形 式 化 方 法:

基于 B 方法的严格软件开发

(12)

实现 (一)

裘宗燕

北京大学数学学院信息科学系

2010年春季

引言

本次课的主要讨论如何通过 "分层" 模块的方式开发大型软件系统，以及抽象机的实现。课程内容主要参考 《B Book》 的第12章

B 方法对软件系统的基本看法是：

- 一个复杂软件由一批相对独立开发的模块组成，模块之间可能有组合关系或者信息参考关系
- 一个模块的抽象规范是一个抽象机，其实现需要通过一系列精化步骤完成，最终得到这一模块的可执行版本
- 模块的最终的版本称为 **IMPLEMENTATION**, 也是一种抽象机
- 一个模块的最终实现可能要导入另一些机器作为其实现的基础（组合），将被导入机器的实现作为本模块的最终实现中的组件
- 一个模块的实现还可能参考另外一些机器的信息

在深入讨论与实现和实现有关的各种问题之前，先看一个简单的例子，它说明了基于 B 方法的软件构造的基本结构

一个简单的精化开发实例

考虑下面的简单抽象机规范

```
MACHINE LMachine
VARIABLES ys
INVARIANT ys: FIN(NAT1)
INITIALISATION ys := {}
OPERATIONS
    enter(nn) =
      PRE nn : NAT1 THEN ys := ys \/ {nn} END;
    mm <-- maximum =
      PRE ys /= {} THEN mm := max(ys) END
END
```

这里的状态用一个自然数集合表示，操作中维护这一集合

但是，从机器的外部看，本抽象机只支持查看集合中最大的元素。因此维护一个集合数据结构，实际上维护了过多的信息

下面的精化就是考虑机器状态的简化

一个简单的精化开发实例

第一个精化机器里用自然数标量变量 zz 精化抽象机器规范里的自然数集合变量 ys，在 zz 里记录已经遇到的最大自然数值

```
REFINEMENT LMachineR
REFINES LMachine
VARIABLES zz
INVARIANT zz : NAT & zz = max(ys \/ {0})
INITIALISATION zz := 0
OPERATIONS
    enter ( nn ) = zz := max({zz, nn}) ;
    mm <-- maximum =
      PRE  zz /= 0 THEN mm := zz END
END
```

相应操作的实现都很直接

注意，这里精化后的 maxium 操作用了另一个前条件，原因是现在的操作可执行有了新的判断条件。实际开发中也常常需要这样做

一个简单的精化开发实例

由于集合元素最大值的判断仍然不是直接可实现的，还要对机器做进一步精化，主要工作是精化其中的操作 enter

```
REFINEMENT LMachineRR
REFINES LMachineR
ABSTRACT_VARIABLES zz1
INVARIANT zz1 = zz
INITIALISATION zz1 := 0
OPERATIONS
   enter ( nn ) =
     IF nn > zz1 THEN zz1 := nn END ;
   mm <-- maximum = BEGIN mm := zz1 END
END
```

至此这一精化已经完成了。实际上可以把这个精化定义为一个实现

下面要做是换一种实现方式。假定已经有另一个比较 “简单” 的机器，现在想直接使用该机器的功能来实现第一个精化机器 LMachineR

一个简单的精化开发实例

假设已经有了一部 “现成的” 机器 Scalar 如下

```
MACHINE Scalar(initval)
CONSTRAINTS initval : NAT
CONCRETE_VARIABLES zz2
INVARIANT zz2 : NAT
INITIALISATION zz2 := initval
OPERATIONS
    modify(val) =
      PRE val : NAT THEN zz2 := val END ;
    val <-- value = BEGIN val := zz2 END
END
```

这部机器只是简单地记录值，还可以随时读取其中保存的值

下面的想法是基于这一简单机器的功能，“实现” 前面的第一个精化机器 LMachineR。注意，目标也是做出最初的抽象机规范 LMachine 的实现

下面给出这样开发的结果，它也是一部机器

一个简单的精化开发实例

```
IMPLEMENTATION LMachineRI
REFINES LMachineR
IMPORTS Scalar(0)
INVARIANT zz2 = zz
OPERATIONS
    enter ( nn ) =
    VAR vv IN
        vv <-- value;
        IF nn > vv THEN modify(nn) END
    END ;
    mm <-- maximum =
      BEGIN mm <-- value END
END
```

这一机器 "实现" 了精化机器 LMachineR，间接地实现了最初的抽象机规范 LMachine

这一实现中，借助 B 语言的 **IMPORTS** 机制使用了另一抽象机 Scalar 的功能，用它记录本机器需要记录的信息

这个例子虽简单，但反映了 B 方法开发的系统的基本 "物理" 组织结构：

- 纵向：从抽象的初始规范，经过一步步精化，最终得到 "实现"
- 横向，一组相对独立开发的模块，通过 **IMPORTS** 集成

具体常量和抽象常量

在细致讨论 **IMPLEMENTATION** 机器之前，先介绍一下 B 中的变量和常量的分类："抽象常量" 和 "具体常量"，"抽象变量" 和 "具体变量"

前面写 **CONSTANTS** 相当于写 **CONCRETE_CONSTANTS**，定义的实际上是 "具体常量"，具体常量是能在程序设计语言实现的常量数据项，且默认地在随后的精化和实现里始终保持。具体常量的值可以是

- 整数或布尔类型的值
- 待定集合或枚举集合的元素
- 自然数的一个有限的非空区间
- 一个待定集合，或者一个数组

开发中也可能需要这样的常量：希望它表示的数据项具有常量值，但其具体的值在写抽象规范时还不能确定，需要在随后的精化中精化。B 为此提供了 "抽象常量"，在 **ABSTRACT_CONSTANTS** 子句里定义

抽象常量的值以非确定性方式给出，而后用更具确定性的值精化。对它们的精化在精化机器的 **ABSTRACT_CONSTANTS**/**CONCRETE_CONSTANTS** 和 **PROPERTIES** 子句里描述（用同样的常量名）

具体常量和抽象常量

具体常量和抽象常量都是 “常量”，这是说它们在完成的软件系统的运行中都始终代表一个确定值，在系统运行中不改变

说常量是 “具体” 或 “抽象”，则是指在软件开发过程中的不同情况

- 如果在开发中（无论何时）需要引入一个常量，当时就可以确定它的具体值，那么就应将其定义为 “具体常量”，定义时给定具体的确定的值
- 如果需要引入的常量（写抽象机器或精化机器时）的值尚无法确定，就应定义一个抽象常量（要通过给定谓词给定类型，也可能给出更多信息，如 $conx \in 1..10$），将确定值的工作留给后续的精化工作
- 在 “实现” 机器（下面讨论）里不能有抽象常量

精化或实现时可以在 **CONCRETE_CONSTANTS** 子句（**CONSTANTS** 子句）里用具体常量去精化被精化机器的抽象常量。在 **CONSTANTS** 里写同样名字的常量，就表示对相应抽象常量的精化。还要在 **PROPERTIES** 里给定常量值

如果一个抽象机里的抽象常量，在精化机器里没有对应的抽象常量，也没被精化为具体常量，那么该常量就消失了。这种情况也常见

抽象变量和具体变量

变量也分为 “抽象变量” 和 “具体变量”，前面用 **VARIABLES** 子句定义变量，相当于用 **ABSTRACT_VARIABLES** 定义，定义的实际上是抽象变量

在写抽象机器或精化机器时，定义变量通常是为了将来精化，或用其他变量取代（改变状态空间，用连接不变式关联），这样的变量就是抽象变量

如果认为要定义的变量应该出现在最终的系统里，不需要再精化了，就应该将其定义为 “具体变量”。具体变量在 **CONCRETE_VARIABLES** 子句里定义，同样需要在 **INVARIANT** 里定类并描述不变性质，在 **INITIALISATION** 里初始化。具体变量应该能在程序设计语言里实现，因此有些特殊规定：

- 这种变量不能在抽象机的精化中精化。在精化或实现机器里，已有定义的具体变量都自动维持（保留），不必另行说明
- 在精化或实现中引进的具体变量不能与已有具体变量重名，但可以与已有的抽象变量同名，这时它就是哪个变量的精化，等于那个抽象变量，自动具有与那个抽象变量相同的类型
- 由于需要实现，具体变量的类型有限制，只能是标量或者表示数组的全函数，不能是其他类型

变量和常量

抽象变量可以具有任何类型，可在机器精化时精化，前面已讨论。还有

- 如果在精化机器里原机器的一个抽象变量没有出现，也没有一个同名的具体变量，那么该抽象变量就消失了，不再是精化机器状态的一部分
- 在实现里不能定义抽象变量

到了实现机器的层面上，只能有具体变量和具体常量了，实现机器里的操作都是对于这些常量和变量进行的

由于实现里的操作都需要能直接翻译到程序设计语言，所以这里的常量和变量也都必须能实现。正因为这样，需要限制具体变量的类型

- 独立的标量集合或标量类型的子集，相应变量翻译为某标量类型的变量
- 从一个标量集合到另一标量集合的全函数，或者从标量集合的笛卡儿积到另一标量集合的全函数。相应变量翻译为数组

下面讨论精化的最后步骤：如何开发软件系统的实现组件

“实现”简介

用 B 方法进行软件开发的最终目标是生成能在计算机上运行的代码。B 方法的“实现机器”是一类特殊的精化机器，其特殊性就在于可以由它生成代码

- 一个系统的抽象模型（抽象机器）可能经过多次精化，直到最后得到能在计算机上运行的最终精化结果，这就是实现
- 这一精化结果可能导入另外一些抽象机作为其实现的基础
- 这样可以形成丰富的组织结构，以便开发大型的抽象机

下面将首先介绍相关的基本情况，而后详细讨论各方面细节

在实现里可以出现两个新的子句：

- 一个 **IMPORTS** 子句，用于建立项目里的模块实例，组合产生复杂的部件
- 一个 **VALUES** 子句，用于给待定集合和具体常量赋值

这两种结构（子句）只能出现在实现机器（**IMPLEMENTATION**）里，不能出现在 **MACHINE** 或者 **REFINEMENT** 机器里

“实现”简介

在实现机器里，不允许定义抽象变量和抽象常量，也就是说，不能出现 **ABSTRACT_CONSTANTS** 和 **ABSTRACT_VARIABLES**（**VARIABLES**） 子句。另外，也不允许出现 **CONSTRAINTS**, **INCLUDES** 和 **USES**子句

如果在实现机器通过 **IMPORTS** 子句导入已有的机器，那么也就间接得到那些机器里的抽象变量和具体变量。有些情况需要说明

- 实现机器里不能直接修改导入的抽象变量，所有修改都只能通过调用被导入机器的操作间接完成
- 可以将导入机器的抽象变量或抽象常量写在不变式里，建立这些量与被本实现机器精化的机器里的常量或变量之间的关系
- 可以在 **PROPERTIES** 子句里描述导入机器里的常量和其他常量的关系
- 可以使用被导入机器里定义的具体变量，但只能以读的方式使用，或调用被导入机器里的操作去使用

IMPORTS 只能用于导入抽象机，不能用于导入精化或实现。在 **IMPORTS** 一个机器时，也可以对它做重命名。回忆一下：重命名机器将重命名其中的变量和操作（需要用加圆点前缀），但不重命名其集合和常量

“实现”简介

实现机器里可以定义自己的具体变量和具体常量，也可以定义与被它精化的机器里的抽象变量同名的具体变量，代表对那些变量的精化

实现机器里也可以有 **EXTENDS** 子句， **EXTENDS** 后面列出一些机器名，作用相当于先 **IMPORTS** 这些机器，而后提升这些机器的所有操作（相当于在抽象机器和精化机器里 **EXTENDS** 与 **INCLUDES** 子句的关系）

被导入的机器里的延期集合和枚举集合，都像本机的相应成分一样使用

在一个实现的各层次抽象（抽象规范 **MACHINE** 或者精化 **REFINEMENT** 里定义的所有待定集合，都必须在这个实现的 **VALUES** 子句里给定具体的值

实现机器里的 **INITIALISATION** 子句和 **OPERATIONS** 子句的作用与抽象机器或精化机器不同，特别是其中的操作必须可以实现。下面讨论

IMPORTS M 与 **INCLUDES** M 不同。**IMPORTS** M 实际上是把 M 的最终实现包含到了本机器所在的系统里。这种做法使 M 可以独立地精化和实现，更有利于复杂系统的模块化开发（使各模块可以独立开发）

IMPORT（导入）子句和被导入的机器

IMPORTS 子句的形式和意义都像 **INCLUDES**，子句里列出要导入机器的名字。如果被导入机器有参数，必须给予实例化。送给被导入机器的实参要满足被导入机器里的 **CONSTRAINTS** 子句的要求

假定是实现机器 M_n 需要为被导入机器 M 提供参数，有下面规定

- 对于被导入机器的标量参数，相应实际参数可以是：
  - M_n 自身的标量形参
  - 被 M_n 精化的机器里定义的枚举集合的枚举值
  - 被 M_n 查看（**SEES**）的机器里的具体常量或枚举值
  - 基本算术表达式，其中可以涉及 M_n 的数值形参， M_n 及被 M_n 精化的机器里声明的具体数值常量，或直接用文字量
- 对于被导入机器的集合参数，对应实参可以是
  - M_n 自己的形参
  - M_n 或被 M_n 精化或查看的机器里声明的枚举集合或待定集合
  - 直接描述的非空区间，其限界是数值，可以用表达式表示

IMPORT 和被导入的机器

在 B 方法的各种结构相互联系时，都规定了相应的可见性规则。《B Book》附录 D 列出了所有可见性规则，在写 B 规范时需要查阅

实际上，大部分可见性规则都是很自然，原因是容易想清楚的

被导入机器里的各部分在当前机器里的可见性见下表

本机的（实现）：	值	性质	不变式	操作
参数（被导入机器）				
集合	X	X	X	X
具体常量	X	X	X	X
抽象常量		X	X	只能在循环不变式里
具体变量			X	X
抽象变量			X	只能在循环不变式里
操作				X

由于各种常量和变量都可以用在 **INVARIANT** 里，这样就可以描述被导入的机器与被精化的机器之间的各种联系（静态联系或动态的状态联系）

有关导入的一些情况

在实现里用 **IMPORTS** 导入了一个抽象机，实际上要做一些工作：

- 用 **IMPORTS** 导入并实例化抽象机，提供实际参数
- 将导入机器的变量结合进来，作为实现里的变量
- 把导入并实例化的机器的不变式结合到实现的不变式里
- 把导入并实例化的机器的初始化动作结合到实现的不变式里
- 在实现的操作里把对导入机器的操作调用展开（做适当代换后）

完成这些工作（B 方法的工具能自动完成）后，事情就回到了一般精化的范围，随后的精化定理及其证明就与普通精化一样了

前面实例中最后实现里的 $enter$ 操作，将调用 $Scalar$ 的操作展开后是

$$
\begin{array}{rcl}
enter(nn) & = & \textbf{VAR } vv \textbf{ IN} \\
 & & \quad \textbf{BEGIN } vv := zz2 \textbf{ END;} \\
 & & \quad \textbf{IF } nn > vv \textbf{ THEN} \\
 & & \quad\quad \textbf{PRE } nn \in \mathbb{N} \textbf{ THEN } zz2 := nn \textbf{ END END} \\
 & & \textbf{END;}
\end{array}
$$

有关导入的一些情况

按前面有关精化的讨论，这时的证明义务是：

$$
\begin{array}{l}
ys : \mathbb{F}(\mathbb{N}_1) \wedge zz = \mathsf{max}(ys \cup \{0\}) \wedge zz2 = zz \wedge zz2 \in \mathbb{N} \wedge nn \in \mathbb{N} \\
\Rightarrow [enter(nn)] \neg [zz := \mathsf{max}(\{zz, nn\})] \neg (zz2 = zz)
\end{array}
$$

做完内层代换，得到：

$$
\begin{array}{l}
ys : \mathbb{F}(\mathbb{N}_1) \wedge zz = \mathsf{max}(ys \cup \{0\}) \wedge zz2 = zz \wedge zz2 \in \mathbb{N} \wedge nn \in \mathbb{N} \\
\Rightarrow [enter(nn)](zz2 = \mathsf{max}(\{zz, nn\}))
\end{array}
$$

把 $enter$ 操作的体带入后消去 **VAR** 结构：

$$
\begin{array}{l}
ys : \mathbb{F}(\mathbb{N}_1) \wedge zz = \mathsf{max}(ys \cup \{0\}) \wedge zz2 = zz \wedge zz2 \in \mathbb{N} \wedge nn \in \mathbb{N} \\
\Rightarrow \\
\forall vv \cdot (\ [\textbf{BEGIN } vv := zz2 \textbf{ END;} \\
\qquad \textbf{IF } nn > vv \textbf{ THEN} \\
\qquad\quad \textbf{PRE } nn \in \mathbb{N} \textbf{ THEN } zz2 := nn \textbf{ END END} \\
\qquad] \ (zz2 = \mathsf{max}(\{zz, nn\})) \ )
\end{array}
$$

由于 vv 在前件里非自由，可以去掉全称量词

$$ys : \mathbb{F}(\mathbb{N}_1) \wedge zz = \max(ys \cup \{0\}) \wedge zz2 = zz \wedge zz2 \in \mathbb{N} \wedge nn \in \mathbb{N}$$
$$\Rightarrow$$
$$[\ \textbf{BEGIN}\ vv := zz2\ \textbf{END};$$
$$\quad \textbf{IF}\ nn > vv\ \textbf{THEN PRE}\ nn \in \mathbb{N}\ \textbf{THEN}\ zz2 := nn\ \textbf{END END}$$
$$\ ]\ (zz2 = \max(\{zz, nn\}))$$

把其中的条件代换展开，得到

$$ys : \mathbb{F}(\mathbb{N}_1) \wedge zz = \max(ys \cup \{0\}) \wedge zz2 = zz \wedge zz2 \in \mathbb{N} \wedge nn \in \mathbb{N}$$
$$\Rightarrow$$
$$[vv := zz2]$$
$$((nn > vv \Rightarrow (nn \in \mathbb{N} \wedge nn = \max(\{zz, nn\})))$$
$$(\neg(nn > vv) \Rightarrow zz2 = \max(\{zz, nn\}))\ )$$

做代换，并根据前件的 $zz2 = zz$ 去掉公式中 $zz2$，得到

$$ys : \mathbb{F}(\mathbb{N}_1) \wedge zz = \max(ys \cup \{0\}) \wedge nn \in \mathbb{N}$$
$$\Rightarrow$$
$$((nn > zz \Rightarrow nn = \max(\{zz, nn\}))$$
$$(nn \leq zz \Rightarrow zz = \max(\{zz, nn\}))\ )$$

后件显然为真

这个公式成立

VALUES 子句

VALUES 子句用于为本实现所精化的各抽象（**MACHINE** 或 **REFINEMENT**）里声明的常量和待定集合确定最终值。下面说明相关规则

下面讨论中将始终假定所考虑的实现机器是 M_n，由初始规范 M_1 经过精化链 $M_1, M_2, \ldots, M_n$ 达到 M_n。再假定 M_n 还导入了机器 M

在 $M_1, M_2, \ldots, M_n$ 里声明的可见常量，都可以在 M_n 的 **VALUES** 子句里给确定值，方式是写一个左部为这种常量的等式，其右部可以是：

- M_n 导入或查看的任一机器里的一个具体常量
- $M_1, M_2, \ldots, M_n$ 之一里的，或 M_n 查看的某机器里的一个具体常量
- 一个算术表达式，其中可以写被导入机器 M 的或被 M_n 查看的任一机器里的具体数值常量或文字量，可以用简单算术运算符
- 一个以外延方式定义的全函数，其值都是上面三种形式的常量
- 一个定义在某区间上的全函数，区间用上面形式的常量描述
- 一个集合，它在 M_n 导入的机器 M 里或 M_n 查看的某机器里可见
- 用上面说的算术表达式形式描述的区间

VALUES 子句

按上面规定，$M_1, M_2, \ldots, M_n$ 里声明的具体常量的值不能来自 M_1 的标量形参或 $M_1, M_2, \ldots, M_n$ 的其他具体常量，这是为避免循环定义。这些具体常量的值只能来自文字量，或被 **IMPORTS** 的或被查看机器的具体变量

如果 $M_1, M_2, \ldots, M_n$ 里声明了具体常量 c，而被 M_n 导入的 M 或查看的某机器里也声明了 c，那么 M_n 的 **VALUES** 子句里不给 c 定值，因为 c 的值应该其定义所在的 M 或被查看机器的实现里给定

$M_1, M_2, \ldots, M_n$ 里声明的待定集合可以在 M_n 的 **VALUES** 子句里获得最终值，方式是写一个等式，其左部是这个集合，右部可以是：

- M_n 导入的 M 或查看的某机器里的一个待定集合
- 一个限界为数值标量的非空区间，符合上面有关描述标量数值的规定

同样为防止循环定义，要求赋给 $M_1, M_2, \ldots, M_n$ 的待定集合的具体值或来自被 M_n 导入的或查看的机器里的具体对象，或是直接描述的非空区间

与常量的情况类似，若 $M_1, M_2, \ldots, M_n$ 里声明了待定集合 S，而 M_n 导入或查看的某机器里也声明了 S，那么 S 的值由那个机器的实现给定

IMPORTS/INCLUDES 和 PROMOTES/EXTENDS

抽象机里可以用 **INCLUDES**，在实现里用 **IMPORTS**，对它们做些比较

INCLUDES 把一部机器包含进另一机器，以构造复杂的规范。**IMPORTS** 把一部机器导入另一机器的实现，则是为了构造复杂的软件系统

规范的组织和软件系统的组织有类似的方面，但不必采用同样分解规则

理论上说，应该先考虑软件系统的分解，设计好分解和集成的方式。在开发具体的软件部件时，再考虑如何分解软件的描述。形式化开发应该一样

两种子句的可见性也有差异。被包含机器的变量在包含机器的 **OPERATIONS** 子句里部分可见（只读，半隐藏），被导入机器里的抽象变量在实现的 **OPERATIONS** 子句里完全不可见。这是可见性方面仅有的不同

一个实现可以用 **PROMOTES** 子句列出导入的机器的一些操作名，让使用本实现的这些操作时自动调用导入机器的操作（不在 **OPERATIONS** 子句里列出它们）；用 **EXTENDS** 相当于 **IMPORTS** 后 **PROMOTES** 所有操作。易见机器的 **PROMOTES**/**EXTENDS**与实现的 **PROMOTES**/**EXTENDS** 的作用不同，因为机器不能 **IMPORTS** 其他机器， 实现也不会包含其他机器

实现使用的代换语言

由于实现（IMPLEMENTATION）要翻译到实际的程序设计语言，因此其中不能使用任意的数学结构，显然要有些限制

- 其中的变量都是具体变量，满足前面说的限制
- 使用的代换都应该能直接翻译到实际程序设计语言的语句

《B Book》的第 12.1.11 节列出了所有可以在实现里使用的代换结构，以及用于构造代换的基础结构。在 Atelier B 语言手册第 7.25 节给出了 B 语言的可执行子集 B0 语言的完整语法，也就是在 B 语言里描述实现时可用的基本语言（加上 **IMPLEMENTATION** 本身的语法结构，就构成实现的语言）

这里不打算详细罗列这一代换语言的语言细节，只说明一些情况

代换语言最上层的结构是语句：

$$
\begin{array}{rcl}
Statement & ::= & \mathsf{skip} \mid Assignment\_Statemant \\
& \mid & Call\_Statement \mid Sequential\_Statement \\
& \mid & Local\_Statement \mid While\_Statement \\
& \mid & If\_Statement \mid Case\_Statement
\end{array}
$$

实现使用的代换语言

任何程序设计语言都包含这些结构，因此它们的可实现性很清楚

应看到，这里的 “语句” 有双重解释：

- 一方面，它们是数学（逻辑）里的代换，是谓词变换器，具有清晰严格的数学意义，因此可以严格推理和证明
- 另一方面，可以用程序设计语言里相应的结构给它们一个操作性的解释：每个代换对应于程序语言描述的（计算机执行的）一些动作

我们就是要利用这种双重意义，完成从数学结构和证明到程序和执行的转换

后面几种组合控制结构的情况都比较简单，不需要过多说明。下面主要讨论基本 $Assignment\_Statemant$ 及其组成部分（表达式）的一些情况

为良好地处理从数学结构到程序的转换，需要把基本赋值和操作调用分别区分为无保护和受保护的两类：

$$
\begin{array}{rcl}
Assignment\_Statemant & ::= & Protected\_Assignment \mid Unprotected\_Assignment \\
Call\_Statement & ::= & Protected\_Call \mid Unprotected\_Call
\end{array}
$$

实现使用的代换语言

之所以要做无保护/受保护的区分，是因为经典数学的算术运算符的性质已经变了，在这里都变成了部分运算（定义域不全了）

无保护的赋值语句有几类：

$$
\begin{array}{lcl}
\mathit{Unprotected\_Assignment} & ::= & \mathit{Identifier\_List} := \mathit{Term\_List} \\
& | & \mathit{Identifier}(\mathit{Term\_List}) := \mathit{Term} \\
& | & \mathit{Identifier} := \mathsf{bool}(\mathit{Condition}) \\
& | & \mathit{Identifier}(\mathit{Term\_List}) := \mathsf{bool}(\mathit{Condition})
\end{array}
$$

第 2 行和第 4 行是数组元素赋值；第 3/4 行右边是从谓词（条件）到布尔值的转换，条件（*Condition*）也作为一个语法范畴

无保护调用也分为几类：

$$
\begin{array}{lcl}
\mathit{Unprotected\_Call} & ::= & \mathit{Identifier\_List} \longleftarrow \mathit{Identifier}(\mathit{Term\_List}) \\
& | & \mathit{Identifier\_List} \longleftarrow \mathit{Identifier} \\
& | & \mathit{Identifier}(\mathit{Term\_List}) \\
& | & \mathit{Identifier}
\end{array}
$$

实现使用的代换语言

上面两类结构里都使用了 *Term*，需要给出它的精确定义

实际上，*Term* 是 B 语言中 *Expression* 的受限形式，它也具有双重意义：

- 作为数学的表达式，它描述的是集合论对象
- 作为程序里的表达式，它提出了要求算出一个值的操作性要求

而一旦这样考虑问题，就会带来一系列无法避免的严重问题

Term 的语法：

$$
\begin{array}{lcl}
\mathit{Term} & ::= & \mathit{Simple\_Term} \\
& | & \mathit{Function\_Constant\_Identifier}(\mathit{Term\_List}) \\
& | & \mathit{Function\_Variable\_Identifier}(\mathit{Term\_List}) \\
& | & \mathit{Term} + \mathit{Term} \mid \mathit{Term} - \mathit{Term} \\
& | & \mathit{Term} * \mathit{Term} \mid \mathit{Term} / \mathit{Term}
\end{array}
$$

实现使用的代换语言

现在要考虑 *Term* 应能翻译到计算机语言而且能由计算机求值，为此就需要仔细处理 *Term* 里面的算术子表达式，因为按照计算机的看法，其中每个算术运算符的意义都出现了新问题，它们的新意义也与在数学里不同

先看直接考虑的 *Simple_Term* 及其成分 *Simple_Constant* 的语法：

$$
\begin{array}{rcl}
\mathit{Simple\_Term} & ::= & \mathit{Simple\_Variable\_Identifier} \\
& | & \mathit{Operation\_Input\_Formal\_Parameter} \\
& | & \mathit{Operation\_Output\_Formal\_Parameter} \\
& | & \mathit{Simple\_Constant} \\
\mathit{Simple\_Constant} & ::= & \mathit{Enumerated\_Set\_Element\_Identifier} \\
& | & \mathit{Scalar\_Constant\_Identifier} \\
& | & \mathit{Numeratic\_Literal}
\end{array}
$$

要考虑实际的计算，就需要把计算中的所有不完全因素考虑进来

实现使用的代换语言

在计算机（和程序语言）里，数据有表示范围，因此每个运算符都是不完全的。这也意味着每个保护数值表达式的 *Assignment_Statemant* 或 *Call_Statemant* 都必须是受保护的（protected）。也就是说，在相应的赋值或调用前都有前条件

$$
\begin{array}{rcl}
\mathit{Protected\_Assignment} & ::= & \textbf{PRE}\ \mathit{Predicate} \\
& & \textbf{THEN}\ \mathit{Unprotected\_Assignment}\ \textbf{END} \\
\mathit{Protected\_Call} & ::= & \textbf{PRE}\ \mathit{Predicate} \\
& & \textbf{THEN}\ \mathit{Unprotected\_Call}\ \textbf{END}
\end{array}
$$

前条件的作用是保证相应算术表达式有明确意义，这一点可以证明

这种前条件可以自动构造，方法是：收集起 *Unprotected_Assignment* 或者 *Unprotected_Call* 里所有的子算术表达式 $\boldsymbol{E}$，做出谓词 $\boldsymbol{E} \in \textbf{INT}$，然后把所有这样的谓词合取起来

注意，**INT** 实际上就是 **MININT..MAXINT**，要证明属于关系，就是证明两个基于比较的条件

实现使用的代换语言

考虑下面的 $Unprotected\_Assignment$：

$$\boldsymbol{a := b + c \times d}$$

很容易得到右部的所有子表达式：$\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{c \times d}$, $\boldsymbol{b + c \times d}$。这样，相应的 $Protected\_Assignment$ 是：

PRE
$\boldsymbol{b} \in \mathbf{INT} \wedge$
$\boldsymbol{c} \in \mathbf{INT} \wedge$
$\boldsymbol{d} \in \mathbf{INT} \wedge$
$\boldsymbol{c \times d} \in \mathbf{INT} \wedge$
$\boldsymbol{b + c \times d} \in \mathbf{INT} \wedge$
THEN
$\boldsymbol{a := b + c \times d}$
END

从无保护赋值或操作调用到受保护的版本的转换是机械的，可以用一个简单程序自动完成。B 工具会自动做这件事情

实现使用的代换语言

前面几种语言结构里牵涉到 $Condition$，其语法如下

$$\begin{array}{rcl} Condition & ::= & Simple\_Term = Simple\_Term \\ & | & Simple\_Term \neq Simple\_Term \\ & | & Simple\_Term > Simple\_Term \\ & | & Simple\_Term \geq Simple\_Term \\ & | & Simple\_Term < Simple\_Term \\ & | & Simple\_Term \leq Simple\_Term \\ & | & Condition \wedge Condition \\ & | & Condition \vee Condition \\ & | & \neg Condition \end{array}$$

$Condition$ 里不包含 $Term$，因此所有 $Condition$ 都不是受保护的

关于 Atelier B 中 B0 语言的具体情况，可以看语言手册 7.25 节。下面说明其中的一些重要情况

Atelier B 的 B0 语言

Atelier B 的 B0 语言对于 B 语言（按《B Book》）有些修订。下面先分类说明一些特殊情况，而后特别介绍 Atelier B 的局部操作。下面将直接说 B0

数组：

按照 B 语言的规定，数组的类型是从 **INT** 到数组元素值的类型的映射的类型。如果元素类型为 T，数组类型就是 $\mathbf{INT} \times T$。显然这个类型过于宽泛，与实际程序语言里相应的类型规定不一致

B0 规定两个数组类型兼容的条件是具有相同下标集合。例如，虽然 $A1 \in 1..10 \rightarrow \mathbf{INT}$ 和 $A2 \in 2..11 \rightarrow \mathbf{INT}$ 元素个数相同，但它们类型不同

另外，如果数组的界用不同具体常量描述，即使这些常量的值相同，数组类型也不相容。例如 $A3 \in 1..cc1 \rightarrow \mathbf{INT}$ 而 $A4 \in 1..cc2 \rightarrow \mathbf{INT}$ 永远不会类型相容，即使有 $cc1 = cc2$

项（Term）：

项（Term）是 B 语言表达式中可以用于 B0 的特殊子类，用在语句和条件里

Atelier B 的 B0 语言

任何在语句里直接使用的项，都必须证明其确实为定义良好的，能够在常规程序设计语言里正确实现所描述的计算

为此需确认如下证明义务：1) 表达式里的基本数据具有所需类型，如整数类型数据满足 **INT** 的取值范围限制。2) 使用算术运算符必须证明下述条件：

B0 arithmetic expression		Condition		
B0 addition	$a + b$	$a \in \mathrm{INT}$	$\wedge\ b \in \mathrm{INT}$	$\wedge\ a + b \in \mathrm{INT}$
B0 subtraction	$a - b$	$a \in \mathrm{INT}$	$\wedge\ b \in \mathrm{INT}$	$\wedge\ a - b \in \mathrm{INT}$
B0 unary minus	$-a$	$a \in \mathrm{INT}$		$\wedge\ -a \in \mathrm{INT}$
B0 multiplication	$a \times b$	$a \in \mathrm{INT}$	$\wedge\ b \in \mathrm{INT}$	$\wedge\ a \in b \in \mathrm{INT}$
B0 integer division	a / b	$a \in \mathrm{INT}$	$\wedge\ b \in \mathrm{INT} - \{0\}$	$\wedge\ a / b \in \mathrm{INT}$
B0 modulo	$a \bmod b$	$a \in \mathrm{NAT}$	$\wedge\ b \in \mathrm{NAT}_1$	$\wedge\ a \bmod b \in \mathrm{INT}$
B0 raise to the power	a^b	$a \in \mathrm{INT}$	$\wedge\ b \in \mathrm{NAT}$	$\wedge\ a^b \in \mathrm{INT}$
B0 successor	$\mathrm{succ}(a)$	$a \in \mathrm{INT}$		$\wedge\ \mathrm{succ}(a) \in \mathrm{INT}$
B0 predecessor	$\mathrm{pred}(a)$	$a \in \mathrm{INT}$		$\wedge\ \mathrm{pred}(a) \in \mathrm{INT}$

Atelier B 的 B0 语言

条件（Condition）：

用在 **IF** 和 **WHILE** 的条件部分

如果相等或不等判断涉及数组，有关数组必须具有前面说的 "相同类型"

语句（在 Atelier B 里称为 Instruction）：

Atelier B 的 B0 增加了一种，允许 **ASSERT** 语句

ASSERT $Predicate$ **THEN** $Statement$ **END**

B0 的规定是，这种语句的 $Predicate$ 并不出现在生成的程序代码里，只是为了做有关 "实现" 的证明。因此这里用 $Predicate$ 而不用 $Condition$

显然，这里的考虑与现在许多编程语言里提供的（或者通过库或编程提供的） assertion 机制的想法不同

对于操作调用语句有如下限制：1) 输入参数或者为项，或者为字符串文字量；2) 如果输入参数是数组，实际参数和形式参数必须具有相同数组类型

CASE 的选择表达式只能是简单项（Simple_Term）

Atelier B 的 B0 语言

对于赋值（:=），它的作用有下面几种不同情况：

- 作用于一个标量数据变量
- 作用于一个数组数据对象。此时要给出所有数组元素的值。产生这种效果的方式是用另一个数组作为赋值操作的右部，或者用一个字面量数组进行赋值，要求数组具有相同类型
- 作用于一个数组项，下标用项表示
- 作用于记录对象的一个数据域

Atelier B 的 LOCAL_OPERATIONS

Atelier B 的一个扩充是增加了一个 **LOCAL_OPERATIONS** 子句，用于描述局部操作。这一子句只能出现在 “实现” 里

在 **LOCAL_OPERATIONS** 子句里声明的操作称为 “局部操作”。局部操作只能在本实现里使用，即只能被本实现里的操作（普通操作或局部操作）调用

在实现的 **LOCAL_OPERATIONS** 子句里写出本实现里的所有局部操作的规范，在 **OPERATIONS** 子句里给出所有局部操作的实现

如果在一个实现的 **OPERATIONS** 里的有一个操作与其 **LOCAL_OPERATIONS** 里的一个局部操作同名，就认为它是该局部操作的实现。**LOCAL_OPERATIONS** 里声明的每个局部操作，都必须在本实现的 **OPERATIONS** 子句里实现

在实现的 **OPERATIONS** 里的操作，必须或者是被它 **REFINES** 的机器里的一个操作，或者是其 **LOCAL_OPERATIONS** 子句里声明的一个操作

局部操作的实现和 **IMPLEMENTATION** 里的一般操作一样：1) 可修改本实现的状态；2) 可有输入和输出参数；3) 只能使用 B0 语言的语句（指令）。但局部操作总是在同一个实现里给出规范和实现，没有精化过程

LOCAL_OPERATIONS

局部操作也带来一些证明义务：

- 需要证明其规范能保证本实现所导入的机器的不变式
- 需要证明某个局部操作的实现（在 **OPERATIONS** 子句里给出）符合其规范（在 **LOCAL_OPERATIONS** 子句里给出）的要求

请特别注意：这里不要求证明局部操作具有不变式保持性。因为局部操作只在一个实现的内部使用。我们只需保证所有非局部操作维持不变式（无论其是不是借助于一些局部操作实现的）

局部操作的主要作用是用于组织 B 实现的描述。可以考虑：

- 把初始化中的复杂动作实现为局部操作，例如要初始化一个数组
- 把一些操作的实现中需要做的公共动作实现为一个局部操作
- 从复杂的操作中分解出一些部分，实现为局部操作

这些情况对应于在常规程序语言里编程中的 “功能分解”。但这里要做的事情和编程中做的有些不同（下面讨论）

LOCAL_OPERATIONS

在 B 语言里，无论何时需要定义一个操作，都必须做两件事情：1) 描述该操作的规范；2) （最终）给出该操作的实现

对局部操作也如此，特殊之处在于规范和实现写在同一实现里：

- 操作的规范写在实现机器的 **LOCAL_OPERATIONS** 部分，应是抽象的高层次的，其中可使用 B 语言的任何代换描述方式。规范的用途有两个方面：
  1. 用于验证局部操作的实现，保证其实现的正确性，即，验证在 **OPERATIONS** 里给出的操作实现满足操作规范的要求
  2. 用于验证使用局部操作的那些操作的正确性

  操作的规范是操作的实现和使用之间的中介，保证使用和定义的一致性
- 操作的实现写在实现机器的 **OPERATIONS** 部分，其中只能使用 B0 语言里的结构。它必须满足相应的操作规范的要求

局部操作可以在实现机器的初始化部分、局部操作或非局部操作里调用

LOCAL_OPERATIONS

右边是 Atelier B 手册里给出的一个简单示例

在局部操作子句里定义了一个求最大值的操作，在 **OPERATIONS** 子句实现了这个操作

局部操作的另外实现方法可以用其他机制实现。Atelier B 手册 168 页介绍了一种方法。可以根据情况考虑其他方法

```
IMPLEMENTATION
    MA_i
...
LOCAL_OPERATIONS
    max_y =
    BEGIN
        x0 := max(y1, y2)
    END
OPERATIONS
    max_y =
    IF y1 ≥ y2 THEN
        x0 := y1
    ELSE
        x0 := y2
    END
    ;
    OpA =
    BEGIN
        ...
        max_y ;
        ...
    END
END
```

B 规范的组织

现在讨论 B 规范的组织和信息共享问题。这部分内容可以参考《B Book》第 12.2 节 “共享” 和 12.4 节 “多重精化和实现”，以及 Atelier B 语言手册的第 8 节 Architecture（体系结构）

基于 B 的一个完整开发在 Atelier B 里就是一个 project（项目），其中用一组 components 建模一个系统，最终的目标是产生一个可执行程序

前面的讨论都是关于如何保证所开发系统的功能安全性（safety），也就是说，保证系统运行中不出现我们不希望出现的状态

现在要研究的是基于 B 方法的项目的概念来构造软件的方法

在基于B 方法的开发中，我们用一个 B 模块模拟软件里的一个子系统，模块由一些 B 组件构成。B 方法里的组件有三种：抽象机，精化和实现

一个 B 模块总有一个抽象机作为它的规范（抽象规范）；它可以有一个实现（机器），还可能有作为其规范和实现之间的桥梁的若干个精化（机器）。进一步说，它还可能有一段代码

B 模块分类

Atelier B 规范提出根据性质把 B 模块分为 3 类：1) 从某个抽象机器出发经过一系列精化开发出来的模块；2) 基本模块；3) 抽象模块。下表是一些情况

性质\模块	开发的模块	基本模块	抽象模块
有无抽象机器	有	有	有
有无实现	有	无	无
有无代码	有	有	无

其中开发的模块的代码通过翻译自动生成，而基本模块的代码由手工写出

下面分门别类讨论几种模块的情况

基本模块:

情况简单，又称基本机器。它们只有一个抽象机。基本模块只能被其他模块导入，作为系统结构图最下面的叶子结点

基本模块必须有关联的代码，这些代码不是由抽象机翻译得到的，而是直接实现的。基本机器通常用作已有代码或某些底层服务的接口，相应功能存在于 B 语言之外。系统提供的输入输出接口函数是典型的例子

B 模块分类

抽象模块:

抽象模块的基本用途是供其他模块（抽象机或精化）**INCLUDES** 以享用其中描述的信息。这种只是作为抽象开发的一种媒介

开发的模块:

这是本课程讨论的主题。首先，有一个抽象机描述了这一模块的规范。这时 B 语言被作为规范语言使用，外部模块只能通过抽象机定义的接口使用这个模块。从外部使用的观点看，B 模块就等于它的抽象机

一部抽象机的精化也是一个部件，它维持了抽象机的接口和行为，但可能重新构造了操作和内部状态，代以更具体的变量。在精化时，抽象机的集合和具体数据都自动保留；其他数据可以精化，即，可以保留、替换或抛弃；也可以引入新的数据。一个精化还可以被下一个精化所精化

一个实现是一个 B 部件，是从一部抽象机出发逐步精化的最后结果，用 B 语言的子集 B0 写出。实现里的数据都能直接对应到实际计算机语言的数据表示，操作体必须用可以直接翻译到高级语言程序的语句（或指令）形式

B 软件项目

一个完整的 B 项目由一集 B 模块实例构成，这些模块实例通过某些连接相互关联成为一体，有关连接要遵循一些规则（下面讨论）

一个模块实例就是某部抽象机的一个拷贝

- 通过实例化，一部抽象机可以在同一项目里多次使用
- 一部抽象机的多个实例具有各自独立的数据空间，这由该抽象机里定义的可修改数据（抽象变量和具体变量）确定
- 抽象机里的常量由抽象机确定，在其所有实例之间共享
- 从一个抽象机实例出发调用操作时，操作中使用的是该实例的变量

需要区分抽象实例和具体实例

- 抽象实例是在规范阶段通过 **INCLUDES** 在抽象数据空间里创建的
- 具体实例是在实现阶段创建的，实际构成了项目开发出的程序的具体数据空间（下面讨论 **IMPORTS** 连接时还会讨论）

抽象机实例

每个抽象机实例都有自己特殊的名字

- 如果没有重命名，实例的名字就是抽象机的名字
- 如果重命名，实例的名字就是重命名前缀加“.”再加抽象机名
- 如果一部抽象机在项目里实例化多次，就必须重命名它们

如果没有重命名，实例的名字就是抽象机的名字（这种情况很常见）。但抽象机实例和抽象机是两种东西，不能搞混了

重命名一个抽象机实例，也就重命名里实例里的所有变量（无论抽象变量或具体变量）和操作，使用时必须采用加重命名前缀的写法

但重命名对集合和常量没有影响，一部抽象机里定义的集合和常量由该抽象机的所有实例共享（无论它们是否重命名）