概要

本节讨论实现中的若干问题：

- 实现中的循环
  - 简单情况
  - 用专门的抽象机实现循环体或循环体中的主要部分
- 集合的实现
- 常量的实现
- 字符串类型 **STRING** 的情况和解释

实现中的循环

前面说过，循环代换 **WHILE** 只能出现在实现里，这也说明

- **WHILE** 体里只能出现可实现的代换
- 而且 **WHILE** 体里可以操作的状态或者是导入的，或者是循环体里局部的
- 要使用导入状态，也就意味着循环里主要是写导入操作的调用

导入机器的状态是完全隐藏的，对导入状态的操作都只能通过相关操作

- 直接通过导入机器的操作改变导入机器的状态
- 通过查询操作取得有关状态的信息，保存在局部变量里

实现中的循环

下面看一个简单例子

抽象机 Mult 实现一个累积的乘积

```
MACHINE Mult
VARIABLES prod
INVARIANT prod : NAT1
INITIALISATION prod := 1
OPERATIONS
    mult(num) =
      PRE num : NAT1
      THEN prod := prod * num
      END;

    val <-- value = val := prod
END
```

下面做出它的一个实现，其中用了一个只使用加法的循环实现乘法

实现中的循环

```
IMPLEMENTATION Mult_Imp0
REFINES Mult
CONCRETE_VARIABLES concprod
INVARIANT concprod : NAT1 & concprod = prod
INITIALISATION concprod := 1
OPERATIONS
   mult ( num ) =
     VAR temp, cnt IN
       temp := concprod; cnt := 1;
       WHILE cnt < num
       DO  temp := temp + concprod; cnt := cnt + 1
       INVARIANT temp = concprod * cnt & cnt : NAT & cnt <= num
       VARIANT num - cnt END;
       concprod := temp
     END;
   val <-- value = val := concprod
END
```


循环体的分离

如果循环体很复杂，可以考虑专门做一个机器来描述循环体里的操作：

- 定义一个导入机器，其中只定义一个外部操作，用作循环体
- 这种做法，可以为循环体需要做什么提供一个规范

用一个独立的机器描述循环体的概念，一个重要优势是使循环的特定实现可以与它的规范分离，这样就可以分别独立考虑循环体的精化和实现

如果用一个独立机器描述循环体并 **IMPORTS** 它来实现循环，这个循环的不变式（出现在实现机器里）就需要关注这个导入机器的状态里的变量

根据循环在实现机器的操作中的位置，与循环有关的证明义务会出现在与精化关系有关的证明义务（关联抽象变量和具体变量）里面

对循环的要求（规范）是在抽象操作里描述的（与相应循环在操作中出现的位置有关），因此循环不变式就需要描述导入状态（循环体机器）和被精化机器的状态之间的关系

循环体的分离

重新考虑前面的 Mult 实例，同样考虑用反复做加法来代替乘法，但用一个循环体机器描述循环体完成的工作的主要部分

先定义一个循环体机器：

```
MACHINE Body
VARIABLES accum
INVARIANT accum : NAT
INITIALISATION accum := 1
OPERATIONS
    val <-- value = val := accum;
    cnt <-- body (val, cnt0) =
    PRE val : NAT & cnt0 : NAT
    THEN accum := accum + val || cnt := cnt0 + 1
    END
END
```

这一机器完成一次累加，并通过返回值返回计数器变量的新值（原值加一）

循环体的分离

基于抽象机 Body 的 Mult 实现：

```
IMPLEMENTATION Mult_Imp
REFINES Mult
IMPORTS Body
PROMOTES value
INVARIANT accum = prod
OPERATIONS
   mult ( num ) =
   VAR cnt, oldv IN
       cnt := 1; oldv <-- value;
       WHILE cnt < num
       DO cnt <-- body(oldv, cnt)
       INVARIANT accum = cnt * prod & oldv = prod &
                 cnt : NAT & cnt <= num
       VARIANT num - cnt END
   END
END
```

证明义务的情况

下面是两种实现的一些情况：

Component	PO	Proved	Unproved
Mult	4	3	1
Mult_Imp0	25	21	4
Body	4	3	1
Mult_Imp	10	9	1

未能证明的几个 PO 主要是有关乘法和加法越界的问题。还有个别很简单的算术表达式简化问题，明显为正确的

对这个具体例子，再加上 Body 的实现，要证明的 PO 实际差不多

操作 `mult` 的实现的证明义务是

$$\boldsymbol{I} \wedge \boldsymbol{J} \wedge \boldsymbol{P} \Rightarrow [\boldsymbol{T}]\neg[\boldsymbol{S}]\neg\boldsymbol{J}$$

其中

$$\begin{array}{rcl} \boldsymbol{J} & = & \texttt{accum = prod} \\ \boldsymbol{S} & = & \texttt{prod := prod * num} \end{array}$$

证明义务的情况

所以

$$\neg[\boldsymbol{S}]\neg\boldsymbol{J} \;=\; \texttt{accum = prod * num}$$

$\boldsymbol{T}$ 是 `mult` 的实现的操作体，于是（$\boldsymbol{Loop}$ 表示循环代换）：

$$[\boldsymbol{T}]\neg[\boldsymbol{S}]\neg\boldsymbol{J} \;=\; \forall \texttt{cnt}, \texttt{oldv} \cdot [\texttt{cnt := 1; oldv <-- value}]([\boldsymbol{Loop}])(\texttt{accum = prod * num})$$

$\boldsymbol{Loop}$ 的 **INVARIANT** 谓词是

```
accum = cnt * prod & oldv = prod & cnt : NAT & cnt <= num
```

它与循环条件 `cnt < num` 的否定显然蕴涵后条件

很容易看出 $\boldsymbol{I} \wedge \boldsymbol{J} \wedge \boldsymbol{P}$ 蕴涵循环的不变式。最后还需证明这个 **WHILE** 循环的 **INVARIANT** 谓词确实是这个循环的不变式

这些证明从略。完成了这些证明后，就可以断言 `Mult_Imp` 确实是 `Mult` 的一个实现

Body 的实现

下面是 Body 的一个实现：

```
IMPLEMENTATION Body_I
REFINES Body
CONCRETE_VARIABLES accum
INVARIANT accum : NAT1
INITIALISATION accum := 1
OPERATIONS
    val <-- value = val := accum;
    cnt <-- body ( val , cnt0 ) =
    BEGIN
        accum := accum + val;
        cnt := cnt0 + 1
    END
END
```

这里用同名的具体变量作为 Body 里抽象变量 accum 的实现，不需要再写它们之间的连接不变式了，它们自动相等

城际交通实例的实现

第 9 次课讨论过一个城际交通的实例，其中先定义了抽象机 Cities，后来做了三次精化，得到精化机器 Cities_RRR。这个抽象机的静态部分是

```
REFINEMENT
   Cities_RRR(CITY)
REFINES
   CitiesRefR
VARIABLES parent, num
INVARIANT parent : CITY --> CITY & num : NAT &
    rep = iterate(parent, num) &
    !ct.(ct : CITY => parent(rep(ct)) = rep(ct))
INITIALISATION num := 0 || parent := id(CITY)
```

其中

- iterate(parent, num) 表示 parent 关系的 num 次复合
- rep(city) 得到 city 的代表元城市
- parent 初始化为 CITY 集合上的恒等关系

城际交通实例的实现

操作定义如下：

```
OPERATIONS
    link (ct1, ct2) =
      VAR rep1, rep2
      IN
        rep1 := iterate(parent, num)(ct1) ;
        rep2 := iterate(parent, num)(ct2) ;
        IF rep1 /= rep2
        THEN parent(rep1) := rep2; num := num + 1 END
      END;
    ans <-- connectedQ (ct1, ct2) =
      IF iterate(parent, num)(ct1) = iterate(parent, num)(ct2)
      THEN ans := connected
      ELSE ans := notconnected
      END
END
```

问题是如何实际计算出迭代关系 `iterate(parent, num)`，需要用循环

城际交通实例的实现

应注意到，这里需要有一种方式确定通过 `parent` 关系前进多少步就保证能到达代表元。现在的想法是用一个计数器抽象机记录这一情况。上次课定义的 `Counter` 可以满足这里的需要

```
MACHINE Counter
VARIABLES counter
INVARIANT counter : NAT
INITIALISATION counter := 0
OPERATIONS
    num <-- number = num := counter;
    setzero = counter := 0;
    inc = counter := counter + 1;
    dec = PRE counter > 0 THEN counter := counter - 1 END
END
```

实际上，这里只需要取出计数值和计数值加一两个操作，其余操作不用

形式化方法的支持系统也可以考虑常规编程语言的 “程序库连接” 技术，在从规范生成程序时，只生成正在开发的系统里用到的部分

城际交通实例的实现

考虑用一个抽象机表示一集元素之间的映射，提供两个操作：get(x) 取得 x 映射到的元素，set(x,y) 令 x 映射到 y

定义下面抽象机：

```
MACHINE Garray(ITEM)
VARIABLES garray
INVARIANT garray : ITEM --> ITEM
INITIALISATION garray := id(ITEM)
OPERATIONS
    set(nd, nnd) =
      PRE nd : ITEM & nnd : ITEM THEN garray(nd) := nnd END;
    nnd <-- get(nd) =
      PRE nd : ITEM THEN nnd := garray(nd) END
END
```


城际交通实例的实现

Cities_RRR 的实现 Cities_RRR_Imp 导入抽象机 Counter 和 Garray，其不变式连接被它实现的 Cities_RRR 以及两个导入抽象机的状态：

```
IMPLEMENTATION
  Cities_RRR_Imp(CITY)
REFINES
  Cities_RRR
IMPORTS Garray(CITY), Counter
INVARIANT
  garray = parent & counter = num
OPERATIONS
  link ( ct1 , ct2 ) =
  VAR rep1, rep2, ii, cnt
  IN
    ii := 0;
    cnt <-- read;
    rep1 := ct1;
    rep2 := ct2;
```

```
    WHILE ii < cnt
    DO rep1 <-- get(rep1); rep2 <-- get(rep2); ii := ii + 1
    INVARIANT
      iterate(parent, num-ii)(rep1) = iterate(parent,num)(ct1) &
      iterate(parent, num-ii)(rep2) = iterate(parent,num)(ct2) &
      ii : NAT & ii <= num & rep1 : CITY & rep2 : CITY &
      garray = parent & cnt = counter & counter = num
    VARIANT num - cnt END;
    IF rep1 /= rep2 THEN set(rep1, rep2); inc END
  END;

  ans <-- connectedQ (ct1, ct2) =
  VAR ... ... /* 前面部分与 link 一样 */
    ... ...
    IF rep1 = rep2 THEN ans := connected
    ELSE ans := notconnected END
  END
END
```


城际交通实例的实现

由于操作 `link` 和 `connectedQ` 中有很大一部分是共同，可以考虑将这一部分定义为一个局部操作

Atelier B 局部操作功能很受限制。根据 Atelier B 手册，局部操作规范的代换里不能访问被精化规范里的常量和变量（C.8），只能在 **ASSERT** 结构里引用，写起来比较麻烦。但这个操作还是可以定义的

在 **LOCAL_OPERATIONS** 部分只描述抽象规范

```
LOCAL_OPERATIONS
    rep1, rep2 <-- rep_cities( ct1 , ct2 ) =
    PRE ct1 : CITY & ct2 : CITY & rep1 : CITY & rep2 : CITY
    THEN ANY r1, r2 WHERE r1 : CITY & r2 : CITY
        THEN ASSERT r1 = iterate(parent, num)(ct1) &
                    r2 = iterate(parent, num)(ct2)
            THEN rep1 := r1 || rep2 := r2 END
        END
    END
```

```
OPERATIONS
    rep1, rep2 <-- rep_cities(ct1, ct2) =
      VAR rep01, rep02, ii, cnt
      IN ii := 0;
        cnt <-- read;
        rep01 <-- get(ct1);
        rep02 <-- get(ct2);
        WHILE ii < cnt
        DO rep01 <-- get(rep01);
           rep02 <-- get(rep02);
           ii := ii + 1
        INVARIANT
           iterate(parent, num - ii)(rep01) =
                iterate(parent, num)(ct1) &
           iterate(parent, num - ii)(rep02) =
                iterate(parent, num)(ct2) &
           ii : NAT & ii <= num & rep01 : CITY & rep02 : CITY &
           garray = parent & cnt = counter & counter = num
        VARIANT num - cnt END;
```



```
        rep1 := rep01;
        rep2 := rep02
      END;

    link ( ct1 , ct2 ) =
      VAR rep1, rep2
      IN rep1, rep2 <-- rep_cities(ct1, ct2);
        IF rep1 /= rep2 THEN set(rep1, rep2); inc END
      END;

    ans <-- connectedQ (ct1, ct2) =
      VAR rep1, rep2
      IN  rep1, rep2 <-- rep_cities(ct1, ct2);
        IF rep1 = rep2
        THEN ans := connected
        ELSE ans := notconnected END
      END
END
```

有关这些实现的证明，与前面情况一样，不重复了

SETS 和 CONSTANTS 的实现

现在讨论一些与集合和常量的实现有关的问题

在实现抽象机时，不仅要实现其状态（变量）、初始化和操作，还需实现抽象机的集合和常量（说明它们怎样实现）

集合和常量是在开发中的某个点上引入的，实现它们需要在开发中的某个地方（抽象机、精化或实现）给出它们的定义

特别的，在实现中，所有被它精化的抽象机或精化机器里引入的待定集合，都必须在这个实现的 **VALUES** 子句里给定值

抽象机 **SETS** 引入的集合有几种情况：

- 枚举集合，它们已经实例化
- 已通过 **PROPERTIES** 子句里的谓词基于本机器的其他集合定义
- 完全没有约束的待定集合，要求在开发的后续阶段明确定义

枚举集合已经有了实现它的所有信息，不需要再考虑

SETS 的实现

对于待定集合，开发的后来阶段必须给予进一步的信息

例如下面抽象机里定义了两个待定集合：

```
MACHINE Name
SETS LENGTH; AREA
... ...
END
```

对这样的定义，LENGTH 和 AREA 被看作互不相同的新集合（也是类型）。它们可能出现在证明义务里，作为不同类型看待

可说明变量或常量的类型为 LENGTH，这种变量（常量）不能与 AREA 类型的变量（常量）相互操作（类型检查和证明义务保证不出这种问题）

这两个类型后来可能都实现为 **INT** 的子集。例如：

```
IMPLEMENTATION Name_I
REFINES Name
VALUES LENGTH = 1..100; AREA = 1..10000
... ...
END
```

SETS 的实现

这一实现给代码生成提供了有关这两种类型的足够信息，工具系统可以直接将其翻译到具体的实现语言（常规的高级语言）

引进上述两个待定集合（而且在开发前期不给予定义）是很好的设计决策。假如把 LENGTH 和 AREA 类型的对象直接定义为 **INT** 或 **NAT**，工具将无法区分它们，在消解证明义务时也不可能检查发现使用混乱的情况

（出现这种现象，实际上是 B 语言的类型系统比较弱。它采用的是类型的结构等价原则。一些程序设计语言采用 "名字等价"，其类型系统就可以静态检查这类错误。B 开发时并没有充分注意类型理论的成果）

B 方法只允许待定集合最终实现为整数区间，这使待定集合的使用受到极大限制，实际上很可能需要将其实现为一个记录类型，或实现为一个笛卡儿积类似（B-Tool 工具支持笛卡儿积类型）

当然，这里需要研究允许怎样的实现不会破坏已有的开发和 PO 证明，又能提供最大的灵活性，支持更广泛的开发需要

CONSTANTS 的实现

CONSTANTS 子句用于为抽象机引入各种常量 （"值"），这些值可以是标量，也可以是更复杂的实体，例如集合或者函数

与 **SETS** 子句引入的集合不同，即使一个常量的值是集合，它也不是新的类型，需要在 **PROPERTIES** 子句里给它们定类型，还可给定更确定的取值。所有常量的类型必须是抽象机里已有的类型

常量也需要给定取值，可以是在定义它们的抽象机的 **PROPERTIES** 子句里，或者是在后续开发中给定取值。在

但并不是所有常量都需要给定明确的实现，完全可能有些常量被操作的实现吸收进去了。尤其是一些复杂的常量，最后可能是由操作实现的

此时，操作实现和相关事宜的验证要保证操作实现的正确性，也就是说，它能满足抽象机的 **PROPERTIES** 对于相关常量性质的描述

这种不实现的常量只起规范的作用，应定义为 **ABSTRACT_CONSTANTS**，它们不出现在最终代码里，前面定义的许多复杂常量可能归于这一类

CONSTANTS 的实现

考虑一个酒店房间的例子。房间分为两类（用枚举集合 SIZE 描述）

- 标准间类别为 std，其房价为默认价
- 豪华间类别为 lux，其房价为默认价的 3/2 倍

下面抽象机里定义了几个常量

- default 是自然数类型的标量常量，表示默认房价
- standard 是标准间的集合
- price 是表示房价，它将每个房间映射到其房价

后两个常量都不是简单常量。我们并不准备实现它们，只准备在规范里描述相应性质，因此将它们定义为 **ABSTRACT_CONSTANTS**

各种常量的性质都要在抽象机的 **PROPERTIES** 子句里说明。下面给出抽象机 Rooms


```
MACHINE Rooms
SETS ROOM; SIZE = {std, lux}
CONSTANTS default
ABSTRACT_CONSTANTS standard, price
PROPERTIES
    default : NAT1 & standard <: ROOM & price : ROOM --> NAT1 &
    price[ standard ] <: {default} &
    price[ ROOM - standard ] <: {default * 3 / 2}
OPERATIONS
    sz <-- sizequery (rm) =
      PRE rm : ROOM
      THEN IF rm : standard
          THEN sz := std
          ELSE sz := lux END
      END;
    pr <-- pricequery(rm) =
      PRE rm : ROOM THEN pr := price(rm) END
END
```

注意其中的 **PROPERTIES** 子句很复杂，描述了几个常量的性质

CONSTANTS 的实现

这一抽象机没有状态，只是提供了几个常量和两个操作

- `sizequery` 查房间的大小
- `pricequery` 查房间的标价

下面考虑这一抽象机的实现。有下面考虑：

- 房间集合用编号集合实现，假设共有 128 和房间
- 房间的默认价为 160 元。这两项都应在实现的 **VALUES** 子句描述
- 两个抽象常量不再实现，通过操作隐含
- 假定单号房间都为标准间，双号为豪华间


```
IMPLEMENTATION Rooms_I
REFINES Rooms
VALUES default = 160; ROOM = 1..128
PROPERTIES standard = {nn | nn : ROOM & nn mod 2 = 1} &
  !rm .(rm : ROOM & rm mod 2 = 1 => rm : standard) &
  !rm .(rm : ROOM => (rm mod 2 = 1 => price(rm) = default)
             & (rm mod 2 /= 1 => price(rm) = default*3/2))
OPERATIONS
  sz <-- sizequery ( rm ) =
    VAR st IN
      st := rm mod 2;
      IF st = 1 THEN sz := std ELSE sz := lux END
    END ;
  pr <-- pricequery ( rm ) =
    VAR st IN
      st := rm mod 2;
      IF st = 1 THEN pr := default ELSE pr := default * 3 / 2 END
    END
END
```

26 个 PO 证明了 23 个。剩下三个里两个很简单（算术证明能力问题）：

```
rm mod 2 <= 2147483647
```

另一个很复杂，整理如下

```
"‘Check that the property
(  !rm.(rm: ROOM => (rm mod 2 = 1 => price(rm) = default) &
                    (rm mod 2 /= 1 => price(rm) = default*3/2))
 & price: ROOM --> NAT1 & price[standard] <: {default}
 & price[ROOM - standard] <: {default*3/2})
   is preserved by the valuations - ref 5.2’"  =>
#price.
  (  !rm.(rm: 1..128 => (rm mod 2 = 1 => price(rm) = 160) &
                        (not(rm mod 2 = 1) => price(rm) = 240))
   & price: 1..128 +-> NAT - {0} & dom(price) = 1..128
   & (not(price[{nn | nn: 1..128 & nn mod 2 = 1}] = {}) =>
      price[{nn | nn: 1..128 & nn mod 2 = 1}] = {160})
   & (not(price[(1..128) - {nn | nn: 1..128 & nn mod 2 = 1}] = {})
      =>
      price[(1..128)-{nn | nn: 1..128 & nn mod 2 = 1}] = {240}))
```


STRING 类型的使用

在 Atelier B 里，STRING 是一个基本类型。但它不能用在大多数其他类型可以使用的地方。目前只能用用在操作的前条件里，为函数的输入参数定类

操作调用时可以给一个字符串文字量作为其参数，这种参数又可以传给其他以字符串为参数的操作

下面是两个抽象机：

```
MACHINE
    UseStr
OPERATIONS
    inpara(s1) = PRE s1 : STRING THEN skip END;
END

MACHINE testStr USES UseStr OPERATIONS
    test(s2) = PRE s2 : STRING THEN inpara(s2) END
END
```

STRING 类型的使用

字符串不是原来 B 语言的类型。在一些 B 系统里提供了实现字符串类型的库。Atelier B 把 **STRING** 作为基本类型，但

- 没提供任何操作
- 不能定义以具体的 **STRING** 为值的常量
- 不能作为状态变量或局部变量的类型（不能赋值）
- 不能作为操作的返回值

Atelier B 有关 **STRING** 使用的设想是

- 只能在具体的操作调用时通过字符串文字量描述
- 可以在操作体内用这种变量去调用其他操作
- 这种传来传去没办法使用。Atelier B 的设想是最终把这样的字符串传到某个能处理 **STRING** 的基本模块，由那里具体使用字符串

现在的系统没有库，无法具体使用 **STRING**。可以将使用封装在抽象机里