程序验证方法

研究生课程

Chapter 7 (7.1, 7.2)
Disjoint Parallel Programs

朱惠彪

华东师范大学 软件工程学院

Syntax

In this part of the book we study parallel programs, and in this chapter we investigate disjoint parallelism, the simplest form of parallelism.

Disjointness means here that the component programs have only reading access to common variables.

Two **while** programs S_1 and S_2 are called *disjoint* if neither of them can change the variables accessed by the other one; that is, if

$$change(S_1) \cap var(S_2) = \emptyset$$

and

$$var(S_1) \cap change(S_2) = \emptyset.$$

Syntax

change(S) is the set of simple and array variables of S that can be modified by it; that is, to which a value is assigned within S by means of an assignment.
Note that disjoint programs are allowed to read the same variables.

Example 7.1. The programs $x := z$ and $y := z$ are disjoint because $change(x := z) = \{x\}$, $var(y := z) = \{y, z\}$ and $var(x := z) = \{x, z\}$, $change(y := z) = \{y\}$.

On the other hand, the programs $x := z$ and $y := x$ are not disjoint because $x \in change(x := z) \cap var(y := x)$, and the programs $a[1] := z$ and $y := a[2]$ are not disjoint because $a \in change(a[1] := z) \cap var(y := a[2])$. $\square$

Syntax

Disjoint parallel programs are generated by the $\boxed{\text{same clauses}}$ as those defining **while** programs in Chapter 3 together with the following clause for *disjoint parallel composition*:

$$S ::= [S_1\|\ldots\|S_n],$$

where for $n > 1$, $S_1, \ldots, S_n$ are pairwise disjoint **while** programs, called the (*sequential*) *components* of S. Thus we do not allow nested parallelism, but we allow parallelism to occur within sequential composition, conditional statements and **while** loops.

Chapter 3 in p57

A **while** *program* is a string of symbols including the keywords **if**, **then**, **else**, **fi**, **while**, **do** and **od**, that is generated by the following grammar:

$$S ::= skip \mid u := t \mid S_1;\ S_2 \mid \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } B \textbf{ do } S_1 \textbf{ od}.$$

Syntax

It is useful to extend the notion of disjointness to expressions and assertions. An expression t and a program S are called *disjoint* if S cannot change the variables of t; that is, if

$$change(S) \cap var(t) = \emptyset.$$

Similarly, an assertion p and a program S are called *disjoint* if S cannot change the variables of p; that is, if

$$change(S) \cap var(p) = \emptyset.$$

Syntax

- Under what conditions can parallel execution be reduced to a sequential execution?
- In other words, is there any simple syntactic criterion that guarantees that all computations of a parallel program are equivalent to the sequential execution of its components?

Semantics

We now define semantics of disjoint parallel programs in terms of transitions. Intuitively, a disjoint parallel program $[S_1\|\ldots\|S_n]$ performs a transition if one of its components performs a transition. This form of modeling concurrency is called *interleaving*. Formally, we expand the transition system for **while** programs by the following transition rule

$$\text{(xvii)}\quad \frac{< S_i, \sigma > \ \rightarrow \ < T_i, \tau >}{< [S_1\|\ldots\|S_i\|\ldots\|S_n], \sigma > \ \rightarrow \ < [S_1\|\ldots\|T_i\|\ldots\|S_n], \tau >}$$

where $i \in \{1, \ldots, n\}$.

For WHILE program in p58

We choose here a "high level" view of an execution, where a configuration is simply a pair $< S, \sigma >$ consisting of a program S and a state σ. Intuitively, a *transition*

$$< S, \sigma > \ \rightarrow \ < R, \tau > \tag{3.1}$$

Semantics

Computations of disjoint parallel programs are defined like those of sequential programs. For example,

$$
\begin{aligned}
& < [x := 1 \| y := 2 \| z := 3], \sigma > \\
\to\ & < [E \| y := 2 \| z := 3], \sigma[x := 1] > \\
\to\ & < [E \| E \| z := 3], \sigma[x := 1][y := 2] > \\
\to\ & < [E \| E \| E], \sigma[x := 1][y := 2][z := 3] >
\end{aligned}
$$

is a computation of $[x := 1 \| y := 2 \| z := 3]$ starting in σ.

For WHILE program in p59

(iii) A computation of S *is terminating in* τ (or *terminates in* τ) if it is finite and its last configuration is of the form $< E, \tau >$.

This identification allows us to maintain the definition of a terminating computation given in Definition 3.1. For example, the final configuration in the above computation is the terminating configuration

$$< E, \sigma[x := 1][y := 2][z := 3] > .$$

Semantics

Lemma 7.1. (Absence of Blocking) *Every configuration* $< S, \sigma >$ *with* $S \not\equiv E$ *and a proper state* σ *has a successor configuration in the transition relation* $\rightarrow$ *.*

Thus when started in a state σ a disjoint parallel program $S \equiv [S_1 \| \ldots \| S_n]$ terminates or diverges. Therefore we introduce two types of input/output semantics for disjoint programs in just the same way as for **while** programs.

Definition 7.1. For a disjoint parallel program S and a proper state σ

(i) the *partial correctness semantics* is a mapping

$$\mathcal{M}[\![S]\!] : \Sigma \rightarrow \mathcal{P}(\Sigma)$$

with

$$\mathcal{M}[\![S]\!](\sigma) = \{\tau \mid < S, \sigma > \rightarrow^* < E, \tau >\}$$

(ii) and the *total correctness semantics* is a mapping

$$\mathcal{M}_{tot}[\![S]\!] : \Sigma \rightarrow \mathcal{P}(\Sigma \cup \{\perp\})$$

with

$$\mathcal{M}_{tot}[\![S]\!](\sigma) = \mathcal{M}[\![S]\!](\sigma) \cup \{\perp \mid S \text{ can diverge from } \sigma\}.$$

Recall that $\perp$ is the error state standing for divergence. □

Semantics Determinism

Unlike **while** programs, disjoint parallel programs can generate more than one computation starting in a given initial state. Thus determinism in the sense of the Determinism Lemma 3.1 does not hold. However, we can prove that all computations of a disjoint parallel program starting in the same initial state produce the same output. Thus a weaker form of determinism holds here, in that for every disjoint parallel program S and proper state σ, $\mathcal{M}_{tot}[\![S]\!](\sigma)$ has exactly one element, either a proper state or the error state $\perp$. This turns out to be a simple corollary to some results concerning properties of abstract *reduction systems*.

Chapter 3 in p60

Lemma 3.1. (Determinism) *For any* **while** *program S and a proper state σ, there is exactly one computation of S starting in σ.*

Semantics Determinism

Definition 7.2. A *reduction system* is a pair $(A, \rightarrow)$ where A is a set and $\rightarrow$ is a binary relation on A; that is, $\rightarrow \subseteq A \times A$. If $a \rightarrow b$ holds, we say that a can be *replaced* by b. Let $\rightarrow^*$ denote the transitive reflexive closure of $\rightarrow$.

We say that $\rightarrow$ satisfies the *diamond property* if for all $a, b, c \in A$ with $b \neq c$

$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ b & & c \end{tikzcd}$$

implies that for some $d \in A$

$$\begin{tikzcd} b \arrow[dr] & & c \arrow[dl] \\ & d. & \end{tikzcd}$$

$\rightarrow$ is called *confluent* if for all $a, b, c \in A$

$$\begin{tikzcd} & a \arrow[dl, "*"'] \arrow[dr, "*"] & \\ b & & c \end{tikzcd}$$

implies that for some $d \in A$

$$\begin{tikzcd} b \arrow[dr, "*"'] & & c \arrow[dl, "*"] \\ & d. & \end{tikzcd}$$

□

Lemma 7.2. (Confluence) *For all reduction systems $(A, \rightarrow)$ the following holds: if a relation $\rightarrow$ satisfies the diamond property then it is confluent.*

Proof. Suppose that $\rightarrow$ satisfies the diamond property. Let $\rightarrow^n$ stand for the n-fold composition of $\rightarrow$. A straightforward proof by induction on $n \geq 0$ shows that $a \rightarrow b$ and $a \rightarrow^n c$ implies that for some $i \leq n$ and some $d \in A$, $b \rightarrow^i d$ and $c \rightarrow^\epsilon d$. Here $c \rightarrow^\epsilon d$ iff $c \rightarrow d$ or $c = d$. Thus $a \rightarrow b$ and $a \rightarrow^* c$ implies that for some $d \in A$, $b \rightarrow^* d$ and $c \rightarrow^* d$.

This implies by induction on $n \geq 0$ that if $a \rightarrow^* b$ and $a \rightarrow^n c$ then for some $d \in A$ we have $b \rightarrow^* d$ and $c \rightarrow^* d$. This proves confluence. $\square$

Semantics Determinism

Lemma 7.3. (Infinity) *Consider a reduction system* $(A, \rightarrow)$ *where* $\rightarrow$ *satisfies the diamond property and elements* $a, b, c \in A$ *with* $a \rightarrow b$, $a \rightarrow c$ *and* $b \neq c$. *If there exists an infinite sequence* $a \rightarrow b \rightarrow \ldots$ *passing through* b *then there exists also an infinite sequence* $a \rightarrow c \rightarrow \ldots$ *passing through* c.

Proof. Consider an infinite sequence $a_0 \rightarrow a_1 \rightarrow \ldots$ where $a_0 = a$ and $a_1 = b$.

Case 1. For some $i \geq 0$, $c \rightarrow^* a_i$.
Then $a \rightarrow c \rightarrow^* a_i \rightarrow \ldots$ is the desired sequence.

Case 2. For no $i \geq 0$, $c \rightarrow^* a_i$.
We construct by induction on i an infinite sequence $c_0 \rightarrow c_1 \rightarrow \ldots$ such that $c_0 = c$ and for all $i \geq 0$ $a_i \rightarrow c_i$. c_0 is already correctly defined. For $i = 1$ note that $a_0 \rightarrow a_1$, $a_0 \rightarrow c_0$ and $a_1 \neq c_0$. Thus by the diamond property there exists a c_1 such that $a_1 \rightarrow c_1$ and $c_0 \rightarrow c_1$.

Consider now the induction step. We have $a_i \rightarrow a_{i+1}$ and $a_i \rightarrow c_i$ for some $i > 0$. Also, since $c \rightarrow^* c_i$, by the assumption $c_i \neq a_{i+1}$. Again by the diamond property for some c_{i+1}, $a_{i+1} \rightarrow c_{i+1}$ and $c_i \rightarrow c_{i+1}$. $\square$

Semantics Determinism

Definition 7.3. Let $(A, \rightarrow)$ be a reduction system and $a \in A$. An element $b \in A$ is $\rightarrow$-maximal if there is no c with $b \rightarrow c$. We define now

$$
\begin{aligned}
yield(a) = \quad & \{b \mid a \rightarrow^* b \text{ and } b \text{ is } \rightarrow\text{-maximal}\} \\
\cup \; & \{\perp \mid \text{ there exists an infinite sequence } a \rightarrow a_1 \rightarrow \ldots\}
\end{aligned}
$$

$\square$

Semantics Determinism

Lemma 7.4. (Yield) *Let* $(A, \rightarrow)$ *be a reduction system where* $\rightarrow$ *satisfies the diamond property. Then for every* a*,* $yield(a)$ *has exactly one element.*

Proof. Suppose that for some $\rightarrow$-maximal b and c, $a \rightarrow^* b$ and $a \rightarrow^* c$. By Confluence Lemma 7.2, there is some $d \in A$ with $b \rightarrow^* d$ and $c \rightarrow^* d$. By the $\rightarrow$-maximality of b and c, both $b = d$ and $c = d$; thus $b = c$.

Thus the set $\{b \mid a \rightarrow^* b, b \text{ is } \rightarrow\text{-maximal}\}$ has at most one element. Suppose it is empty. Then $yield(a) = \{\bot\}$.

Semantics Determinism

Lemma 7.5. (Diamond) *Let S be a disjoint parallel program and σ a proper state. Whenever*

$$\begin{array}{ccc} & < S, \sigma > & \\ \swarrow & & \searrow \\ < S_1, \sigma_1 > & \neq & < S_2, \sigma_2 >, \end{array}$$

then for some configuration $< T, \tau >$

$$\begin{array}{ccc} < S_1, \sigma_1 > & & < S_2, \sigma_2 > \\ & \searrow \quad \swarrow & \\ & < T, \tau >. & \end{array}$$

Proof. By the Determinism Lemma 3.1 and the interleaving transition rule (viii), the program S is of the form $[T_1 \| \ldots \| T_n]$ where $T_1, \ldots, T_n$ are pairwise disjoint **while** programs, and S_1 and S_2 result from S by transitions of two of these **while** programs, some T_i and T_j, with $i \neq j$. More precisely, for some **while** programs T_i' and T_j'

Semantics Determinism

$$S_1 = [T_1\|\dots\|T_i'\|\dots\|T_n],$$
$$S_2 = [T_1\|\dots\|T_j'\|\dots\|T_n],$$
$$< T_i, \sigma > \ \rightarrow \ < T_i', \sigma_1 >,$$
$$< T_j, \sigma > \ \rightarrow \ < T_j', \sigma_2 > .$$

Define T and τ as follows:

$$T = [T_1'\|\dots\|T_n'],$$

where for $k \in \{1, \dots, n\}$ with $k \neq i$ and $k \neq j$

$$T_k' = T_k$$

and for any variable u

$$\tau(u) = \begin{cases} \sigma_1(u) \text{ if } & u \in \mathit{change}(T_i), \\ \sigma_2(u) \text{ if } & u \in \mathit{change}(T_j), \\ \sigma(u) \ \text{ otherwise.} & \end{cases}$$

By disjointness of T_i and T_j, the state τ is well defined. Using the Change and Access Lemma 3.4 it is easy to check that both $< S_1, \sigma_1 > \ \rightarrow \ < T, \tau >$ and $< S_2, \sigma_2 > \ \rightarrow \ < T, \tau >$. $\square$

Lemma 3.4

Lemma 3.4. (Change and Access)

(i) For all proper states σ and τ, $\tau \in \mathcal{N}[\![S]\!](\sigma)$ implies

$$\tau[Var - change(S)] = \sigma[Var - change(S)].$$

(ii) For all proper states σ and τ, $\sigma[var(S)] = \tau[var(S)]$ implies

$$\mathcal{N}[\![S]\!](\sigma) = \mathcal{N}[\![S]\!](\tau) \ \mathbf{mod}\ Var - var(S).$$

Proof. See Exercise 3.2. □

Recall that *Var* stands for the set of all simple and array variables. Part (i) of the Change and Access Lemma states that every program S changes at most the variables in $change(S)$, while part (ii) states that every program S accesses at most the variables in $var(S)$. This explains the name of this lemma. It is used often in the sequel.

Semantics Determinism

Lemma 7.6. (Determinism) *For every disjoint parallel program S and proper state σ, $\mathcal{M}_{tot}[\![S]\!](\sigma)$ has exactly one element.*

Proof. By Lemmata 7.4 and 7.5 and observing that for every proper state σ, $\mathcal{M}_{tot}[\![S]\!](\sigma) = yield(< S, \sigma >)$. $\square$

Semantics Sequentialization

- The Determinism Lemma helps us provide a quick proof that disjoint parallelism reduces to sequential composition.
- To relate the computations of sequential and parallel programs, we use the following general notion of equivalence.

Definition 7.4. Two computations are *input/output equivalent*, or simply *i/o equivalent*, if they start in the same state and are either both infinite or both finite and then yield the same final state. In later chapters we also consider error states such as **fail** or Δ among the final states. $\square$

Lemma 7.7. (Sequentialization) *Let* $S_1, \ldots, S_n$ *be pairwise disjoint* **while** *programs. Then*

$$\mathcal{M}[\![[S_1\|\ldots\|S_n]]\!] = \mathcal{M}[\![S_1;\ \ldots;\ S_n]\!],$$

and

$$\mathcal{M}_{tot}[\![[S_1\|\ldots\|S_n]]\!] = \mathcal{M}_{tot}[\![S_1;\ \ldots;\ S_n]\!].$$

Proof. We call a computation of $[S_1\|\ldots\|S_n]$ *sequentialized* if the components $S_1, \ldots, S_n$ are activated in a sequential order: first execute exclusively S_1, then, in case of termination of S_1, execute exclusively S_2, and so forth.

We claim that every computation of $S_1;\ \ldots;\ S_n$ is i/o equivalent to a sequentialized computation of $[S_1\|\ldots\|S_n]$.

This claim follows immediately from the observation that the computations of $S_1;\ \ldots;\ S_n$ are in a one-to-one correspondence with the sequentialized computations of $[S_1\|\ldots\|S_n]$. Indeed, by replacing in a computation of $S_1;\ \ldots;\ S_n$ each configuration of the form

$$< T;\ S_{k+1};\ \ldots;\ S_n, \tau >$$

Semantics Sequentialization

by

$$< [E\|\ldots\|E\|T\|S_{k+1}\|\ldots\|S_n], \tau >$$

we obtain a sequentialized computation of $[S_1\|\ldots\|S_n]$. Conversely, in a sequentialized computation of $[S_1\|\ldots\|S_n]$ each configuration is of the latter form, so by applying to such a computation the above replacement operation in the reverse direction, we obtain a computation of $S_1;\ \ldots;\ S_n$.

This claim implies that for every state σ

$$\mathcal{M}_{tot}[\![S_1;\ \ldots;\ S_n]\!](\sigma) \subseteq \mathcal{M}_{tot}[\![[S_1\|\ldots\|S_n]]\!](\sigma).$$

By the Determinism Lemmata 3.1 and 7.6, both sides of the above inclusion have exactly one element. Thus in fact equality holds. This also implies

$$\mathcal{M}[\![S_1;\ \ldots;\ S_n]\!](\sigma) = \mathcal{M}[\![[S_1\|\ldots\|S_n]]\!](\sigma)$$

and completes the proof of the lemma. $\square$