \sqsubseteq_A \mathcal{T}(s, Q)$$

The conclusion  $P \sqsubseteq_A Q$  follows from Theorem 2.3.

The opposite inequation  $(\sqsubseteq_A \subseteq \sqsubseteq_{rel})$  follows from Theorem 2.2(1) and the definition of  $\sqsubseteq_{rel}$ .

Moreover, the mapping  $[\ ]$  is a homomorphism.

### Theorem 2.5

- (1)  $[P \sqcap Q] = [P] \cup [Q]$
- (2)  $[P \triangleleft b \triangleright Q] = [P] \triangleleft (s; b) \triangleright [Q]$
- (3)  $[P; Q] = [P] \circ [Q]$
- (4)  $[\mu X \bullet P(X)] = \bigcap_n [P^n(\perp)]$

#### Proof of (3)

$$\begin{aligned}
 & (s, t) \in [P; Q] && \{\text{Definition 2.1}\} \\
 \equiv & \mathcal{T}(s, (P; Q)) \sqsubseteq_A t && \{\text{Theorem 2.2(3)}\} \\
 \equiv & \sqcap \{\mathcal{T}(u, Q) \mid \mathcal{T}(s, P) \sqsubseteq_A u\} \sqsubseteq_A t && \{\text{Lemma 2.2}\} \\
 \equiv & \exists u \bullet (\mathcal{T}(s, P) \sqsubseteq_A u) \wedge (\mathcal{T}(u, Q) \sqsubseteq_A t) && \{\text{Definition 2.1}\} \\
 \equiv & \exists u \bullet ((s, u) \in [P]) \wedge ((u, t) \in [Q]) && \{\text{Definition of } \circ\} \\
 \equiv & (s, t) \in ([P] \circ [Q])
 \end{aligned}$$

## 2.4 Double Predicates Model

In [6], a precondition is defined as a predicate describing the initial values of program variables of a program before it is activated, whereas a postcondition is a predicate only mention of the final values of program variables after the execution of a program terminates. Following the VDM approach [10] we permit a postcondition to refer to both initial and final values of program variables in the following discussion.

### Definition 2.2 (Double predicates)

$$\begin{aligned}
 \text{pre}(P)(v_0) &=_{df} \neg(\mathcal{T}(v := v_0, P) \sqsubseteq_A \perp) \\
 \text{post}(P)(v_0, v') &=_{df} \mathcal{T}(v := v_0, P) \sqsubseteq_A (v := v')
 \end{aligned}$$

where  $v_0$  and  $v'$  stand for the initial and final values of the program variables  $v$ .

In the above definition, the postcondition meets the following constraint, which enables us to simplify the definition of refinement in the double predicate model later.

**Theorem 2.6.**  $\mathbf{post}(P) \equiv (\mathbf{pre}(P) \Rightarrow \mathbf{post}(P))$

The refinement order  $\sqsubseteq_{dp}$  on the double predicate model is defined by

$$P \sqsubseteq_{dp} Q =_{df} \forall v_0 \bullet (\mathbf{pre}(P) \Rightarrow \mathbf{pre}(Q)) \wedge \forall v_0, v' \bullet (\mathbf{post}(Q) \Rightarrow \mathbf{post}(P))$$

**Theorem 2.7**

$$\begin{aligned}
& \sqsubseteq_A = \sqsubseteq_{dp} \\
& P \sqsubseteq_A Q \quad \{\text{Theorem 2.3}\} \\
& \equiv \forall v_0 \bullet \mathcal{T}(v := v_0, P) \sqsubseteq_A \mathcal{T}(v := v_0, Q) \quad \{\text{Theorem 2.1}\} \\
& \equiv \forall v_0 \bullet \left( \left( \mathcal{T}(v := v_0, Q) \sqsubseteq_A \perp \Rightarrow \right) \wedge \right. \\
& \quad \left. \forall v' \bullet \left( (\mathcal{T}(v := v_0, Q) \sqsubseteq_A (v := v')) \Rightarrow \right) \right) \quad \{\text{Definition 2.2}\} \\
& \equiv \left( \forall v_0 \bullet (\mathbf{pre}(P) \Rightarrow \mathbf{pre}(Q)) \wedge \right. \\
& \quad \left. \forall v_0, v' \bullet (\mathbf{post}(Q) \Rightarrow \mathbf{post}(P)) \right) \quad \{\text{Def of } \sqsubseteq_{dp}\} \\
& \equiv P \sqsubseteq_{dp} Q
\end{aligned}$$

Definition 2.2 enables us to transform the original definition of the programming combinators in the double predicates model [9, 10] into a set of the compositional laws in our test-generated model:

**Theorem 2.8**

$$(1) \mathbf{pre}(\perp) \equiv \mathbf{false}$$

$$(2) \mathbf{pre}(P \sqcap Q) \equiv \mathbf{pre}(P) \wedge \mathbf{pre}(Q)$$

$$(3) \mathbf{pre}(P \triangleleft b(v) \triangleright Q) \equiv \mathbf{pre}(P) \triangleleft b(v_0) \triangleright \mathbf{pre}(Q)$$

$$(4) \mathbf{pre}(P; Q) \equiv \mathbf{pre}(P) \wedge \neg \exists c \bullet (\mathbf{post}(P)[c/v'] \wedge \neg \mathbf{pre}(Q)[c/v_0])$$

$$(5) \mathbf{pre}(\mu X \bullet P(X)) = \bigvee_n \mathbf{pre}(P^n(\perp))$$

**Proof of (4)**

$$\begin{aligned}
& \neg \mathbf{pre}(P; Q)(v_0) \quad \{\text{Defintion 2.2}\} \\
& \equiv \mathcal{T}(v := v_0, (P; Q)) =_A \perp \quad \{\text{Theorem 2.2(3)}\} \\
& \equiv \mathcal{T}(v := v_0, P) =_A \perp \vee \\
& \quad \exists t \bullet \mathcal{T}(v := v_0, P) \sqsubseteq_A t \wedge \mathcal{T}(t, Q) =_A \perp \quad \{\text{Definition 2.2}\} \\
& \equiv \neg \mathbf{pre}(P)(v_0) \vee \exists c \bullet \mathbf{post}(P)(v_0, c) \wedge \neg \mathbf{pre}(Q)(c)
\end{aligned}$$



**Theorem 2.9**

- (1)  $\mathbf{post}(\perp) \equiv \mathbf{true}$
- (2)  $\mathbf{post}(P \sqcap Q) \equiv \mathbf{post}(P) \vee \mathbf{post}(Q)$
- (3)  $\mathbf{post}(P \triangleleft b(v) \triangleright Q) \equiv \mathbf{post}(P) \triangleleft b(v_0) \triangleright \mathbf{post}(Q)$
- (4)  $\mathbf{post}(P; Q) \equiv \exists c \bullet (\mathbf{post}(P)[c/v'] \wedge \mathbf{post}(Q)[c/v_0])$
- (5)  $\mathbf{post}(\mu X \bullet P(X)) \equiv \bigwedge_n \mathbf{post}(P^n(\perp))$

**Proof** From Theorem 2.2.

**2.5 Predicate Transformer**

Given a postcondition  $r$  and a proposed design of a final program segment  $Q$ , it is possible to deduce the *weakest* precondition under which the execution of  $Q$  will end with the states that satisfy the postcondition  $r$ . This precondition can often be strengthening, and then taken as the postcondition in the design of the next preceding segment of the program. In this method, a program is identified as a predicate transformer mapping the given postcondition to the corresponding weakest precondition [3]. Based on the algebra of tests, this section redefines the predicate transformer as follows:

**Definition 2.3 (Weakest precondition)**

Define

$$\mathbf{wp}(Q, r)(v_0) =_{df} \mathcal{T}(v := v_0, Q) \sqsubseteq_A \sqcap \{v := c \mid r(c)\}$$

In our method, the refinement relation  $\sqsubseteq_{wp}$  is defined by

$$P \sqsubseteq_{wp} Q =_{df} \forall r, \forall v_0 \bullet (\mathbf{wp}(P, r)(v_0) \Rightarrow \mathbf{wp}(Q, r)(v_0))$$

**Theorem 2.10**

$$\sqsubseteq_{wp} = \sqsubseteq_A$$

**Proof**

$$\begin{aligned}
& P \sqsubseteq_A Q && \{\text{Theorem 2.3}\} \\
\equiv & \forall v_0 \bullet (\mathcal{T}(v := v_0, P) \sqsubseteq_A \mathcal{T}(v := v_0, Q)) && \{\text{Transitivity of } \sqsubseteq_A\} \\
\equiv & \forall r, \forall v_0 \bullet && \\
& \left( \mathcal{T}(v := v_0, P) \sqsubseteq_A \sqcap \{v := c \mid r(c)\} \Rightarrow \right. && \{\text{Definition 2.3}\} \\
& \left. \mathcal{T}(v := v_0, Q) \sqsubseteq_A \sqcap \{v := c \mid r(c)\} \right) && \\
\equiv & \forall r, \forall v_0 \bullet (\mathbf{wp}(P, r)(v_0) \Rightarrow \mathbf{wp}(Q, r)(v_0)) && \{\text{Definition of } \sqsubseteq_{wp}\} \\
\equiv & P \sqsubseteq_{wp} Q && 
\end{aligned}$$

Like in Sects. 2.3 and 2.4, the new definition of the predicate transformer enables us to verify the following family of so called *healthiness* conditions presented in [3].

### Theorem 2.11

- (1)  $\mathbf{wp}(Q, \text{false}) = \text{false}$
- (2)  $\mathbf{wp}(Q, r_1 \wedge r_2) = \mathbf{wp}(Q, r_1) \wedge \mathbf{wp}(Q, r_2)$

#### Proof of (2)

$$\begin{aligned}
 & \mathbf{wp}(Q, r_1) \wedge \mathbf{wp}(Q, r_2) && \{\text{Definition 2.3}\} \\
 \equiv & \left( \mathcal{T}(v := v_0, Q) \sqsubseteq_A \sqcap \{v := c \mid r_1(c)\} \wedge \right. && \{\text{Corollary of Lemma 2.2}\} \\
 & \left. \mathcal{T}(v := v_0, Q) \sqsubseteq_A \sqcap \{v := c \mid r_2(c)\} \right) \\
 \equiv & \mathcal{T}(v := v_0, Q) \sqsubseteq_A \sqcap \{v := c \mid (r_1 \wedge r_2)(c)\} && \{\text{Definition 2.3}\} \\
 \equiv & \mathbf{wp}(Q, r_1 \wedge r_2)
 \end{aligned}$$

The next theorem links the double predicates model with the predicate transformer model.

### Theorem 2.12

$$\mathbf{wp}(Q, r(v)) \equiv \mathbf{pre}(Q) \wedge \neg \exists c \bullet (\mathbf{post}(Q)[c/v'] \wedge \neg r(c))$$

$$\begin{aligned}
 & \mathbf{wp}(Q, r(v)) && \{\text{Definition 2.3}\} \\
 \equiv & \mathcal{T}(v := v_0, Q) \sqsubseteq_A \sqcap \{v := c \mid r(c)\} && \{\text{Theorem 2.1}\} \\
 \text{Proof} \quad \equiv & \left( (\mathcal{T}(v := v_0, Q) \not\sqsubseteq_A \perp) \wedge \right. && \{\text{Definition 2.2}\} \\
 & \left. \forall t \bullet (\mathcal{T}(v := v_0, Q) \sqsubseteq_A t \Rightarrow (t \in \{v := c \mid r(c)\})) \right) \\
 \equiv & \mathbf{pre}(Q) \wedge \forall c \bullet (\mathbf{post}(Q)[c/v'] \Rightarrow r(c)) && \{\text{calculation}\} \\
 \equiv & \mathbf{pre}(Q) \wedge \neg \exists c \bullet (\mathbf{post}(Q)[c/v'] \wedge \neg r(c))
 \end{aligned}$$

### Corollary

- (1)  $\mathbf{pre}(P) \equiv \mathbf{wp}(P, \text{true})$
- (2)  $\mathbf{post}(P) \equiv \neg \mathbf{wp}(P, v \neq v')$

The following theorem validates the original definition of the predicate transformer given in [3].

### Theorem 2.13

- (1)  $\mathbf{wp}(\perp, r) \equiv \text{false}$
- (2)  $\mathbf{wp}(P \sqcap Q, r) \equiv \mathbf{wp}(P, r) \wedge \mathbf{wp}(Q, r)$
- (3)  $\mathbf{wp}(P \triangleleft b(v) \triangleright Q, r) \equiv \mathbf{wp}(P, r) \triangleleft b(v_0) \triangleright \mathbf{wp}(Q, r)$
- (4)  $\mathbf{wp}(P; Q, r) \equiv \mathbf{wp}(P, \mathbf{wp}(Q, r))$
- (5)  $\mathbf{wp}(\mu X \bullet P(X)) \equiv \bigvee_n \mathbf{wp}(P^n(\perp), r)$

**Proof of (4)**

$$\begin{aligned}
& \mathbf{wp}(P, \mathbf{wp}(Q, r)) && \{\text{Theorem 2.12}\} \\
\equiv & \mathbf{pre}(P) \wedge \neg \exists c \bullet \left( \mathbf{post}(P)[c/v'] \wedge \right. && \{\text{Theorem 2.12}\} \\
& \quad \left. \neg \mathbf{wp}(Q, r)[c/v_0] \right) \\
\equiv & \left( \mathbf{pre}(P) \wedge \neg \exists c \bullet \left( \mathbf{post}(P)[c/v'] \wedge \right. \right. && \{\text{calculation}\} \\
& \quad \left. \neg \exists c \bullet \left( \left( \neg \mathbf{pre}(Q)[c/v_0] \vee \right. \right. \right. \\
& \quad \quad \left. \left. \left. \exists d \bullet \left( \mathbf{post}(Q)[c, d/v_0, v'] \wedge \neg r(d) \right) \right) \right) \right) \right) \\
\equiv & \left( \mathbf{pre}(P) \wedge \neg \exists c \bullet \left( \mathbf{post}(P) \wedge \neg \mathbf{pre}(Q)[c/v_0] \right) \wedge \right. && \{\text{Theorem 2.8 and 2.9}\} \\
& \quad \left. \neg \exists d \bullet \left( \exists c \bullet \left( \mathbf{post}(P)[c/v_0] \wedge \mathbf{post}(Q)[c, d/v_0, v'] \right) \wedge \neg r(d) \right) \right) \\
\equiv & \mathbf{pre}(P; Q) \wedge \neg \exists d \bullet (\mathbf{post}(P; Q)[d/v'] \wedge \neg r(d)) && \{\text{Theorem 2.12}\} \\
\equiv & \mathbf{wp}(P; Q, r)
\end{aligned}$$

**2.6 Hoare Triple**

In [6], the correctness of a program was interpreted as the triple

$$\text{precondition } \{ \text{program} \} \text{ postcondition}$$

known as a Hoare triple, where the postcondition only refers to the final values of program variables.

**Definition 2.4 (Hoare triple)**

Define

$$p\{Q\}r \stackrel{\text{df}}{=} \forall v_0 \bullet (p(v_0) \Rightarrow (\mathcal{T}(v := v_0, Q) \sqsubseteq_A \sqcap \{v := c \mid r(c)\}))$$

**Theorem 2.14**

$$p\{Q\}r \equiv \forall v_0 \bullet (p(v_0) \Rightarrow \mathbf{wp}(Q, r(v)))$$

**Proof.** From Definitions 2.3 and 2.4.

Based on the new definition of Hoare triple we are able to reestablish the soundness of Hoare logic used for verification of programs.

**Theorem 2.15 (Hoare triple proof rules)**

- (1) If  $p\{Q\}r_1$  and  $p\{Q\}r_2$  then  $p\{Q\}(r_1 \wedge r_2)$
- (2) If  $p\{Q\}r$  and  $q\{Q\}r$  then  $(p \vee q)\{Q\}r$
- (3) If  $p\{Q\}r$  then  $p\{Q\}(q \vee r)$
- (4)  $r(e)\{v := e\}r(v)$
- (5) If  $(p \wedge b)\{Q_1\}r$  and  $(p \wedge \neg b)\{Q_2\}r$  then  $p\{Q_1 \triangleleft b \triangleright Q_2\}r$
- (6) If  $p\{Q_1\}q$  and  $q\{Q_2\}r$  then  $p\{Q_1; Q_2\}r$

- (7) If  $p\{Q_1\}r$  and  $p\{Q_2\}r$  then  $p\{Q_1 \sqcap Q_2\}r$   
 (8)  $false\{Q\}r$

### Proof of (6)

$$\begin{aligned}
 & p\{Q_1; Q_2\}r && \{\text{Theorem 2.14}\} \\
 \equiv & \forall v_0 \bullet (p(v_0) \Rightarrow \mathbf{wp}(Q_1; Q_2, r)) && \{\text{Theorem 2.13(4)}\} \\
 \equiv & \forall v_0 \bullet (p(v_0) \Rightarrow \mathbf{wp}(Q_1, \mathbf{wp}(Q_2, r))) && \{\text{Theorem 2.11(2) and } q\{Q_2\}r\} \\
 \Leftarrow & \forall v_0 \bullet (p(v_0) \Rightarrow \mathbf{wp}(Q_1, q)) && \{p\{Q_1\}q\} \\
 \equiv & true
 \end{aligned}$$

## 3 Operational Approach

Let  $\rightarrow$  be a step relation on machine states. Its *reflexive transitive closure* is defined by

$$\rightarrow^* =_{df} \nu X \bullet (id \vee (\rightarrow; X))$$

where  $\nu X.G(X)$  stands for the *greatest fixed point* of function  $G$ .

We define the concept of *divergence*, being a machine state that can lead to an infinite execution

$$(s, P) \uparrow =_{df} \forall n, \exists t, Q \bullet ((s, P) \rightarrow^n (t, Q))$$

where  $\rightarrow^1 =_{df} \rightarrow$   
 and  $\rightarrow^{n+1} =_{df} (\rightarrow^1; \rightarrow^n)$

### Definition 3.1

A step relation is *consistent* with the algebraic semantics if for any machine state  $(s, P)$

- (1)  $\mathcal{T}(s, P) =_A \sqcap \{\mathcal{T}(t, Q) \mid (s, P) \rightarrow (t, Q)\}$ , and  
 (2)  $(s, P) \uparrow$  implies  $\mathcal{T}(s, P) =_A \perp$

In the following discussion we will extend the definition of the test operator to cope with the empty program text

$$\mathcal{T}(s, \epsilon) =_{df} s$$

### Theorem 3.1

If  $\rightarrow$  is consistent then

$$\mathcal{T}(s, P) =_A \perp \triangleleft (s, P) \uparrow \triangleright \sqcap \{t \mid (s, P) \rightarrow^* (t, \epsilon)\}$$

**Proof.** First we show that  $\mathcal{T}(s, P) =_A \perp \Rightarrow (s, P) \uparrow$

From Theorem 2.2 and the condition (1) of Definition 3.1 it follows that there exists machine state  $(t, Q)$  such that

$$(s, P) \rightarrow (t, Q) \text{ and } \mathcal{T}(t, Q) =_A \perp$$

With induction we conclude that for all  $n \geq 0$  there exists a machine state  $(t_n, Q_n)$  satisfying

$$\mathcal{T}(t_n, Q_n) =_A \perp \text{ and } (s, P) \rightarrow^n (t_n, Q_n)$$

which leads to the conclusion  $(s, P) \uparrow$

If  $(s, P)$  is not divergent, then from the condition (1) of Definition 3.1 we can show by induction

$$\mathcal{T}(s, P) =_A \sqcap \{t \mid (s, P) \rightarrow^* (t, \epsilon)\}$$

Definition 3.1 explores the following correspondence between the step relation of the operational semantics with the refinement relation of algebraic semantics

- (1) Whenever a machine state  $(s, P)$  is divergent, then the execution of test  $\mathcal{T}(s, P)$  will end with a chaotic state.
- (2) If  $(s, P)$  is not divergent, then the final states that it can reach via step transitions are exactly those delivered by the execution of the test  $\mathcal{T}(s, P)$ .

### Definition 3.2 (Operational Refinement)

Let  $\rightarrow$  be a consistent step relation. Define

$$P \sqsubseteq_O Q \stackrel{df}{=} \left( \begin{array}{l} \forall s \bullet ((s, Q) \uparrow \Rightarrow (s, P) \uparrow) \wedge \\ \forall t \bullet ((s, Q) \rightarrow^* (t, \epsilon)) \Rightarrow \left( \begin{array}{l} (s, P) \uparrow \vee \\ ((s, P) \rightarrow^* (t, \epsilon)) \end{array} \right) \end{array} \right)$$

### Theorem 3.2

$$\sqsubseteq_O = \sqsubseteq_A$$

#### Proof

$$\begin{aligned} & P \sqsubseteq_A Q && \{\text{Theorem 2.3}\} \\ \equiv & \forall s \bullet (\mathcal{T}(s, P) \sqsubseteq_A \mathcal{T}(s, Q)) && \{\text{Theorem 2.1}\} \\ \equiv & \forall s \bullet \left( \begin{array}{l} (\mathcal{T}(s, Q) =_A \perp) \Rightarrow (\mathcal{T}(s, P) =_A \perp) \\ \wedge \\ \forall t \bullet (\mathcal{T}(s, Q) \sqsubseteq_A t) \Rightarrow (\mathcal{T}(s, P) \sqsubseteq_A t) \end{array} \right) && \{\text{Lemma 2.2 and Theorem 3.1}\} \\ \equiv & \forall s \bullet \left( \begin{array}{l} ((s, Q) \uparrow \Rightarrow (s, P) \uparrow) \wedge \\ \forall t \bullet ((s, Q) \rightarrow^* (t, \epsilon)) \Rightarrow \left( \begin{array}{l} (s, P) \uparrow \vee \\ ((s, P) \rightarrow^* (t, \epsilon)) \end{array} \right) \end{array} \right) && \{\text{Definition 3.2}\} \\ \equiv & P \sqsubseteq_O Q \end{aligned}$$

If  $(s, P)$  is not divergent, then from the condition (1) of Definition 3.1 we can show by induction

$$\mathcal{T}(s, P) =_A \sqcap \{t \mid (s, P) \rightarrow^* (t, \epsilon)\}$$

Definition 3.1 explores the following correspondence between the step relation of the operational semantics with the refinement relation of algebraic semantics

- (1) Whenever a machine state  $(s, P)$  is divergent, then the execution of test  $\mathcal{T}(s, P)$  will end with a chaotic state.
- (2) If  $(s, P)$  is not divergent, then the final states that it can reach via step transitions are exactly those delivered by the execution of the test  $\mathcal{T}(s, P)$ .

### Definition 3.3

In [9], the following transition system was given to the Guarded Command Language:

- (1) Assignment  
 $(s, v := e) \rightarrow (s; (v := e), \epsilon)$ , where  $(v := c); (v := e) = (v := e[c/v])$
- (2) Choice  
 $(a) ((s, P \sqcap Q) \rightarrow (s, P))$   
 $(b) ((s, P \sqcap Q) \rightarrow (s, Q))$
- (3) Conditional  
 $(a) (s, P \triangleleft b \triangleright Q) \rightarrow (s, P) \quad \text{if } (s; b) = \text{true}$   
 $(b) (s, P \triangleleft b \triangleright Q) \rightarrow (s, Q) \quad \text{if } (s; b) = \text{false}$
- (4) Composition  
 $(a) (s, P; Q) \rightarrow (t, R; Q) \quad \text{if } (s, P) \rightarrow (t, R)$   
 $(b) (s, P; Q) \rightarrow (t, Q) \quad \text{if } (s, P) \rightarrow (t, \epsilon)$
- (5) Recursion  
 $(s, \mu X \bullet P(X)) \rightarrow (s, P(\mu X \bullet P(X)))$
- (6) Chaos  
 $(s, \perp) \rightarrow (s, \perp)$

In the following we are going to establish the consistency of the step relation of Definition 3.3 with respect to the algebra of programs.

First, we show that the step relation of Definition 3.3 meets the condition (1) of Definition 3.1.

### Theorem 3.3

$$\mathcal{T}(s, P) =_A \sqcap \{ \mathcal{T}(t, Q) \mid (s, P) \rightarrow (t, Q) \}$$

**Proof:** Direct from Theorem 2.2 and the rule (1)–(7) of Definition 3.3.

### Theorem 3.4

If  $P$  is a finite program, then

$$(s, P) \uparrow \Rightarrow \mathcal{T}(s, P) =_A \perp$$

**Proof:** We give an induction proof based on the structure of program text  $P$ :

**Base case:** Clearly the conclusion holds for the case  $P = v := e$  and  $P = \perp$

**Inductive step:**

$$\begin{aligned}
& (s, P_1 \sqcap P_2) \uparrow && \{\text{Rule (2) in Definition 3.3}\} \\
\Rightarrow & (s, P_1) \uparrow \sqcap (s, P_2) \uparrow && \{\text{Induction hypothesis}\} \\
\Rightarrow & (\mathcal{T}(s, P_1) =_A \perp) \vee (\mathcal{T}(s, P_2) =_A \perp) && \{\text{Theorem 2.2(1)}\} \\
\Rightarrow & \mathcal{T}(s, P_1 \sqcap P_2) =_A \perp \\
\\
& (s, P_1 \triangleleft b \triangleright P_2) \uparrow && \{\text{Rule (3) in Definition 3.3}\} \\
\Rightarrow & (s, P_1) \uparrow \triangleleft s; b \triangleright (s, P_2) \uparrow && \{\text{Induction hypothesis}\} \\
\Rightarrow & (\mathcal{T}(s, P_1) =_A \perp) \triangleleft s; b \triangleright (\mathcal{T}(s, P_2) =_A \perp) && \{\text{Theorem 2.2(2)}\} \\
\Rightarrow & \mathcal{T}(s, P_1 \triangleleft b \triangleright P_2) =_A \perp \\
\\
& (s, P_1; P_2) \uparrow && \{\text{Rule (4) in Definition 3.3}\} \\
\Rightarrow & (s, P_1) \uparrow \vee \exists t \bullet \left( (s, P_1) \rightarrow (t, \epsilon) \wedge \right. && \{\text{Induction hypothesis}\} \\
& \quad \left. (t, P_2) \uparrow \right) \\
\Rightarrow & \mathcal{T}(s, P_1) =_A \perp \vee \\
& \quad \exists t \bullet (\mathcal{T}(s, P_1) \sqsubseteq_A t \wedge \mathcal{T}(t, P_2) =_A \perp) && \{\text{Theorem 2.2(3)}\} \\
\Rightarrow & \mathcal{T}(s, P_1; P_2) =_A \perp
\end{aligned}$$

Finally we are going to tackle infinite programs.

**Lemma 3.1**

If  $(s, G(Q)) \rightarrow^* (t, \epsilon)$ , then either  $(s, G(\perp)) \uparrow$  or  $(s, G(\perp)) \rightarrow^* (t, \epsilon)$ .

**Proof:** See Appendix.

**Lemma 3.2**

- (1)  $(s, F(P)) \uparrow \Rightarrow (s, \mathcal{F}(\perp)) \uparrow$  for any program  $P$ .
- (2)  $(s, F(\mu X \bullet P(X))) \uparrow \Rightarrow (s, F(P(\mu X \bullet P(X)))) \uparrow$

**Proof:** See Appendix.

**Theorem 3.5**

$(s, F(\mu X \bullet P(X))) \uparrow \Rightarrow \mathcal{T}(s, F(\mu X \bullet P(X))) =_A \perp$

**Proof:**

$$\begin{aligned}
& (s, F(\mu X \bullet P(X))) \uparrow && \{\text{Lemma 3.2(2)}\} \\
\Rightarrow & \forall n \bullet (s, F(P^n(\mu X \bullet P(X)))) \uparrow && \{\text{Lemma 3.2(1)}\} \\
\Rightarrow & \forall n \bullet (s, F(P^n(\perp))) \uparrow && \{\text{Theorem 3.4}\} \\
\Rightarrow & \forall n \bullet \mathcal{T}(s, F(P^n(\perp))) =_A \perp && \{\text{Continuity of } F\} \\
\Rightarrow & \mathcal{T}(s, F(\mu X \bullet P(X))) =_A \perp
\end{aligned}$$

Combining Theorems 3.3, 3.4 and 3.5 we conclude

**Theorem 3.6**

The step relation defined in Definition 3.3 is consistent.

**4 Conclusion**

This paper proposes a new roadmap for linking theories of programming. From the investigation of the Guarded Command Language it becomes clear that algebraic refinement relation plays a key role in building various denotational models and their links. Our work also shows that the formalisation of consistency of operational semantics can be simplified by separation progress requirement (condition (1) in Definition 3.2) from livelock-free constraint (condition (2)):

- The first requirement excludes the error of omission of a transition. Validation of the consistent condition (1) is quite straightforward, because it only needs to examine one step transition.
- the second requirement avoids the inclusion of too many transitions. To handle this type of livelock free properties, this paper adopts quite tedious structural induction because it has to deal with recursion and multiple step transition.

We will extend this paper by applying this algebraic approach to build the mathematical framework for CSP and probabilistic programming languages in the near future.

**Acknowledgements.** This work is supported by National Natural Science Foundation of China (Grant No. 61321064), Shanghai Knowledge Service Platform Project (No. ZF1213) and the NSFC-Zhejiang Joint Fund for the Integration of Industrialization and Informatization (No. U1509219).

**Appendix****Lemma 3.1**

If  $(s, G(Q)) \rightarrow^* (t, \epsilon)$ , then either  $(s, G(\perp)) \uparrow$  or  $(s, G(\perp)) \rightarrow^* (t, \epsilon)$ .

**Proof:** Induction on the structure of  $G$ .

**Base case:**  $G(Q) = Q$ . The conclusion follows from From Rule (6)

$$(s, \perp) \rightarrow (s, \perp)$$

in Definition 3.2.

**Inductive step:**

(1)  $G(Q) = G_1(Q) \sqcap G_2(Q)$ .

$$\begin{aligned}
 & (s, G(Q)) \rightarrow^* (t, \epsilon) && \{\text{Role (2) in Def 3.2}\} \\
 \Rightarrow & \left( \begin{array}{l} (s, G_1(Q)) \rightarrow^* (t, \epsilon) \vee \\ (s, G_2(Q)) \rightarrow^* (t, \epsilon) \end{array} \right) && \{\text{Induction hypothesis}\} \\
 \Rightarrow & \left( \begin{array}{l} (s, G_1(\perp)) \uparrow \vee (s, G_1(\perp)) \rightarrow^* (t, \epsilon) \vee \\ (s, G_2(\perp)) \uparrow \vee (s, G_2(\perp)) \rightarrow^* (t, \epsilon) \end{array} \right) && \{\text{Role (2) in Def 3.2}\} \\
 \Rightarrow & (s, G(\perp)) \uparrow \vee (s, G(\perp)) \rightarrow^{ast} (t, \epsilon)
 \end{aligned}$$



$$\begin{aligned}
(2) \quad & G(Q) = G_1(Q) \triangleleft b \triangleright G_2(X) \\
& (s, G(Q)) \rightarrow^* (t, \epsilon) \quad \{\text{Role (3) in Def 3.2}\} \\
& \Rightarrow \left( \begin{array}{c} (s, G_1(Q)) \rightarrow^* (t, \epsilon) \\ \triangleleft(s; b) \triangleright \\ (s, G_2(Q)) \rightarrow^* (t, \epsilon) \end{array} \right) \quad \{\text{Induction hypothesis}\} \\
& \Rightarrow \left( \begin{array}{c} (s, G_1(\perp)) \uparrow \vee (s, G_1(\perp)) \rightarrow^* (t, \epsilon) \\ \triangleleft(s; b) \triangleright \\ (s, G_2(\perp)) \uparrow \vee (s, G_2(\perp)) \rightarrow^* (t, \epsilon) \end{array} \right) \quad \{\text{Role (3) in Def 3.2}\} \\
& \Rightarrow (s, G(\perp)) \uparrow \vee (s, G(\perp)) \rightarrow^{ast} (t, \epsilon) \\
(3) \quad & G(Q) = G_1(Q); G_2(Q) \\
& (s, G(Q)) \rightarrow^* (t, \epsilon) \quad \{\text{Role (4) in Def 3.2}\} \\
& \Rightarrow \exists u \bullet \left( \begin{array}{c} (s, G_1(Q)) \rightarrow^* (u, \epsilon) \vee \\ (u, G_2(Q)) \rightarrow^* (t, \epsilon) \end{array} \right) \quad \{\text{Induction hypothesis}\} \\
& \Rightarrow \exists u \bullet \left( \begin{array}{c} (s, G_1(\perp)) \uparrow \vee (s, G_1(\perp)) \rightarrow^* (u, \epsilon) \vee \\ (u, G_2(\perp)) \uparrow \vee (u, G_2(\perp)) \rightarrow^* (t, \epsilon) \end{array} \right) \quad \{\text{Role (4) in Def 3.2}\} \\
& \Rightarrow (s, G(\perp)) \uparrow \vee (s, G(\perp)) \rightarrow^{ast} (t, \epsilon) \\
(4) \quad & G(Q) = \mu X \bullet P(Q, X) \\
& (s, \mu X \bullet P(G, X)) \rightarrow (t, \epsilon) \quad \{\text{Role (5) in Def 3.2}\} \\
& \Rightarrow (s, P(G, \mu X \bullet P(G, X))) \rightarrow (t, \epsilon) \quad \{\text{Induction hypothesis}\} \\
& \Rightarrow \left( \begin{array}{c} (s, P(\perp, \mu X \bullet P(\perp, X))) \uparrow \vee \\ (s, P(\perp, \mu X \bullet P(\perp, X))) \rightarrow (t, \epsilon) \end{array} \right) \quad \{\text{Role (5) in Def 3.2}\} \\
& \Rightarrow \left( \begin{array}{c} (s, \mu X \bullet P(\perp, X)) \uparrow \vee \\ (s, \mu X \bullet P(\perp, X)) \rightarrow (t, \epsilon) \end{array} \right)
\end{aligned}$$

### Lemma 3.2

$$\begin{aligned}
(1) \quad & (s, F(P)) \uparrow \Rightarrow (s, \mathcal{F}(\perp)) \uparrow \\
(2) \quad & (s, F(\mu X \bullet P(X))) \uparrow \Rightarrow (s, F(P(\mu X \bullet P(X)))) \uparrow
\end{aligned}$$

**Proof (1).** Based on induction on the structure of  $F$ .

Base case:  $F(X) = X$ . The conclusion follows from the rule (6).

Inductive Step:

$$\begin{aligned}
& (s, F_1(Q) \sqcap F_2(Q)) \uparrow \quad \{\text{rule (2)}\} \\
\Rightarrow & (s, F_1(Q)) \uparrow \vee (s, F_2(Q)) \uparrow \quad \{\text{induction hypothesis}\} \\
\Rightarrow & (s, F_1(\perp)) \uparrow \vee (s, F_2(\perp)) \uparrow \quad \{\text{rule (2)}\} \\
\Rightarrow & (s, (F_1(\perp) \sqcap F_2(\perp))) \uparrow \\
& (s, F_1(Q) \triangleleft b \triangleright F_2(Q)) \uparrow \quad \{\text{rule (3)}\} \\
\Rightarrow & (s, F_1(Q)) \uparrow \triangleleft(s; b) \triangleright (s, F_2(Q)) \uparrow \quad \{\text{induction hypothesis}\} \\
\Rightarrow & (s, F_1(\perp)) \uparrow \triangleleft(s; b) \triangleright (s, F_2(\perp)) \uparrow \quad \{\text{rule (3)}\} \\
\Rightarrow & (s, (F_1(\perp) \triangleleft b \triangleright F_2(\perp))) \uparrow
\end{aligned}$$

$$\begin{aligned}
& (s, F_1(Q); F_2(Q)) \uparrow && \{\text{rule (4)}\} \\
\Rightarrow & (s, F_1(Q)) \uparrow \vee \\
& \exists t \bullet (s, F_1(Q)) \rightarrow^* (t, \epsilon) \wedge (t, F_2(Q)) \uparrow && \{\text{Lemma 3.1}\} \\
\Rightarrow & (s, F_1(\perp)) \uparrow \vee (s, F_1(\perp)) \rightarrow^* (t, \epsilon) \wedge (t, F_2(\perp)) \uparrow && \{\text{rule (4)}\} \\
\Rightarrow & (s, F_1(\perp); F_2(\perp)) \uparrow
\end{aligned}$$

**Proof** of (2): Similar to (1).

## References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge Press, Cambridge (1996)
2. Abrial, J.-R.: Modelling in Event-B: System and Software Engineering. Cambridge Press, Cambridge (2010)
3. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
4. Henner, E.C.R.: Predicative programming, Part 1, 2. Commun. ACM **27**(2), 134–151
5. Hennessy, M.C.: Algebraic Theory of Process. The MIT Press, Cambridge (1988)
6. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**, 576–583 (1969)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Upper Saddle River (1985)
8. Hoare, C.A.R., et al.: Laws of programming. Commun. ACM **30**(8), 672–686 (1987)
9. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall, Englewood Cliffs (1998)
10. Jones, C.B.: Systematic Software Development Using VDM. Prentice Hall, Englewood Cliffs (1986)
11. Milner, R.: Communicating and Mobile Systems: The  $\pi$ -Calculus. Cambridge Univ. Press, Cambridge (1999)
12. G.D. Plotkin. A structural approach to operational semantics. Technical report, DAIMI-FN-19, Aarhus University, Denmark, (1981)
13. Roscoe, A.W.: Laws of occam programming. Theoret. Comput. Sci. **60**, 177–229 (1988)
14. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1998)
15. Spivey, J.M., Notation, T.Z.: A Reference Manual. Prentice Hall, Englewood Cliffs (1992)

Unifying Theories of Programming

6th International Symposium, UTP 2016, Reykjavik,  
Iceland, June 4-5, 2016, Revised Selected Papers

Bowen, J.P.; Zhu, H. (Eds.)

2017, IX, 217 p. 36 illus., Softcover

ISBN: 978-3-319-52227-2