形 式 化 方 法:

基于 B 方法的严格软件开发

(6) 非确定性和其他

裘宗燕

北京大学数学学院信息科学系

2010年春季

SELECT 代换（《B Book》5.1.1节）

CHOICE 结构可以有任意多个选择分支，要执行这个结构，任何一个分支都可能被选。这种结构对于分支的选择没有任何控制

实际中我们可能希望有多种选择，但又希望对实际选择有一定控制，希望在某些状态下只能选（或不能选）某个或某些分支

给分支增加控制的方法是提供一个允许（enable）条件，只有被允许的分支才是可能被选的分支。B 语言的相应结构是 **SELECT**

SELECT P_1 **THEN** S_1
WHEN P_2 **THEN** S_2
WHEN P_3 **THEN** S_3
... ...
ELSE V
END

这里的选择条件 $P_1, P_2, P_3, \ldots$ 称为 “卫式条件” 或者 “卫”

注意，**SELECT** 并不等价于一系列嵌套的 **IF**

IF P_1 **THEN** S_1 **ELSE IF** P_2 **THEN** S_2**ELSE IF** P_3 **THEN** S_3 **ELSE ...**
... END END END

SELECT 代换

关于 **SELECT** 结构的意义有如下规定

- 一个 **SELECT** 结构里的不同卫式条件并不要求互相不重叠
- 只有允许的分支才能执行，只有一个允许分支最终被执行
- 如果多个分支被允许，在这些分支中的选择是非确定性的。（**SELECT** 结构中各分支的书写顺序对分支的选择没有影响）
- 如果所有带有卫分支的条件都不成立，且该 **SELECT** 结构有 **ELSE** 分支，则执行 **ELSE** 分支的代换；如果此时没有 **ELSE** 分支，结果无定义

假定棋子可在 8×8 棋盘里上下左右移动，但不能移出棋盘。一个棋子一次移动导致的可能状态变化可描述如下（(x, y) 表示棋子当时位置）

$$
\begin{array}{llll}
\textbf{SELECT} & x > 1 & \textbf{THEN} & x := x - 1 \\
\textbf{WHEN} & x < 8 & \textbf{THEN} & x := x + 1 \\
\textbf{WHEN} & y > 1 & \textbf{THEN} & y := y - 1 \\
\textbf{WHEN} & y < 8 & \textbf{THEN} & y := y + 1 \\
\textbf{END} & & &
\end{array}
$$

显然，任何状态至少使这里两个分支得到允许，它们之间的选择是非确定的

SELECT

对于 **SELECT** 代换，要保证其后条件成立，就要保证无论哪个分支得到允许，代换后相应的后条件都能成立。这样就得到

$$
\left[\begin{array}{llll}
\textbf{SELECT} & P_1 & \textbf{THEN} & S_1 \\
\textbf{WHEN} & P_2 & \textbf{THEN} & S_2 \\
\textbf{WHEN} & P_3 & \textbf{THEN} & S_3 \\
\dots & \dots & & \\
\textbf{ELSE}\ V & & & \\
\textbf{END} & & &
\end{array}\right] Q = \begin{array}{ll}
 & (P_1 \Rightarrow [S_1]Q) \\
\wedge & (P_2 \Rightarrow [S_2]Q) \\
\wedge & (P_3 \Rightarrow [S_3]Q) \\
\dots & \dots \\
\wedge & (\neg(P_1 \vee P_2 \vee P_3 \vee \dots) \Rightarrow [V]Q)
\end{array}
$$

如果 **SELECT** 没有 **ELSE** 分支，逻辑公式就没有最后的合取分支

如果满足一个 **SELECT** 代换各个卫的状态集合互不相交（对任一状态，至多有一个卫成立），那么该 **SELECT** 代换的行为就是确定性的

如果这些集合互不相交，而且它们的并是整个状态空间（或者有 **ELSE**分支），这些分支就形成了对状态空间的一个划分

SELECT

考虑前面例子，证明移动棋子后 $x < 4$

$$\left[\begin{array}{llll} \textsf{SELECT} & x > 1 & \textsf{THEN} & x := x - 1 \\ \textsf{WHEN} & x < 8 & \textsf{THEN} & x := x + 1 \\ \textsf{WHEN} & y > 1 & \textsf{THEN} & y := y - 1 \\ \textsf{WHEN} & y < 8 & \textsf{THEN} & y := y + 1 \\ \textsf{END} & & & \end{array}\right](x < 4)$$

$$= \begin{array}{ll} & (x > 1 \Rightarrow [x := x - 1](x < 4)) \\ \wedge & (x < 8 \Rightarrow [x := x + 1](x < 4)) \\ \wedge & (y > 1 \Rightarrow [y := y - 1](x < 4)) \\ \wedge & (y < 8 \Rightarrow [y := y + 1](x < 4)) \end{array} = \begin{array}{ll} & (x > 1 \Rightarrow x < 5) \\ \wedge & (x < 8 \Rightarrow x < 3) \\ \wedge & (y > 1 \Rightarrow x < 4) \\ \wedge & (y < 8 \Rightarrow x < 4) \end{array}$$

$$= \quad I \wedge x < 5$$

这里的 I 是棋子棋盘关系的不变式

IF, CHOICE 和 SELECT

如果一个 **SELECT** 的所有分支的卫都是逻辑公式 **TRUE**，得到的就是这些分支之间的非确定性选择，与 **CHOICE** 的行为完全一样

所以，可以把 **CHOICE** 结构看作 **SELECT** 结构的一种特殊情况

另一方面，**IF**结构也就是一种确定性的 **SELECT**

$$\textsf{IF } P \textsf{ THEN } S_1 \textsf{ ELSE } S_2 \textsf{ END}$$

等价于

$$\begin{array}{l} \textsf{SELECT } P \textsf{ THEN } S_1 \\ \textsf{WHEN } \neg P \textsf{ THEN } S_2 \textsf{ END} \end{array} \quad \text{或者} \quad \begin{array}{l} \textsf{SELECT } P \textsf{ THEN } S_1 \\ \textsf{ELSE } S_2 \textsf{ END} \end{array}$$

可见，**SELECT** 是一种最一般性的选择结构

IF 结构的扩充形式

IF 提供了在一些编程语言可以看到的 **ELSIF** 简写形式

IF P_1 **THEN** S_1
ELSIF P_2 **THEN** S_2
...
ELSE S_n **END**
	相当于	**IF** P_1 **THEN** S_1
		ELSE IF P_2 **THEN** S_2
		  ELSE IF ...
		    ...
		      ELSE S_n
		      END
		    ...
		  END
		END

如果没有最后的 **ELSE** 部分，也相当于这里有一个 skip 代换

这种扩充使人可以比较方便地描述一系列条件，而且不需要一层层退格，视觉上比较清晰

CASE 结构

最后一种选择结构是 **CASE** 代换，形式上类似于常见编程语言里的多分支 **CASE** 语句。在 B 里它也是一类特殊的 **SELECT** 结构

CASE E **OF** **EITHER** l_1 **THEN** S_1 **OR** l_2 **THEN** S_2 **OR** l_3 **THEN** S_3 ... **ELSE** S_n **END**	相当于	**SELECT** $E \in \{l_1\}$ **THEN** S_1 **WHEN** $E \in \{l_2\}$ **THEN** S_2 **WHEN** $E \in \{l_3\}$ **THEN** S_3 ... **ELSE** S_n **END**

这里要求 E 是表达式，$l_1, \ldots,$ 是一些互不相同的标识符或者数值文字量

CASE 结构是一种确定性结构，它就是根据表达式 E 在当前状态中的取值确定不同的分支代换

实例：故事盒

假定我们要设计一种故事盒，作为儿童教育的辅助设备

- 盒里保存了一批小故事，可以自动播放
- 有两个家长特权按钮，另有两个供儿童选择故事和播放的按钮
- 一个家长按钮用于给儿童增加积分，一个扣除积分（奖惩）
- 一个儿童按钮用于选择故事，选入一个待讲列表；另一个儿童按钮用于启动讲故事。讲了的故事从待讲表删除
- 选故事将使用积分，但也可能不扣积分奖励一个故事

MACHINE $StBox$
SETS $Story$
CONSTANTS max_scores
PROPERTIES $max\_scores \in \mathbb{N}_1$
VARIABLES $scores, slist$
INVARIANT
  $scores \in \mathbb{N} \wedge scores \leq max\_scores \wedge slist : \mathbb{P}(Story)$
INITIALISATION $scores := 0 \mid\mid slist := \{\}$

实例：故事盒

$gives(sc) =$
PRE $sc \in \mathbb{N}_1$
THEN
  $scores := \mathrm{min}(scores + sc, max\_scores)$
END;

$penalty =$
SELECT
  $scores > 0$ **THEN** $scores := scores - 1$
WHEN $slist \neq \{\}$ **THEN**
  ANY st
  WHERE $st \in slist$
  THEN $slist := slist - \{st\}$
  END
ELSE
  skip
END;

实例：故事盒

$select(st) =$
PRE $scores > 0 \wedge st \in Story$
THEN
  CHOICE $scores := scores - 1 \;||\; slist := slist \cup \{st\}$
  OR $slist := slist \cup \{st\}$
  END
END;

$st \longleftarrow tell =$
IF $slist \neq \{\}$
THEN
  ANY $st1$ WHERE $st1 \in slist$
  THEN
    $st := st1 \;||\; slist := slist - \{st1\}$
  END
END
END

: () 代换

有时我们可能希望某变量 x 取得一个任意的满足谓词 P 的值，显然这可以用下面代换表示

$$x :\in \{y | P(y)\}$$

B 语言为此代换提供了一种专门形式

$$x : (P)$$

可以认为这只是一种简写形式

显然这里的 P 应该是一个牵涉到 x 的谓词，其中应该给出 x 的类型，还可以给出其他限制。例如：

$$x : (x \in \mathbb{N} \wedge x \bmod 2 = 0)$$

$$y : (y \in \mathbb{Z} \wedge -3 \leq y \wedge y < 8)$$

: () 代换称为 “become such that” 代换。: 前面可以写多个变量。如

$$x, y : (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x + y < 5)$$

ASSERT 代换

ASSERT 代换引进新的局部条件断言，其形式是：

$$\textbf{ASSERT } P \textbf{ THEN } S \textbf{ END}$$

这里的 P 是谓词，S 是一个代换

要保证 **ASSERT** 代换之后谓词 Q 成立，最弱前条件定义是

$$[\textbf{ASSERT } P \textbf{ THEN } S \textbf{ END}]Q \ \widehat{=}\ P \wedge (P \Rightarrow [S]Q)$$

根据这个定义，假如我们要证明描述机器状态的谓词 R 能保证经过上面 **ASSERT** 代换后 Q 成立，那么就需要证明

$$R \Rightarrow P \wedge (P \Rightarrow [S]Q)$$

这等价于要求证明

$$R \Rightarrow P \quad \text{以及} \quad R \Rightarrow (P \Rightarrow [S]Q)$$

第二个逻辑式等价于

$$R \wedge P \Rightarrow [S]Q$$

顺序代换

B 语言也支持写顺序代换

$$S_1 \ ; \ S_2$$

这表示由顺序进行的两个代换合成的代换

要保证一个顺序代换后谓词 Q 成立，其最弱前条件是：

$$[S_1 \ ; \ S_2]Q \ \widehat{=}\ [S_1][S_2]Q$$

代换作为一元操作是右结合的，上式右边等价于 $[S_1]([S_2]Q)$

根据定义，不难确定代换顺序复合操作 “;” 满足结合律：

$$(S_1 \ ; \ S_2) \ ; \ S_3 \ = \ S_1 \ ; \ (S_2 \ ; \ S_3)$$

顺序代换的例子不讨论了

应该指出，如果代换之间没有必要的顺序关系，我们就不应该用顺序代换，而应该用 “同时代换”

局部变量代换

如果在描述代换时只是需要引进一个或几个局部变量，并不需要限定它们满足的约束条件，可以使用局部变量代换

$$\textsf{VAR}\ x_1, \ldots, x_n\ \textsf{IN}\ S\ \textsf{END}$$

其中 x_i $(i = 1, \ldots, n)$ 是一组互不相同的变量，S 是个代换

例如，用局部变量保存中间结果：

$$\textsf{VAR}\ x\ \textsf{IN}\ x := y + 1\ ;\ z := x * x\ \textsf{END}$$

至此，我们已经讨论了 B 方法里的广义代换（除一种之外）的全部代换形式。B 抽象机里的代换都基于这些形式写出，操作都基于这些代换定义。有关情况可以查阅语言手册或《B Book》一书的附录 C.12

数组（array）

数组是各种编程语言里的重要机制，用于组织程序里的数据

下面考虑数组的形式化理论，将数组看作一类函数。这样我们就可以在规范里定义数组，访问数组元素。还可以看到，也可以很自然地定义给数组元素赋值的操作

在编程语言里，一个数组（数组变量）有一个名字，包含一组元素。可以通过数组的名字和下标访问这些元素

访问数组元素的描述形式在不同语言里有所不同，基本上是两种形式

1. $a[i]$，Pascal、C 语言及其后继语言
2. $a(i)$，Fortran、Ada 等语言

n 个元素的数组和 n 个独立变量（同一类型）之间的差异，就在于对数组元素有一种统一的访问方式。通过下标表达式的不同取值，可以统一地访问数组里的一批元素（甚至全部元素）

数组：看作固定的有限定义域的函数

由于数组元素可以独立变化，一个 n 元素的数组 a 应该相当于一集 n 个变量，而给了数组名 a 一个具体下标，就得到这些变量里的一个

抽象看，a 是从数组下标集合到数组元素类型的一个有限函数，$a(i)$ 看作是数组 a 的第 i 个元素，也可看作是函数 a 作用于 i 的结果。下面数组

$$a_1 \;=\; \{1 \mapsto 3, 2 \mapsto 5, 3, \mapsto 7, 4 \mapsto 9\}$$

是个数组，其类型是 $1..4 \rightarrow \mathbb{N}$

B 将数组看作是有限函数，采用整数作为下标时总从 1 开始。即，类型总是 $1..n \rightarrowtail T$（对某 n，T 是数组元素类型。也允许以其他类型为下标）。由于允许数组对其中一些元素无定义，因此类型应是部分函数 $1..n \nrightarrow T$。由于数组是函数，各种函数操作都可以自然地用于数组

数组与序列的不同点在于数组的大小是固定的，也就是说，在数组的存在期间其大小不会改变。序列不是这样，其大小可变

虽然数组的类型是函数，但抽象机里定义的数组也是变量。作为变量的数组可以像其他变量一样使用，包括对整个数组做代换

数组：元素

在常规编程语言里，最基本的数组操作是元素的访问和修改（重新赋值）

在 B 里，元素访问是函数作用，修改元素是修改函数的定义，用覆盖描述。要给前面的数组 a_1 的下标为 3 元素重新赋值 11，得到的函数为

$$a_1 \Leftarrow\!\!\!+ \{3 \mapsto 11\}$$

要做这样的“元素赋值代换”，改变抽象机状态，就应该写

$$a_1 := a_1 \Leftarrow\!\!\!+ \{3 \mapsto 11\}$$

一般而言，修改数组 a 的第 i 个元素的操作写为

$$a := a \Leftarrow\!\!\!+ \{i \mapsto E\}$$

由于这种操作在规范里也很常用，B 语言提供了一种简写形式

$$a(i) := E$$

写规范不仅是为了描述抽象模型，更重要的是为了做推理和验证。从这种角度看，a 是变量而 $a(i)$ 不是变量，变换 $a(i) := E$ 实际改变的是整个数组（函数）a，但改变后的 a 除了对 i 的应用之外都没有变

数组：验证

数组“赋值”操作也是代换，其前条件是

$$[a(i) := E]Q \;=\; [a \mathbin{\lhd\!\!\!+} \{i \mapsto E\}/a]Q$$

即，将后条件谓词 Q 里的 a 换成覆盖后的函数。如果 Q 里有 $a(j)$，就应该代换为 $(a \mathbin{\lhd\!\!\!+} \{i \mapsto E\})(j)$

根据前面介绍过的覆盖的定义：

$$(a \mathbin{\lhd\!\!\!+} \{i \mapsto E\})(j) \;=\; \begin{cases} E & \text{如果 } i = j \\ a(j) & \text{否则} \end{cases}$$

下面是两个具体问题的推导

$$\begin{aligned} [a(2) := y + 3](a(2) = 5) &= (a \mathbin{\lhd\!\!\!+} \{2 \mapsto y + 3\})(2) = 5 \\ &= y + 3 = 5 \;=\; y = 2 \end{aligned}$$

$$\begin{aligned} [a(i) := 3](a(2) = 5) &= (a \mathbin{\lhd\!\!\!+} \{i \mapsto 3\})(2) = 5 \\ &= \begin{cases} 3 = 5 & \text{如果 } i = 2 \\ a(2) = 5 & \text{否则} \end{cases} \\ &= i \neq 2 \wedge a(2) = 5 \end{aligned}$$

数组

除了可以对数组的个别元素提出要求（后条件）外，也完全可以对整个数组提出要求。例如要求数组是个全函数，有如下推导

$$\begin{aligned} [a(i) := E](a \in 1..n \to T) &= (a \mathbin{\lhd\!\!\!+} \{i \mapsto E\}) \in 1..n \to T \\ &= i \in 1..n \wedge E \in T \wedge \\ &\quad\; (\{i\} \mathbin{\lhd\!\!\!-} a) \in (1..n - \{i\}) \to T \end{aligned}$$

也就是说，还要求代换前 a 是从 $(1..n - \{i\})$ 到 T 的全函数

这里是条件 $i \in 1..n$ 很重要，只有满足这个条件，$a(i) := E$ 才有意义

如果值要求是一个数组，也就是说是 $1..n$ 到某个 T 的偏函数，其前条件也只要求是偏函数

$$\begin{aligned} [a(i) := E](a \in 1..n \mathrel{+\!\!\!\to} T) &= (a \mathbin{\lhd\!\!\!+} \{i \mapsto E\}) \in 1..n \mathrel{+\!\!\!\to} T \\ &= i \in 1..n \wedge E \in T \wedge \\ &\quad\; (\{i\} \mathbin{\lhd\!\!\!-} a) \in (1..n - \{i\}) \mathrel{+\!\!\!\to} T \\ &= i \in 1..n \wedge E \in T \wedge a \in 1..n \mathrel{+\!\!\!\to} T \end{aligned}$$

数组

如果抽象机的不变式提出了更多要求，对于数组元素“赋值”就可能生成很强的前条件。例如要求数组里不包含重复元素可以描述为

$$a \in 1..n \rightarrowtail\!\!\!\!\twoheadrightarrow T$$

如果要求在代换 $a(i) := E$ 后这个条件成立，就要求

- i 在 $1..n$ 范围内且 $E \in T$
- a 在其定义域不包括 i 时是个片内射（单值）
- a 对于其去掉 i 的定义域中任何值都不映射到 E

相应的前条件可以推导出来：

$$\begin{aligned}
[a(i) := E](a \in 1..n \rightarrowtail\!\!\!\!\twoheadrightarrow T) &= (a \Lleftarrow \{i \mapsto E\}) \in 1..n \rightarrowtail\!\!\!\!\twoheadrightarrow T \\
&= i \in 1..n \wedge E \in T \wedge \\
&\quad\; (\{i\} \lhd\!\!\!- a) \in (1..n - \{i\}) \rightarrowtail\!\!\!\!\twoheadrightarrow T \wedge \\
&\quad\; E \notin a[1..n - \{i\}]
\end{aligned}$$

这一前条件包含很多内容

数组

考虑更复杂的谓词，例如：

$$\Sigma\, i.(i \in 1..n \mid a(i)) \leq imax$$

下面考虑对上述谓词做代换 $a(j) := 4$ 的前条件：

$$\begin{aligned}
&[a(j) := 4](\Sigma\, i.(i \in 1..n \mid a(i)) \leq imax) \\
= \;& j \in 1..n \wedge \Sigma\, i.(i \in 1..n \mid (a \Lleftarrow \{j \mapsto 4\})(i)) \leq imax \\
= \;& j \in 1..n \wedge \Sigma\, i.(i \in 1..n - \{j\} \mid a(i)) + \Sigma\, i.(i \in \{j\} \mid 4) \leq imax \\
= \;& j \in 1..n \wedge \Sigma\, i.(i \in 1..n - \{j\} \mid a(i)) + 4 \leq imax
\end{aligned}$$

现在考虑如何给整个数组的所有元素赋初值

假定 $a \in 1..n \twoheadrightarrow\!\!\!\!\!\!+\;\; \mathbb{N}$，下面代换将 a 的所有元素置为无定义

$$a := \{\}$$

下面代换将 a 的所有元素置为 0

$$a := 1..n \times \{0\}$$

数组

对于 $a \in 1..n \nrightarrow T$，B 里不允许写 $a(i), a(j) := E, F$，因为 $a(i) := E$ 是函数覆盖的简写形式。因此上式左边的两项都是给 a 赋值，违背了 B 语言多重代换里同一个变量不能出现两次的规定

回归函数覆盖的本来面貌，这个赋值应写为

$$a := a \Leftarrow \{i \mapsto E, j \mapsto F\}$$

这一表达式要合法，也有一些条件：

- $i \in 1..n$ 而且 $j \in 1..n$
- $i \neq j$，或者 $E = F$

第一条保证 a 的类型，第二条保证 $\Leftarrow$ 右边是函数，覆盖后 a 还是函数

同时赋值的一种特殊情况是交换两个元素的值

$$a := \{i \mapsto a(j), j \mapsto a(i)\}$$

只要两个下标都在合法范围内，这一操作就合法

实例：旅馆管理

考虑一个旅馆，它有一批房间，其中

- 小房间（标准间）可住 1-2 人
- 大房间（例如套间）可住 1-4 人

考虑用集合表示房间，用数组记录房间居住人数（从房间到人数的函数）

MACHINE *Hotel*
SETS *Room*
CONSTANTS *small*
PROPERTIES
  $small \subseteq Room$
VARIABLES
  $numbers$
INVARIANT
  $numbers \in Room \rightarrow 0..4 \wedge numbers[small] \subseteq 0..2$
INITIALISATION
  $numbers := Room \times 0$

实例：旅馆管理

由于经常需要考虑一种房间是否有空的，给出如下定义：

DEFINITIONS

$haveRoom(rms)$ == $\mathsf{max}(numbers[rms]) > \mathbf{0}$

操作 $checkin$ 获得一个空的能容纳所需人数的房间。这个操作比较复杂，有复杂的前条件和代换结构：

OPERATIONS

$rm \longleftarrow checkin(nn) =$
PRE $nn \in \mathbf{1..4} \wedge (nn \leq \mathbf{2} \Rightarrow haveRoom(Room)) \wedge$
$\quad (nn > \mathbf{2} \Rightarrow haveRoom(Room - small))$
THEN IF $nn \leq \mathbf{2}$
$\quad$ **THEN ANY** $rm_\mathbf{0}$ **WHERE** $rm_\mathbf{0} \in Room \wedge numbers(rm_\mathbf{0}) = \mathbf{0}$
$\quad\quad$ **THEN** $rm := rm_\mathbf{0} \mid\mid numbers(rm_\mathbf{0}) := nn$ **END**
$\quad$ **ELSE ANY** $rm_\mathbf{0}$ **WHERE** $rm_\mathbf{0} \in Room - small \wedge numbers(rm) = \mathbf{0}$
$\quad\quad$ **THEN** $rm := rm_\mathbf{0} \mid\mid numbers(rm_\mathbf{0}) := nn$ **END**
$\quad$ **END**
END;

实例：旅馆管理

$checkout(rm) =$
PRE
$\quad rm \in Room \wedge numbers(rm) \neq \mathbf{0}$
THEN
$\quad numbers(rm) := \mathbf{0}$
END;

$nn \longleftarrow roomquery(rm) =$
PRE
$\quad rm \in Room$
THEN
$\quad nn := numbers(rm)$
END;

$nn \longleftarrow vacancies = nn := \mathsf{card}(numbers \rhd \mathbf{0});$

$nn \longleftarrow totalguests = nn := \mathbf{\Sigma}\, xx.(xx \in Room \mid numbers(xx));$

实例：旅馆管理

现在考虑一个交换房间的操作，如果：

- 一个大房间入住人数不超过2人
- 一个小房间空闲

可以考虑将大房间入住的旅客调换到小房间

$swap(rm_1, rm_2) =$
  PRE $rm_1 \in Room \wedge rm_2 : Room$
  THEN IF
    $(rm_1 \in small \wedge rm_2 \notin small \wedge numbers(rm_1) = 0 \wedge$
     $numbers(rm_2) \in 1..2$
    $\vee$
     $rm1 \notin small \wedge rm_2 \in small \wedge numbers(rm_2) = 0 \wedge$
     $numbers(rm_1) \in 1..2 \;)$
  THEN
    $numbers := numbers \lhd\!\!- \{rm_1 \mapsto numbers(rm_2), rm_2 \mapsto numbers(rm_1)\}$
  END
END

总结

这两次课主要讨论了非确定性问题，也介绍了另外一些代换机制，还介绍了数组的概念

非确定性代换包括：

- **SELECT**
- **CHOICE**
- **ANY**
- $:\in$ 和 $:()$

非确定性在规范描述中扮演着重要角色

本节还介绍了

- **LET**
- **VAR**
- **ASSERT**
- 等等