形 式 化 方 法:

基于 B 方法的严格软件开发

(4) 代换、抽象机和证明义务

裘宗燕

北京大学数学学院信息科学系

2010年春季

规范、抽象机和广义代换

B 方法的基本想法是提供一套描述软件的方法，用它描述的软件最终可以用过程性的高级语言实现（如用 C、Ada等等），甚至用汇编语言

这一实现可能自动生成或人工写出。 B 方法的主要威力在于帮助我们开发完整且一致地描述了系统行为的严格规范，生成代码只是其一种可能应用。把抽象机精化到计算机语言可表达的结构后，人工翻译成某种语言也不困难

用 B 语言描述的高层抽象不能直接在计算机上执行，但它们可能揭示并严格刻画了软件的复杂行为，通过了严格的数学分析，具有很强的功能保证

在 B 语言里包含了一些在常规高级语言中常见的结构（如赋值和条件），也包含了一些不常见的结构。后部分结构在描述和设计软件时非常重要，但它们并不直接对应于计算机可实现的结构，不能执行

在基本的 B 语言里不包含顺序和循环，因为这两种结构属于“怎样做”的范畴，而在写软件规范时，我们主要关心的是这一软件“做什么”

B 语言描述软件规范的核心概念是“抽象机记法”（AMN, Abstract Machine Notation）和“广义代换语言”（GSL, General Substitution Language）

下面先简单介绍代换概念，而后结合介绍抽象机和广义代换的细节

代换（《B Book》21页）

代换，就是把一个公式（表达式或谓词）里某个（某些）变量的全部自由出现都统一地代换为一个表达式

给定变量 $\boldsymbol{x}$，表达式 $\boldsymbol{E}$ 和公式 $\boldsymbol{F}$，代换

$$[\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{F}$$

表示将 $\boldsymbol{F}$ 里所有自由出现的 $\boldsymbol{x}$ 统一代换为表达式 $\boldsymbol{E}$ 得到的公式

对于逻辑公式和一般表达式，代换都可以严格定义，如

$$[\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{F} = \begin{cases} \boldsymbol{E} & \text{若 } \boldsymbol{F} \text{ 就是 } \boldsymbol{x} \\ \boldsymbol{F} & \text{若 } \boldsymbol{x} \text{ 在 } \boldsymbol{F} \text{ 无自由出现} \\ [\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{P} \wedge [\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{Q} & \text{若 } \boldsymbol{F} \text{ 是 } \boldsymbol{P} \wedge \boldsymbol{Q} \\ & \text{其他二元运算符都类似} \\ \neg[\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{P} & \text{若 } \boldsymbol{F} \text{ 是 } \neg\boldsymbol{P} \\ \forall \boldsymbol{x}.\boldsymbol{P} & \text{若 } \boldsymbol{F} \text{ 是 } \forall \boldsymbol{x}.\boldsymbol{P} \\ \forall \boldsymbol{y}.[\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{P} & \text{若 } \boldsymbol{F} \text{ 是 } \forall \boldsymbol{x}.\boldsymbol{P} \text{ 且 } \boldsymbol{y} \text{ 不是 } \boldsymbol{x} \\ & \text{且 } \boldsymbol{y} \text{ 在 } \boldsymbol{E} \text{ 无自由出现} \end{cases}$$

最后一条“无自由出现”很重要，如果真有，可以在局部将 $\boldsymbol{y}$ 重命名

代换

在逻辑里，有一些有关代换的经典结果。如

$$\frac{\mathsf{HYP} \vdash \forall \boldsymbol{x}.\boldsymbol{P}}{\mathsf{HYP} \vdash [\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{P}}$$

也就是说，如果在假设集合 HYP 下 $\forall \boldsymbol{x}.\boldsymbol{P}$ 成立，那么将 $\boldsymbol{P}$ 中 $\boldsymbol{x}$ 代换为任意公式的 $\boldsymbol{E}$，得到的公式在同样假设下也必定成立

这一规则称为“删除规则”

另一重要规则是莱布尼茨法则（等量代换）

$$\frac{\mathsf{HYP} \vdash [\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{P} \qquad \mathsf{HYP} \vdash \boldsymbol{E} = \boldsymbol{F}}{\mathsf{HYP} \vdash [\boldsymbol{x} := \boldsymbol{F}]\boldsymbol{P}}$$

如果用相等的表达式代换，能证明的东西依然能证明

还有下面规则

$$\frac{\mathsf{HYP} \vdash \boldsymbol{F_1} = \boldsymbol{F_2}}{\mathsf{HYP} \vdash [\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{F_1} = [\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{F_2}}$$

在两个相等的表达式里做相同代换，得到的表达式依然相等

抽象机

抽象机是 B 方法中描述软件系统的基本单元，其基本想法就是基于状态和改变状态的操作的观点构造软件系统的模型

要描述软件系统的规范，就要分析待开发系统的潜在行为，根据对系统行为的认识去描述它。系统的行为分为“静态行为”和“动态行为”

- 静态行为对应于系统的状态
- 动态行为对应于改变系统状态的各种操作

其中静态行为是系统内部的东西，而操作形成了提供给系统的用户的接口。最简单的抽象机也包括下面几个基本部分：

MACHINE ...
VARIABLES ...
INVARIANT ...
INITIALISATION ...
OPERATIONS ...
END

静态行为

系统的静态行为描述一个系统的静态构造，主要包括两部分：

- 系统中的变量，它们是系统的状态的组成部分
- 不变式，描述了在系统状态不断变化中那些保持不变的东西

不变式是系统的静态规则，用谓词的形式描述。它约束了变量的取值范围和相互关系，它实际上规定了系统的合法状态空间

基本设计规定了某个变量取值于某个类型。为抽象机定义了一组状态变量后，其可能的状态空间就是这些变量的类型的笛卡儿积

但是实际上，并不是所有可能状态都是我们所希望的。这种状态限制就需要通过抽象机的不变式描述

例如要描述一个学生选课系统，学校制度可能规定：

- 一个学生的选课门数不超过某个整数，例如 6
- 一门课程是选课人数不超过课程教室的容量
- 一个学生所选的不同课程不能同时上课，等等

这些（对变量可能取值的）限制都要用抽象机的不变式描述

谓词和状态

一旦定义了一组变量和它们的可能取值，实际上就定义了一个状态空间

例如，假设定义了变量 x 和 y 均由集合 $\{1, 2, 3\}$ 取值，那么可能的状态空间就包括 9 个具体状态。对于个复杂的系统，其可能状态的数量非常巨大，描述这样的系统时不可能去列举一个个状态

经典方法是用谓词描述状态，一个谓词代表满足这一谓词的所有状态

例如，在上述状态空间里，谓词 $x = y + 1$ 表示的状态集合是：

$$\{(2, 1), (3, 2)\}$$

而谓词 $x > y$ 表示的状态集合是

$$\{(2, 1), (3, 1), (3, 2)\}$$

谓词 $x = 3$ 代表的状态集合是

$$\{(3, 1), (3, 2), (3, 3)\}$$

谓词 $x = 3 \wedge y \neq 2$ 代表的状态集合是

$$\{(3, 1), (3, 3)\}$$

用了谓词的语言，就可以比较方便地描述一组所需状态

动态行为

抽象机的行为用操作描述，操作改变抽象机的状态

写一个操作的规范，一种可能方法是逐个列举出它对各具体机器状态的变换结果（回想有穷自动机）。如果状态空间很大，这一做法不实用

一种可能：把变换前后的状态看作一个关系，设法描述这种关系。一种典型的写法是为表示后状态的变量加撇号，如写：

$$seat' = seat + 1$$

表示经过某变换，座位的个数增加了一个

上面这样的公式建立了后状态中的变量值对前状态变量值的一种依赖关系。同样可以只要求两者之间具有非确定性的关联，如

$$seat' < seat$$

这种逻辑语句称为“前-后谓词”，是 VDM 和 Z 采用的技术

B方法没采用上述方式（下面一些解释里借用了这种记法），而是将操作定义为状态的变换。由于状态用谓词表示，所以操作就是谓词变换器

动态行为

在定义了抽象机的状态之后，就可以考虑它的各种操作了

操作用 **OPERATIONS** 子句描述，其中给出各操作的规范

抽象机用户只能激活其操作，不能直接访问其内部状态。这一信息隐藏原理很重要，它使人能以改变状态表示并修改操作的方式去"精化"抽象机，保持操作名不变

用户使用这些操作时，可根据抽象规范理解其作用，而实际用的是这些操作经过精化后的最终形式：用某种语言形式写出的程序

《B Book》讨论了这种方法的合理性，我们后面考虑

设计抽象机时，应考虑如何提供适当的供"最终用户"使用的接口，作为一套操作这一机器的指令。这样才能使做出的机器更有通用性，也更容易做精化

以前面讨论的座位预定系统为例（见右），这里为 *booking* 抽象机定义了两个操作

MACHINE
booking
VARIABLES
seat
INVARIANT
$seat \in \mathbb{N}$
OPERATIONS
$book \cdots$
$cancel \cdots$
END

不变式、操作和证明义务

描述了一个操作后，需要确认它没有破坏抽象机的一致性，也就是要证明这一操作的规范"维持了不变式"，或说，它没破坏系统的静态行为准则

如果操作的规范维持了抽象机的不变式，作为其实现的实际操作满足操作的规范， 那么就能保证实际操作在运行中维持不变式（我们的追求）

对抽象机 *booking*，要证明 *cancel* 维持不变式，就是要证明如果做 *cancel* 之前不变式 $seat \in \mathbb{N}$ 成立，操作后它仍然成立

假设我们用下面前后状态关系谓词操作描述 *cancel* 的意义：

$$seat' = seat + 1$$

维持不变式，意味着机器的不变式应该对所有通过 $seat' = seat + 1$ 联系起来的 $seta'$ 和 $seat$ 都成立。也就是说，需要证明

$$seat \in \mathbb{N} \Rightarrow \forall seat' . (seat' = seat + 1 \Rightarrow seat' \in \mathbb{N})$$

这个谓词显然成立。它就是证明 *cancel* 满足要求的证明义务（PO）

【注意，如果考虑的是可实现的自然数集合，相应的谓词就有问题了】

前后谓词和代换

蕴涵式

$$seat \in \mathbb{N} \Rightarrow \forall seat'.(seat' = seat + \mathbf{1} \Rightarrow seat' \in \mathbb{N})$$

的左边是一个只包含前状态的谓词，它是保证本操作能（重新）建立满足不变式的状态，在操作之前的状态必须满足的最小（最弱）条件

下面考虑 B 方法的规范描述技术

B 方法的基础之一是用代换也可以做好操作的形式化工作

对任何公式 $\boldsymbol{P}$，变量 $\boldsymbol{x}$ 和表达式 $\boldsymbol{E}$，代换式

$$[\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{P}$$

表示将 $\boldsymbol{P}$ 中所有自由出现的 $\boldsymbol{x}$ 都用 $\boldsymbol{E}$ 取代得到的公式，例如

$$[\boldsymbol{x} := \boldsymbol{x} + \mathbf{1}](\boldsymbol{x} \in \mathbb{N})$$

就是一个代换式，通过实际代换，它就变成了

$$\boldsymbol{x} + \mathbf{1} \in \mathbb{N}$$

前后谓词和代换

前面说，证明 *cancel* 维持不变式需要证明

$$seat \in \mathbb{N} \Rightarrow \forall seat'.(seat' = seat + \mathbf{1} \Rightarrow seat' \in \mathbb{N})$$

用代换可以简化这一语句的形式，有关简化归功于下面逻辑法则

$$\forall \boldsymbol{x}.(\boldsymbol{x} = \boldsymbol{E} \Rightarrow \boldsymbol{P}) \Leftrightarrow [\boldsymbol{x} := \boldsymbol{E}]\boldsymbol{P}$$

这一法则称为 “单点规则”。把这一法则应用于上式，得到

$$seat \in \mathbb{N} \Rightarrow [seat' = seat + \mathbf{1}](seat' \in \mathbb{N})$$

应用代换得到：

$$seat \in \mathbb{N} \Rightarrow seat + \mathbf{1} \in \mathbb{N}$$

这也就是

$$seat \in \mathbb{N} \Rightarrow [seat := seat + \mathbf{1}](seat \in \mathbb{N})$$

这里没有量词也没有表示后状态的撇号。比前面语句更短也更简单

这一段讨论说明，可以用前后谓词写一个操作的规范，也可以用代换描述，两种陈述方式具有等价性（可以证明）。B 方法采用的是后者

不变式和证明义务

一般而言，对于不变式 $\boldsymbol{I}$，代换 $\boldsymbol{S}$，证明义务就是

$$\boldsymbol{I} \Rightarrow [\boldsymbol{S}]\boldsymbol{I}$$

下面的工作是推广代换的概念，使之能更好满足描述各种实际软件系统的需要。由此就得到了所谓的 “广义代换"

操作 *cancel* 的规范最后写成

$$cancel = \textbf{BEGIN } seat := seat + 1 \textbf{ END}$$

这实际上引进了一种新的代换形式

$$\textbf{BEGIN } \boldsymbol{S} \textbf{ END}$$

这里的 **BEGIN** 和 **END** 只是一对括号，做怎样的代换完全由 $\boldsymbol{S}$ 确定。即

$$\textbf{BEGIN } \boldsymbol{S} \textbf{ END } \mathrel{\widehat{=}} \boldsymbol{S}$$

也就是说，**BEGIN** $\boldsymbol{S}$ **END** 是这类代换的语法形式，其意义就是 $\boldsymbol{S}$：

语法	定义
BEGIN $\boldsymbol{S}$ **END**	$\boldsymbol{S}$

这样构造出的表达式还是一个代换，下面会看到更多新的代换形式

前条件代换

现在换一个问题，考虑 “并不是任何情况下都可用的" 代换，以抽象机 *booking* 里预定座位的 *book* 操作为例

如果简单地把 *book* 定义为（预定一个，座位数少一个）：

$$book = \textbf{BEGIN } seat := seat - 1 \textbf{ END}$$

根据前面的讨论，要证明这一操作维持不变式，就要证明：

$$seat \in \mathbb{N} \Rightarrow [seat := seat - 1](seat \in \mathbb{N})$$

也就是说，需要证明（实际代换后的）：

$$seat \in \mathbb{N} \;\Rightarrow\; seat - 1 \in \mathbb{N}$$

显然不可能证明，特别的，对 $seat = 0$ 上式为假

从实际的角度看，仅当目前还有剩余座位的情况下才能接受预定。这也就是说，*book* 操作的执行是有先决条件的（下面称为 “前条件"）

前条件也应该用谓词描述。有剩余座位可以用下面谓词描述

$$0 < seat$$

前条件代换

为描述有前条件的代换，B 引进了一种新的代换形式

$$\boldsymbol{P} \mid \boldsymbol{S}$$

其中 $\boldsymbol{P}$ 是谓词，$\boldsymbol{S}$ 是代换。其实际使用的语法形式是

语法	定义
PRE $\boldsymbol{P}$ **THEN** $\boldsymbol{S}$ **END**	$\boldsymbol{P} \mid \boldsymbol{S}$

使用这一代换形式写出的操作 $book$ 的定义是

$$book \;=\; \textbf{PRE } \mathbf{0} < seat \textbf{ THEN } seat := seat - \mathbf{1} \textbf{ END}$$

也可以格式化地写成：

$book =$
PRE $\mathbf{0} < seat$
THEN
$seat := seat - \mathbf{1}$
END

在 B 语言里，任意的换行和退格都不改变规范的意义

前条件代换的证明义务

要讨论前条件代换的证明义务，首先要说明前条件代换的意义。用前条件代换 $\boldsymbol{S} \mid \boldsymbol{P}$ 对谓词 $\boldsymbol{Q}$ 做代换的意义定义为：

$$[\boldsymbol{P} \mid \boldsymbol{S}]\boldsymbol{Q} \;\widehat{=}\; \boldsymbol{P} \wedge [\boldsymbol{S}]\boldsymbol{Q}$$

当前条件 $\boldsymbol{P}$ 不成立时，广义代换 $[\boldsymbol{P} \mid \boldsymbol{S}]$ 不保证做任何事情，自然不会保证完成 $\boldsymbol{Q}$（无论 $\boldsymbol{Q}$ 是什么）

这相当于一般认为的系统崩溃：无论希望它做什么，它都不保证完成

这种不能保证完成任何工作的代换称为 “非终止代换”（《B Book》6.3.1）

考虑有前条件情况下的证明义务，这时有一个附加前提，即要求证明：

$$\boldsymbol{I} \wedge \boldsymbol{P} \;\Rightarrow\; \boldsymbol{P} \wedge [\boldsymbol{S}]\boldsymbol{I}$$

注意：此式等价于

$$\boldsymbol{I} \wedge \boldsymbol{P} \;\Rightarrow\; [\boldsymbol{S}]\boldsymbol{I}$$

对操作 $book$，得到的证明义务是

$$\begin{aligned} & seat \in \mathbb{N} \wedge \mathbf{0} < seat \Rightarrow [seat := seat - \mathbf{1}](seat \in \mathbb{N}) \\ = \;& seat \in \mathbb{N} \wedge \mathbf{0} < seat \Rightarrow seat - \mathbf{1} \in \mathbb{N} \end{aligned}$$

抽象机参数

如果一个抽象机没有参数，那么它描述的就是一个确定的系统模块

参数化为抽象机规范留下有穷个可能的变化维度，在实例化抽象机时给定具体的实参，就能得到不同的实例抽象机

可以认为，一个参数化的抽象机描述了一类系统模块。这些模块具有类似结构和功能，它们相互差别就在于一些参数不同

抽象机的参数可以是简单标量（整数、布尔值等），也可以是有穷的非空集合。参数写在机器名后面的括号里，多个参数用逗号分隔

Atelier B 要求集合参数的名字全部由大写字母拼写，标量参数的名字里至少有一个小写字母（系统按这种方式区分是什么参数）

对参数的限制用 **CONSTRAINTS** 子句描述，其中应是一系列联立的限制条件，包括对标量参数的限制（如标量的类型），对集合参数的附加限制

所有集合参数都相互独立，但可以要求某标量参数是某集合参数的成员

抽象机参数看作抽象机里的常量，不能修改

参数化的抽象机

右边是我们熟悉的抽象机，它有一个标量参数，**CONSTRAINTS** 子句说明该参数的类型

这里的不变式改为

$$seat \in \mathbf{0}..max\_seat$$

意味着要求所有操作要考虑 $seat$ 当时的值（都需要有条件，或者是前条件，或者是 **IF** 条件）

初始化子句 **INITIALISATION** 也用代换描述，它也带来了证明义务，要保证初始化后的状态满足不变式

也就是说，要证明一个包含初始化代换的证明义务

$$? \Rightarrow [S_I]I$$

假设 I 是抽象机的不变式，S_I 是初始化代换

对现在的 $booking$，需证明（显然成立，对任何“?”）

$$\begin{aligned} & ? \Rightarrow [seat := max\_seat](seat \in \mathbf{0}..max\_seat) \\ = \; & ? \Rightarrow max\_seat \in \mathbf{0}..max\_seat \end{aligned}$$

MACHINE
$booking(max\_seat)$
CONSTRAINTS
$max\_seat \in \mathbb{N}$
VARIABLES
$seat$
INVARIANT
$seat \in \mathbf{0}..max\_seat$
INITIALISATION
$seat := max\_seat$
OPERATIONS
$book \cdots$
$cancel \cdots$
END

对一般情况，初始化产生的证明义务后面讨论

带参数的操作

操作可以有参数，参数看作操作里的常量

一个操作可以有任意多的输入和输出参数，一个操作的不同参数的名字必须互不相同

操作的输入参数必须通过操作开头的 **PRE** 结构给出类型，也就是说，带输入参数的操作的操作体一定是一个前条件代换

右边是预定和撤消若干座位的操作，其前条件为参数确定类型，并提出了其他要求

根据前面讨论，*bookn* 产生的证明义务是

$$\begin{array}{ll} & seat \in \mathbf{0}..max\_seat \wedge sn \in \mathbb{N} \wedge sn \leq seat \\ & \Rightarrow \\ & [seat := seat - sn](seat \in \mathbf{0}..max\_seat) \\ = & seat \in \mathbf{0}..max\_seat \wedge sn \in \mathbb{N} \wedge sn \leq seat \\ & \Rightarrow \\ & seat - sn \in \mathbf{0}..max\_seat \end{array}$$

$bookn(sn) =$
PRE
  $sn \in \mathbb{N} \wedge$
  $sn \leq seat$
THEN
  $seat := seat - sn$
END;

$canceln(sn) =$
PRE
  $sn \in \mathbb{N} \wedge$
  $seat + sn \leq max\_seat$
THEN
  $seat := seat + sn$
END

带参数的操作

根据自然数的理论，这一证明义务明显成立：

$$seat \in \mathbf{0}..max\_seat \wedge sn \in \mathbb{N} \wedge sn \leq seat \Rightarrow seat - sn \in \mathbf{0}..max\_seat$$

对于 *canceln*，也可以类似地写出证明义务：

$$seat \in \mathbf{0}..max\_seat \wedge sn \in \mathbb{N} \wedge seat + sn \leq max\_seat \Rightarrow [seat := seat + sn](seat \in \mathbf{0}..max\_seat)$$

代换后得到下式，它明显成立：

$$seat \in \mathbf{0}..max\_seat \wedge sn \in \mathbb{N} \wedge seat + sn \leq max\_seat \Rightarrow seat + sn \in \mathbf{0}..max\_seat$$

操作的输出参数写在操作名前 "$\longleftarrow$" 之前，可以是一组参数。输出参数需要在操作体中给值，实际上也给出了它们的类型。例如

$$snum \longleftarrow seat\_aval \ = \ \textbf{BEGIN } snum := seat \textbf{ END}$$

不仅 *snum* 得到了值，也确定了类型 $\mathbb{N}$。规范中也允许非确定性的输出

多重代换

下面讨论广义代换，介绍一些代换描述形式。下节还会讨论一些形式

先回忆一下多重代换，其形式是

$$[x_1, x_2 := E_1, E_2]F$$

左边是变量的有序对，右边是表达式的有序对

多重代换的意义是同时做代换，可以如下定义

$$[x_1, x_2 := E_1, E_2]F \ \widehat{=}\ [y := E_2][x_1 := E_1][x_2 := y]F$$

其中 y 是个新变量，与 x_1, x_2 都不同，且不在 F, E_1, E_2 自由出现

注意，同时代换与顺序代换不同，两者可能得到不同结果。例如

$$[y := 1][x := y + 1](x + y > 0) \ = \ [y := 1](2y + 1 > 0) \ = \ 3 > 0$$

$$[y, x := 1, y + 1](x + y > 0) \ = \ y + 2 > 0$$

显然，对各种复杂的表达式形式，都需要严格定义代换的作用。但实际上，只要看清公式中的哪些变量是自由的，对该种形式的公式做代换的效果就清楚了。在《B Book》里有这方面的清晰定义，这里将不再列举

多重代换

多重代换允许对任意多个变量做代换，其一般形式是

$$x_1, \ldots, x_n := E_1, \ldots, E_n$$

这里的 $x_1, \ldots, x_n$ 是 n 个互不相同的变量，$E_1, \ldots, E_n$ 是 n 个表达式

$$[x_1, \ldots, x_n := E_1, \ldots, E_n]P$$

表示将谓词 P 中 $x_1, \ldots, x_n$ 的所有自由出现同时代换为 $E_1, \ldots, E_n$

如果多重代换很长或者很多，写和读不方便，可以采用竖排形式

$$x_1 := E_1 \ ||$$
$$x_2 := E_2$$

竖排形式只是基本形式的另一种写法

$$x_1 := E_1 \ || \ x_2 := E_2 \ \widehat{=}\ [x_1, x_2 := E_1, E_2]F$$

抽象机的初始化和操作体里都可以写非确定性的代换

$$x :\in s$$

可以认为确定性代换是非确定性代换的特殊情况，$x := E = x :\in \{E\}$

空代换和条件代换

空代换什么也不做，将它作用于任意谓词还得到原来的谓词

$$[\mathsf{skip}]P \;=\; P \qquad \text{对任意的 } P$$

条件代换具有结构 **IF ... THEN ... ELSE ... END**，定义是

$$[\textbf{IF } P \textbf{ THEN } S_1 \textbf{ ELSE } S_2 \textbf{ END}]Q \;\widehat{=}\; (P \Rightarrow [S_1]Q) \wedge (\neg P \Rightarrow [S_2]Q)$$

假定我们需要证明条件代换 **IF** P **THEN** S_1 **ELSE** S_2 **END** 维持当前抽象机的不变式 I，那就是要证明：

$$I \;\Rightarrow\; (P \Rightarrow [S_1]I) \wedge (\neg P \Rightarrow [S_2]I)$$

根据谓词演算规则，这等价于证明

$$(I \;\Rightarrow\; (P \Rightarrow [S_1]I)) \wedge (I \;\Rightarrow\; (\neg P \Rightarrow [S_2]I))$$

这等价于证明

$$(I \wedge P \Rightarrow [S_1]I) \wedge (I \wedge \neg P \Rightarrow [S_2]I)$$

也就是说，一个条件代换可以导出两个证明义务，分别关注它的两个分支代换的不变式维持性：$I \wedge P \Rightarrow [S_1]I$ 和 $I \wedge \neg P \Rightarrow [S_2]I$

条件代换

简约形式的条件代换（没有第二个分支）可以基于上面基本形式定义：

$$\textbf{IF } P \textbf{ THEN } S_1 \textbf{ END} \;\widehat{=}\; \textbf{IF } P \textbf{ THEN } S_1 \textbf{ ELSE } \mathsf{skip} \textbf{ END}$$

不难推导出它表示的代换

$$[\textbf{IF } P \textbf{ THEN } S_1 \textbf{ END}]Q \;\widehat{=}\; (P \Rightarrow [S_1]Q) \wedge (\neg P \Rightarrow Q)$$

前面看到，如果一个操作的体是一个 **IF** 代换，由它将导出两个证明义务。对于简约形式的 **IF**，情况简单一些：

$$\begin{aligned} & I \;\Rightarrow\; (P \Rightarrow [S_1]I) \wedge (\neg P \Rightarrow I) \\ = \;& (I \wedge P \Rightarrow [S_1]I) \wedge (I \wedge \neg P \Rightarrow I) \\ = \;& I \wedge P \Rightarrow [S_1]I \end{aligned}$$

回忆前面前条件代换，可以看到，如果操作体是下面两个代换

$$\textbf{IF } P \textbf{ THEN } S \textbf{ END} \qquad\qquad \textbf{PRE } P \textbf{ THEN } S \textbf{ END}$$

导出的证明义务相同，都是 $I \wedge P \Rightarrow [S]I$。它们之间的不同在于条件不成立时的情况：**IF** 保证状态不变，而 **PRE** 没有任何保证

证明义务：实例

MACHINE
$enum\_machine$
VARIABLES
$tnext, snext$
INVARIANT
$tnext \in 1..80000 \wedge$
$snext \in 1..80000 \wedge$
$snext \leq tnext$
INITIALISATION
$tnext, snext := 1, 1$
OPERATIONS
$reset =$ **BEGIN**
$tnext, snext := 1, 1$
END;
$nn \longleftarrow waitn =$ **BEGIN**
$nn := tnext - snext$
END;

考虑左边银行取号和服务机器的证明义务

为了方便书写，下面用 N 代替 80000，用 I 表示抽象机的不变式

从初始化子句生成证明义务：

$$[tnext, snext := 1, 1]I$$
$$= 1 \in 1..N \wedge 1 \in 1..N \wedge 1 \leq 1$$

显然成立

操作 $reset$ 生成的证明义务是

$$I \Rightarrow [tnext, snext := 1, 1]I$$
$$= I \Rightarrow 1 \in 1..N \wedge 1 \in 1..N \wedge 1 \leq 1$$

显然成立

操作 $waitn$ 不生成任何证明义务

证明义务：实例

$nn \longleftarrow takeNaxt =$ **IF**
$tnext < 80000$
THEN
$nn, tnext := tnext, tnext + 1$
ELSE
$nn := 0$
END;

$nn \longleftarrow serveNext =$ **IF**
$snext < tnext$
THEN
$nn, snext := snext, snext + 1$
ELSE
$nn := 0$
END
END

操作 $takeNext$ 的体是 **IF** 代换， 它将生成两条证明义务，第一条

$$I \wedge tnext < N \Rightarrow$$
$$[nn, tnext := tnext, tnext + 1]I$$
$$= snext \in 1..N \wedge snext \leq tnext \wedge$$
$$tnext \in 1..N \wedge tnext < N$$
$$\Rightarrow$$
$$snext \in 1..N \wedge snext \leq tnext + 1 \wedge$$
$$tnext + 1 \in 1..N$$

第二条对于状态相当于 skip 代换

$serveNext$ 生成的第一条证明义务

$$tnext \in 1..N \wedge snext \leq tnext \wedge$$
$$snext \in 1..N \wedge snext < tnext$$
$$\Rightarrow$$
$$tnext \in 1..N \wedge snext + 1 \in 1..N \wedge$$
$$snext + 1 < tnext$$

集合和集合子句

现在介绍抽象机的集合和常量特征

集合通过 **SETS** 子句引进。可以同时说明多个集合，集合间用分号分隔

集合分为两种：

- 待定集合（或延期集合），说明时只给出集合名
- 枚举集合，说明时给出集合名，以及等号后用花括号括起的枚举符表

枚举符是枚举集合的元素，它们应互不相同。一个枚举集合的枚举符也被看作是以这一枚举集合作为类型的常量

枚举或待定集合都表示独立的类型，不能对它们的性质给以进一步限定

待定集合用于定义那些我们不计划在规范层次给出详尽定义的集合，待定集合都假定是非空的有限集合

待定集合最终需要在精化中用 **VALUES** 子句实例化为具体的集合。所有待定集合最终都将被实例化为一个非空的有限整数区间

常量

常量用 **CONSTANTS** 子句引入，然后在 **PROPERTIES** 子句里给定属性。属性是一组联立的谓词（用 $\wedge$ 连接）可以包括：

- 常量的类型
- 常量的值

可以以确定性或者非确定的方式指定常量的相应值，例如

PROPERTIES
$max\_num \in \mathbf{NAT} \wedge$
$max\_num = \mathbf{10000} \wedge$
$some\_num \in \mathbf{NAT} \wedge$
$some\_num \in \mathbf{1..100}$

抽象机里的集合和常量被看作抽象机规范描述的上下文信息， 它们起的作用与抽象机的集合和标量参数类似

不同之处在于参数来自抽象机的外部， 而集合和常量在抽象机内部定义

将上下文信息定义为参数，可以用不同的具体集合或标量实例化

常量

常量可以是

- 某一标量集合的常量，其说明的形式是：

$$c \in T$$

- 从一个或几个集合到一个集合的全函数

$$c \in S \rightarrow T \qquad c \in S_1 \times \cdots \times S_n \rightarrow T$$

- 某个标量集合的子集

$$c \in \mathbb{P}\, T$$

这些构造中的每个标识符都可以是一个待定集合，或者一个枚举集合，或者一个明确描述的整数区间

几个部分之间的关系

前面介绍了抽象机的一些基本部分，其中

- 有些提供了抽象机的基本信息，包括抽象机的参数、集合和常量
- 有些定义了抽象机状态的基本情况，就是变量定义
- 有些以谓词方式描述对上述各个方面的限制，包括 **CONSTRAINTS** 子句，**PROPERTIES** 子句和 **INVARIANT** 子句
- 最后是描述抽象机操作的 **OPERATIONS** 子句

它们之间有如下的允许引用关系（几个基本集合和常量，字面量总能用）

- 在 **OPERATIONS** 和 **INVARIANT** 部分，可以引用参数、集合、常量和变量部分引进的所有标识符
- 在 **PROPERTIES** 子句里，只能引用集合和常量部分引进的集合和常量。B 方法特别规定在这个子句里不能引用机器的参数，这是为了保证在复杂的机器组合中不出现循环
- **CONSTRAINTS** 子句里只能引用机器的参数

实例：阅读情况追踪

考虑下面追踪幼儿园小朋友阅读情况的系统。静态状态定义：

MACHINE
$Reading$
SETS
$READER; BOOK; COPY; RESPONSE = \{yes, no\}$
CONSTANTS
$copy$
PROPERTIES
$copy \in COPY \twoheadrightarrow BOOK$
VARIABLES
$hasread, reading$
INVARIANT
$hasread \in READER \leftrightarrow BOOK \wedge$
$reading \in READER \nrightarrow COPY \wedge$
$(reading; copy) \cap hasread = \varnothing$
INITIALISATION
$hasread := \varnothing \;||\; reading := \varnothing$

实例：操作定义

抽象机的动态行为定义

OPERATIONS
$take(rd, cp) =$
PRE
$rd \in READER \wedge cp \in COPY \wedge copy(cp) \notin hasread[\{rd\}] \wedge$
$rd \notin \mathsf{dom}(reading) \wedge cp \notin \mathsf{ran}(reading)$
THEN
$reading := reading \cup \{rd \mapsto cp\}$
END;

$return(rd, cp) =$
PRE
$rd \in READER \wedge cp \in COPY \wedge cp \in \mathsf{ran}(reading)$
THEN
$hasread := hasread \cup \{rd \mapsto copy(cp)\} ||$
$reading := \{rd\} \lhd reading$
END;

实例：操作定义

$res \longleftarrow isreading(rd) =$
PRE $rd \in READER$
THEN IF $rd \in \mathsf{dom}(reading)$ **THEN** $res := yes$
  ELSE $res := no$ **END**
END;

$bk \longleftarrow curreading(rd) =$
PRE $rd \in READER \wedge rd \in \mathsf{dom}(reading)$
THEN $bk := copy(reading(rd))$ **END;**

$res \longleftarrow rdhasread(rd, bk) =$
PRE $rd \in READER \wedge bk \in BOOK$
THEN IF $bk \in hasread[\{rd\}]$ **THEN** $res := yes$
  ELSE $res := no$ **END**
END
END

序列（参看《B Book》3.7节）

如果需要表示某类型的元素的有序序列，可以用序列表达式（sequence）表示。一个序列是同类型元素的一个有穷的有序列表

序列可以直接描述（外延表示），在一对方括号里列举元素，如：

$$genius = [Lao\_tzu, Chuang\_tzu, Congfucius]$$

没有元素的序列称为空序列，用 [] 表示。其他序列都是非空序列

序列有许多预定义的操作，假设 s 是一个序列

操作	解释
$\mathsf{size}(s)$	s 序列中元素的个数，序列大小
$\mathsf{first}(s)$	s 的第一个元素
$\mathsf{last}(s)$	s 的最后一个元素
$\mathsf{tail}(s)$	s 除去第一个元素剩下的序列
$\mathsf{front}(s)$	s 除去最后一个元素剩下的序列
$\mathsf{rev}(s)$	s 翻转得到的序列

这些操作都以一个序列作为操作对象，得到子序列或者元素

序列

另一组操作从已有序列和元素构造出序列（显然有类型问题）

操作	解释	正文符号
$s_1 \frown s_2$	两个序列的拼接	^
$e \rightarrow s$	前端加入	->
$s \leftarrow e$	后端加入	<-
$s \uparrow n$	前缀（取前 n 个元素的子序列）	/| \
$s \downarrow n$	后缀（去掉前 n 个元素后的子序列）	| /
$\mathsf{conc}(s)$	广义拼接	conc

注意：这里虽然说加入，拼接，实际上都是做出新序列

上述序列操作之间有些重要的关系。如

$$s = \mathsf{first}(s) \rightarrow \mathsf{tail}(s)$$

$$s = \mathsf{front}(s) \leftarrow \mathsf{last}(s)$$

在做前端限制时，如果 n 大于序列的长度，得到的还是原序列

在做后端限制时，要求 n 不大于序列的长度

序列：定义

$\mathsf{seq}(S)$ 表示 S 上的所有可能序列的集合

实际上，序列就是一类特殊形式的函数。集合 $\mathsf{seq}(S)$ 的定义是

$$\mathsf{seq}(S) \mathrel{\widehat{=}} \bigcup n.(n \in \mathbb{N} \mid \mathbf{1..}n \rightarrow S)$$

也就是说，$\mathsf{seq}(S)$ 是所有从 1 开始的有穷区间到 S 的函数，其中 $n = \mathbf{0}$ 的情况是空区间 $\mathbf{1..0}$，对应的是空序列

因此，序列不是新东西。中国历史人物的序列也可用函数（或集合）写出

$$genius = \{\mathbf{1} \mapsto Lao\_tzu, \mathbf{2} \mapsto Chuang\_tzu, \mathbf{3} \mapsto Congfucius\}$$

由于序列就是函数，可以利用函数的记法

$$s(\mathbf{1}),\ s(\mathbf{2}),\ \ldots$$

分别得到序列 s 里的第一个、第二个、. . . 元素

很容易得到序列里某个特定元素的出现次数：

$$\mathrm{card}(s^{-1}[\{e\}])$$

序列的集合

B 方法里还定义了另外几个序列集合

语法	解释	正文形式
$\mathsf{seq}_1(S)$	S 上的所有非空序列的集合	`seq1()`
$\mathsf{iseq}(S)$	S 上所有内射序列的集合	`iseq()`
$\mathsf{iseq}_1(S)$	S 上所有内射非空序列的集合	`iseq1()`
$\mathsf{perm}(S)$	S 上所有排列的集合	`perm()`

所谓内射序列，就是要求序列中没有重复元素。因此作为函数它是内射

S 上的排列是包含了 S 中所有元素的内射序列（显然要求 S 有穷）

基于函数概念和记法，上述集合都可以严格定义。例如

$$\mathsf{iseq}(S) \quad \widehat{=} \quad \{s \mid s \in \mathsf{seq}(S) \wedge s \in \mathbb{N}_1 \rightarrowtail S\}$$

$$\mathsf{perm}(S) \quad \widehat{=} \quad \{s \mid s \in \mathsf{iseq}(s) \wedge s \in \mathbb{N}_1 \twoheadrightarrow S\}$$

- 内射序列就是序列，而且是内射（部分内射，因为定义域是自然数区间）
- 排列序列是内射（没有重复元素），而且是满射（S 里的某个元素都出现）。说是部分满射的原因同上

序列：实例和性质

收到的信用卡帐单是一个序列，其中包含一系列付款项。假设付款项的集合是 $Payment$，那么一个具体帐单就是

$$bill \in \mathsf{seq}_1(Payment)$$

因为空帐单不会发过来，所以用 seq_1。另一方面，完全可能有同样付款项在一个帐单上多次出现，因此将帐单定义为 $\mathsf{iseq}_1(Payment)$ 就不合适

序列有许多性质，例如：

1. $(s \uparrow n)\,\hat{}\,(s \downarrow n) = s$，要求 $n \leq \mathsf{size}(s)$
2. $0 \leq m \leq n \leq \mathsf{size}(s) \Rightarrow (s \uparrow n \uparrow m = s \uparrow m)$
3. $n + m \in 0..\mathsf{size}(s) \Rightarrow (s \downarrow n \downarrow m = s \downarrow (n + m))$
4. $0 \leq m \leq n \leq \mathsf{size}(s) \Rightarrow s \uparrow n \downarrow m = s \downarrow m \uparrow (n - m)$
5. $\mathsf{first}(x \rightarrow s) = x$ 而且 $\mathsf{tail}(x \rightarrow s) = s$
6. $\mathsf{last}(s \leftarrow x) = x$ 而且 $\mathsf{front}(s \leftarrow x) = s$

抽象机实例：终点记录机

现在要设计一台机器，用于在马拉松终点记录选手到达的结果

假定选手集合是 $RUNNER$，要记录到达顺序，自然需要用一个序列变量

$$finish \in \mathsf{seq}(RUNNER)$$

机器的状态定义是：

MACHINE
  $Results$
SETS
  $RUNNER$
VARIABLES
  $finish$
INVARIANT
  $finish \in iseq(RUNNER)$
INITIALISATION
  $finish := [\,]$

抽象机实例：终点记录机

记录机的操作

OPERATIONS
  $finished(rn) =$
  PRE
    $rn \in RUNNER \wedge rn \notin \mathsf{ran}(finish)$
  THEN
    $finish := finish \leftarrow rn$
  END;

  $rn \longleftarrow query(pos) =$
  PRE
    $pos \in \mathbb{N}_1 \wedge pos \leq \mathsf{size}(finish)$
  THEN
    $rn := finish(pos)$
  END;

- $finish$ 将一个到达选手记入；$query$ 查询第 pos 位置的选手

抽象机实例：终点记录机

其他操作

$dequalify(pos) =$
PRE
$pos \in \mathbb{N}_1 \wedge pos \leq size(finish)$
THEN
$finish := (finish \uparrow (pos - 1)) \frown (finish \downarrow pos)$
END;

$ss \longleftarrow medals =$
BEGIN
$ss := finish \uparrow 3$
END
END

- *dequalify* 去掉一名犯规选手的排名
- *medals* 选出得奖牌的前三名

总结

本节讨论了 B 方法的基础，包括：

- 代换：概念和代换操作
- 抽象机结构的重要部分及其相关形式规定
- B 方法的理论基础，如何证明抽象机的一致性
- 不变式和证明义务
- 抽象机的参数
- 带参数的操作和证明义务
- 一些广义代换结构：前条件代换，条件代换，空代换
- 集合和常量
- 抽象机各部分之间的相互引用关系
- 序列：概念，描述，操作，性质等