形 式 化 方 法:

基于 B 方法的严格软件开发

(10)

精化和证明义务

裘宗燕

北京大学数学学院信息科学系

2010年春季

概要

本次课首先考虑精化中非确定性的消除，然后研究精化的证明义务

这里主要讨论初始化和操作里的非确定性

- 同一操作，具有更少的非确定性，就是原来的操作的精化
- 操作精化通常在同样的状态空间里进行，精化后的操作具有较少的非确定性，复杂操作的精化也可能通过多步完成，最终达到确定性操作

后一部分研究精化的理论，研究为保证正确精化所需的证明义务：

- 新的初始化带来的证明义务
- 只修改状态的操作带来的证明义务
- 产生输出的操作带来的证明义务

证明了所有这些证明义务，也就能保证做出的新的精化机器确实是原来的抽象机的精化，原规范已经证明的性质现在仍能保持

非确定性及其精化问题

前面介绍了非确定性的概念，以及 B 方法描述中的非确定性结构

在软件的系统设计阶段，对不能清晰做出抉择的多种可能处理方式，应该留下变化的可能性，供后来开发中根据实际情况做出更好的选择

在写抽象规范时也是这样，因此 B 规范里可以写出多种非确定性结构

应该看到，顺序程序语言里没有非确定性结果，规范中的非确定性最终都要消去。这也是精化阶段的一项工作

选择适当方式消去规范里的非确定性，是重要的精化任务。本节课先讨论这个方面，提出一些认识和做法，用实例说明要考虑的问题和可能处理方法

除数据表示方面的问题外，非确定性的典型情况出现在抽象机的初始化子句或操作的定义里，其表现形式是一个代换表示多种不同的可能行为

所谓消除非确定性，就是用只有一种行为的代换取而代之

实际上，消除非确定性的工作也可以一步步完成，用具有较少非确定性的动作取代具有较多非确定性的动作，通过若干步最终达到确定性行为

消去非确定性

精化有非确定性的机器，基本方法是在从它的多种可能行为中选择一种，在精化机器里实现这种特定行为。也就是说，在精化中说明怎样实际选择

例如：设有一个操作，其输出或对状态的修改用 $x :\in S$ 描述 （从集合 S 里任选元素）。精化它的操作可以在 S 里实际选一个元素，如选最小元素（如果 S 的元素可以比较大小），用 $x := \min(S)$ 精化上述行为

在前面讨论数据精化的例子里，精化机器的状态变量都与原机器不同，需要用连接不变式建立两者的联系。这种精化里也可能出现实际上减少了非确定性的情况（如原机器的许多状态对应精化机器的一个状态）

下面讨论另一情况：一步精化中的状态变量不变，这时原机器的变量和精化机器的同样变量之间实际上有很强的连接：要求它们取值总相同

这时：

- 非确定性的消去是在精化后的状态空间，不是在原机器的状态空间
- 是否消去了某些非确定性，要考虑两个状态空间之间的连接

实例：电话号码分配机器的规范

考虑一个实例，描述一个分配电话号码的机器，它：

- 维持一个已分配电话号码集 $allocated$，它是非负自然数的子集
- 遇到新分配需求，任意分配一个不属于 $allocated$ 的自然数
- 还支持选择特定号码（前提是原来未分配）和查询某号码是否已分配

机器的状态部分很简单：

MACHINE
$Allocate$
VARIABLES $allocated$
INVARIANT $allocated \subseteq \mathbb{N}_1$
INITIALISATION $allocated := \varnothing$

几个操作的定义见下页

电话号码分配机器：规范

OPERATIONS
$choose(num) =$
PRE $num \in \mathbb{N}_1 \wedge num \notin allocated$
THEN $allocated := allocated \cup \{num\}$
END;

$num \longleftarrow allocate =$
ANY nm **WHERE** $nm \in \mathbb{N}_1 - allocated$
THEN $num := nm \parallel allocated := allocated \cup \{nm\}$
END;

$ans \longleftarrow query(num) =$
PRE $num \in \mathbb{N}_1$
THEN $ans :=$ bool$(num \in allocated)$
END
END

几个操作的定义都很简单

关键的非确定性出现在操作 $allocate$，其中新号码选择是非确定的

电话号码分配机器：精化

现在考虑上述抽象机的精化，基本考虑是消除其中的非确定性

如何消除非确定性是一个设计选择，可以考虑用任何确定性的方式代替原有的非确定性，当然要保证功能“正确”（任何满足规范的确定性的方式，都是这种非确定性选择的精化。“功能正确”的严格意义下面讨论）

下面的考虑是每次任意选择时，总选未分配号码中最小的一个

下面精化机器的状态空间与原机器完全一样，重要的精化是操作的定义

REFINEMENT
$Allocate\_R$
REFINES
$Allocate$
VARIABLES
$allocated$
INITIALISATION
$allocated := \varnothing$

电话号码分配机器：精化

操作 $choose$ 和 $query$ 的定义与原机器完全一样

OPERATIONS
$choose(num) = allocated := allocated \cup \{num\};$
$num \longleftarrow allocate =$
BEGIN
$num := \mathsf{min}(\mathbb{N}_1 - allocated);$
$allocated := allocated \cup \{num\}$
END;
$ans \longleftarrow query(num) =$
$ans := \mathsf{bool}(num \in allocated)$
END

操作 $allocate$ 重新定义。现在用了一个顺序复合代换，其中第一个代换取得未分配的最小号码，第二个代换修改集合 $allocated$

精化和连接不变式

前面说，连接不变式关联起抽象机器和精化机器的状态。一般说这是一种“多对多”关系，也就是说多个抽象状态对应着多个精化后的状态

- 一个抽象状态对应一些精化后的状态
- 一个精化后的状态对应一些抽象状态

这是最一般的情况。具体连接不变式产生的关系完全可以不是“多对多”

如果抽象机器和精化后的机器里出现相同的变量，连接不变式就要求同一变量在两个机器里的值一样（不用明确写出来）

这样，精化机器要合法，其初始化和操作都应在抽象机器允许的范围内：

- 精化机器的初始化产生的状态要满足抽象规范的不变式
- 精化机器的操作产生的输出必须是抽象机器允许的输出
- 精化机器的操作能达到的状态必须也是抽象机器能够达到的

精化和非确定性

如果一个抽象操作具有非确定性，实际上它就有多个可能的执行。这个操作的规范规定了它有哪些合法的不同执行情况

精化机器里的同一操作不必也同样产生所有这些可能执行情况，消除非确定性的意思就是在这些可能执行中做选择

- 一方面，精化机器里的操作可以减少选择的可能性
- 但相应精化后的操作的每个可能执行情况，都必须是原规范允许的

以上面电话号码分配机器 $Allocate$ 和 $Allocate\_R$ 为例，假设现在机器的状态是 $allocated = \mathbf{\{6, 13\}}$，现在调用 $allocate$ 操作

- 抽象机器 $Allocate$ 可能选择除 $\mathbf{6, 13}$ 之外的任何自然数
- 精化机器 $Allocate\_R$ 将分配电话号码 $\mathbf{1}$，这是抽象规范允许的

一方面，这一精化大大消减了原机器的非确定性（实际上，这个具体例子是把一个非确定性操作精化为一个确定性操作）

另一方面，精化后的操作这时绝不能分配 $\mathbf{6}$ 或 $\mathbf{13}$，因为原规范禁止

非确定性的重新安排

精化是两个机器之间作为整体的一种关系，包括

- 两者的状态之间的精化关系
- 两者的操作所引起的状态变化方式之间的关系

上述两方面各自满足了一些条件，才能说一个机器是另一个的精化。当然还要提出最基本的要求：精化机器的行为满足抽象机器的规范

也就是说，只要行为满足要求又消除了一些非确定性，就是原机器的精化。消除非确定性的具体方式是什么并没有关系

（如前面所说，消除非确定性的方式是设计中的重要选择。实际中人们还可以追求更有效、更高效、更合理、更 ... 的方式。这些东西在实际中都非常重要，但已不在功能正确的范畴内了）

上面讨论说明一个情况：抽象规范里的非确定性，完全可以以另一种形式出现在精化机器里（例如，更容易消除的形式）

当然还要求：精化机器的状态转移总能与原抽象机器的状态转移相互匹配

非确定性的重新安排

一般说，精化机器仍然可以保留一些非确定性，情况：

- 其操作仍可以有多个可能执行和效果
- 但其每个执行都应对应于抽象机器的一个或一些执行
- 其所有可能行为都是抽象规范允许的行为

考虑描述一个有关图书阅读的机器，它维持尚未阅读的书籍的记录

1. 基本书籍的集合 $BOOK$ 由参数确定
2. 机器维护的集合 $\boldsymbol{read}$ 表示已经阅读的书籍，其初始值为空集
3. 操作 $\boldsymbol{newbook}$ 每次给出一本尚未阅读的书籍

$\boldsymbol{newbook}$ 给出书籍的方式是非确定性的，从未读过的书中任选一本

虽然这里存在非确定性，但这一规范把系统的基本特征描述的非常清楚：需要维持什么样的阅读状态，什么样的书籍可以推荐等等

阅读记录和推荐机器

下面是描述书籍记录推荐系统的抽象规范：

MACHINE $\boldsymbol{Books}(BOOK)$
VARIABLES $read$
INVARIANT $read \subseteq BOOK$
INITIALISATION $read := \varnothing$
OPERATIONS
  $\boldsymbol{bk} \longleftarrow \boldsymbol{newbook} =$
    PRE $read \neq BOOK$
    THEN
      ANY $\boldsymbol{bk1}$ **WHERE** $\boldsymbol{bk1} \in BOOK - read$
      THEN $\boldsymbol{bk} := \boldsymbol{bk1} \parallel read := read \cup \{\boldsymbol{bk1}\}$
      END
    END
END

由于存在非确定性，这一规范距离可实现的系统还远，需要进一步精化

精化中的主要工作就是消除规范里存在的非确定性

阅读记录和推荐机器

下面考虑用一个变量显式表示未读书籍，然后考虑从这一变量中选取元素

- 如果用集合表示，选元素操作只能是任意选取，无法消除非确定性
- 解决问题的一种方法是用有序结构代替集合，用一个按某种顺序列出书籍的列表记录这些书籍
- 下面用一个 iseq 序列 $scheme$ 记录这些书籍

注意：变量 $scheme$ 是一个单射序列，说明序列里的对象没有重复。这里表示尚未阅读过的每本书只在序列里出现一次

考虑连接不变式，应该是

$$\mathsf{ran}(scheme) = BOOK - read$$

也就是说，未读过的书籍都在序列里。加上前面单射序列的条件，可以保证每本这样的书籍在序列里恰好出现一次

写出的精化规范见下页

阅读记录和推荐机器：精化

注意用同样的参数（Atelier B 的要求，B Book 没说明）

REFINEMENT
  $\boldsymbol{Books\_R}(BOOK)$
REFINES
  $\boldsymbol{Books}$
VARIABLES $scheme$
INVARIANT $scheme \in \mathsf{iseq}(BOOK) \wedge \mathsf{ran}(scheme) = BOOK - read$
INITIALISATION $scheme :\in \mathsf{perm}(BOOK)$
OPERATIONS
  $\boldsymbol{bk \longleftarrow newbook} =$
    BEGIN $\boldsymbol{bk} := \mathsf{first}(scheme) \parallel scheme := \mathsf{tail}(scheme)$ **END**
END

初始时 $read$ 为空，$scheme$ 里包含所有书籍。随着阅读的进展，这些书被按照它们在 $scheme$ 里排列的顺序一本本取出

$\mathsf{perm}(\boldsymbol{s})$ 得到 $\boldsymbol{s}$ 的排列的集合，在一个排列中 $\boldsymbol{s}$ 的每个元素出现且仅出现一次，序列变量 $scheme$ 取 $BOOK$ 的元素的一个排列作为值

阅读记录和推荐机器：精化

这样得到的精化机器仍然不是确定性的，序列 $scheme$ 的初始化为非确定性的，它被初始化为 $BOOK$ 里所有书籍的某个排列序列，具体是哪个排列并没有确定。而与之对应的 $read$ 初始化是确定性的：

$$read := \varnothing$$

即使如此，这一精化仍是合适的，因为无论 $scheme$ 被初始化为 $BOOK$ 的哪个排列，连接不变式

$$\mathsf{ran}(scheme) = BOOK - read$$

都成立。也就是说，无论精化机器里非确定性初始化得到哪个初始化状态，它都关联于原机器里的抽象状态 $read = \varnothing$

出现这种非确定性也不奇怪：精化机器里的信息表示更为具体，因此对于同样的信息，可能存在多种不同的表示形式

$scheme$ 的值确定了，随后的状态变化都是确定性的：每次取出哪本书，序列（系统的状态）如何更新，都已经确定了。$\boldsymbol{newbook}$ 变的非常简单

阅读记录和推荐机器：精化

在精化前的抽象机器里，非确定性出现在每次调用 $newbook$ 的时候，这时做出如何选一本书的决策

而在精化后的新机器里，处理非确定性的决策出现在系统的初始化动作中。也就是说，非确定性被做了重新安排 （在这个具体例子里，好像是有先知，事先知道后来一系列选择的结果）

但从用户的角度，这种内部实现方式的变化是完全感觉不到的，看不到实际做出选择的时间—是在初始化时做的还是在工作过程中

有交换图

$$\begin{array}{ccc} read = \varnothing & \longrightarrow & scheme = [x, ...] \\ \downarrow newbook_a & & \downarrow newbook_r \\ read = \{x\} & \longrightarrow & scheme = [...] \end{array}$$

（这里只是作为例子，是否应该把每次做选择的活动提前到系统的初始化操作里，是具体的设计问题。并不是一般性地倡导这种做法）

阅读记录和推荐机器：精化

上述精化机器还可以进一步精化，考虑

- 用数组代替序列，用数组里的元素位置分隔已读过的书和未读的书
- 取下一本书和更新剩余图书集合就是简单更新指示数组元素的位置变量

基于这些想法，定义的精化机器的静态部分如下：

REFINEMENT $Books\_R\_R(BOOK)$
REFINES $Books\_R$
VARIABLES $counter, bookarr$
INVARIANT
$counter \in 0..\mathsf{card}(BOOK) \land$
$bookarr \in 1..\mathsf{card}(BOOK) \rightarrowtail\!\!\!\!\!\!\rightarrow BOOK \land$
$(0..counter \lhd bookarr)\hat{}\,scheme = bookarr$
INITIALISATION
$counter := 0 \;||$
$bookarr :\in 1..\mathsf{card}(BOOK) \rightarrowtail\!\!\!\!\!\!\rightarrow BOOK$

$bookarr$ 是一个单且满的全函数，初始化仍然是非确定性的

阅读记录和推荐机器：精化

newbook 操作非常简单：

OPERATIONS
  $bk \longleftarrow newbook =$
    BEGIN
      $counter := counter + 1;$
      $bk := bookarr(counter)$
    END
END

注意这里的连接不变式

$$(\mathbf{0..}counter \lhd bookarr)\hat{}scheme = bookarr$$

它说明 $scheme$ 总等于去掉前 $counter$ 个元素的数组形成的序列

初始化将 $counter$ 设置为 $\mathbf{0}$ 建立起这一连接不变式，此时

$$scheme = bookarr$$

newbook 操作也会维持这个不变式

精化的操作和前条件

考虑操作 ***newbook***，在几个机器里有它的三个不同版本

注意，在精化机器里（无论是第一个还是第二个），操作都是在原抽象机器里 ***newbook*** 的前条件下执行的。如果前条件为假，$\neg(read \neq BOOK)$ 意味着所有书籍都已读完。在这种情况下

- 在第一个精化里，出现的情况是序列 $scheme$ 为空序列，对这一序列执行 first 或 tail 操作都会出错
- 对第二个精化，这时 $counter$ 大于数组的最大下标 $\mathsf{card}(BOOK)$，导致 $bookarr(counter)$ 无定义，和它所关联的 $scheme$ 无法进行操作一致

前条件告诉使用者，所有书籍全读完后，就不能再来调用 ***newbook***，或者说，保证了用户不会在书全读完后再调用这个操作

因此，在两个精化里都不必考虑这一前条件不成立的情况

也就是说，操作的前条件总被精化机器的同一操作继承，不需要重新写

实例：故事盒系统的精化

考虑前面的故事盒系统，其中记录一集故事，积分，和选出可播放的故事集

系统的操作包括

- 家长操作：奖励积分和惩罚（减分或删去故事）
- 儿童操作：选故事和播放故事

规范如下（变量名改了，用 $sset$ 而不是原来的 $slist$）：

MACHINE $StBox$
SETS $STORY$
CONSTANTS max_score
PROPERTIES $max\_score \in \mathbb{N}_1$
VARIABLES $score, sset$
INVARIANT $score \in \mathbb{N} \wedge score \leq max\_score \wedge sset : \mathbb{P}(STORY)$
INITIALISATION $score := 0 \;||\; sset := \{\}$

OPERATIONS
$\boldsymbol{gives(sc)} =$
PRE $sc \in \mathbb{N}_1$ **THEN** $score := \min(\{score + sc, max\_score\})$ **END;**
$\boldsymbol{penalty} =$
SELECT $score > 0$ **THEN** $score := score - 1$
WHEN $sset \neq \{\}$ **THEN**
ANY st **WHERE** $st \in sset$
THEN $sset := sset - \{st\}$ **END**
ELSE skip **END;**
$\boldsymbol{select(st)} =$
PRE $score > 0 \wedge st \in STORY$
THEN
CHOICE $score := score - 1 \;||\; sset := sset \cup \{st\}$
OR $sset := sset \cup \{st\}$ **END**
END;
$\boldsymbol{st \longleftarrow tell} =$
IF $sset \neq \{\}$ **THEN**
ANY $st1$ **WHERE** $st1 \in sset$
THEN $st := st1 \;||\; sset := sset - \{st1\}$ **END**
END
END

故事盒系统的精化

这一规范里有许多需要精化的方面：

- 首先，系统的状态是用集合表示的，需要用更接近计算机的结构取代
- 下一个播放的故事是非确定性的，需要消除
- 用 *select* 选取故事时，可能扣分或不扣分，需要确定一个规则
- 惩罚操作可能扣分也可能删掉一个故事，也为非确定性的操作

原机器的变量 *score* 并不需要变（不必考虑其他方式表示这一信息），但在精化机器里用另一取自然数值的变量 $scoreR$ 代替

已选故事的集合 *sset* 应考虑数据精化。由于后来需要取其中元素，如果放入集合的故事后来要取出，选取元素的操作必定带来非确定性。下面也用一个序列 （*slist*）取代它，采用先进先出的方式保存和选取其中元素

要想消除惩罚操作的非确定性，需要规定惩罚的策略。下面采取一种简单策略，如果有已选的故事就删去一个故事，否则扣1分

故事盒系统的精化

要消除选故事操作 *select* 里的非确定性（非确定地扣分或不扣分），需要设计一种策略。下面考虑的策略是：

- 采用循环计数的方式，一个周期可能有一次选取不扣分
- 用常量 *free* 表示周期长度
- 用变量 *isfree* 计数，当它的值到达 *free* 可能免除扣分一次，否则就扣分

REFINEMENT $\boldsymbol{SBox\_R}$
REFINES $\boldsymbol{SBox}$
CONSTANTS $free$
PROPERTIES $free \in \mathbb{N}_1$
VARIABLES $scoreR, slist, isfree$
INVARIANT
$scoreR \in \mathbb{N} \wedge scoreR = score \wedge$
$slist \in \mathsf{iseq}(STORY) \wedge \mathsf{ran}(slist) = sset \wedge$
$isfree \in \mathbf{0}..free$
INITIALISATION $scoreR := 0 \parallel slist := [\,] \parallel isfree := 0$

故事盒系统的精化

OPERATIONS

$\boldsymbol{gives(sc)} =$
  BEGIN $scoreR$ **:=** min($\{scoreR + \boldsymbol{sc}, max\_score\}$) **END;**
$\boldsymbol{st \longleftarrow tell} =$
  BEGIN $\boldsymbol{st}$ **:=** first($slist$) || $slist$ **:=** tail($slist$) **END;**
$\boldsymbol{penalty} =$
  IF $slist \neq [\,]$ **THEN** $slist$ **:=** tail($slist$)
  ELSIF $\boldsymbol{scoreR} > 0$ **THEN** $scoreR$ **:=** $scoreR - 1$ **END;**
$\boldsymbol{select(st)} =$
  BEGIN
    IF $\boldsymbol{st} \notin$ ran($slist$) **THEN** $slist$ **:=** $slist \leftarrow \boldsymbol{st}$ **END ;**
    IF $isfree = free$ **THEN**
      CHOICE $isfree$ **:=** $\mathbf{0}$
      OR $isfree$ **:=** $\mathbf{0}$ || $scoreR$ **:=** $scoreR - 1$ **END**
    ELSE $isfree$ **:=** $isfree + 1$; $scoreR$ **:=** $scoreR - 1$ **END**
  END
END

故事盒系统的精化

这一精化机器还有一些非确定性，包括：

- 常量 *free* 的值没有确定，这导致多少次扣分后可能出现一次免扣分的周期长度没有确定。这个值的确定推迟到系统开发的后期
- 当计数值达到 *free* 时，这里是 "可能" 免分。也可以考虑用更确定性的操作代替它，例如，把最后那个 **IF** 结构写成：

  IF $isfree = free$
  THEN $isfree$ **:=** $\mathbf{0}$
  ELSE $isfree$ **:=** $isfree + 1$; $scoreR$ **:=** $scoreR - 1$ **END**

  这样，选择 *free* 次就必然有一次免积分

精化的总结

上次课和这次课的前半部分讨论了抽象机的精化，有关的情况

- 在 B 方法里做精化的方式是定义精化机器，首先要说明它是哪个抽象机器的精化。写一个精化描述实际上也是定义一个抽象机
- 精化关系是相对的，被精化的机器更 “抽象"，精化机器更 “具体"，更接近实现。一般而言精化形成一条精化链，从一个初始的抽象机定义开始
- 精化机器需要定义一集变量表示自己的状态，并用连接不变式建立其状态与它所精化的原抽象机的状态之间的联系
- 一种特殊情况是精化机器具有与原抽象机一样的变量，这时不写连接不变式，实际上要求两个状态里的变量值相同
- 对抽象机器的每个操作，精化要定义一个同名、输入输出参数相同的操作
  - 该操作完成的工作应该是原操作的精化（常常消除一些非确定性）
  - 这些操作自动继承原机器中对应操作的前条件（不必重新写）
- 精化中最常见的工作是：数据精化，用更接近实现的数据结构取代抽象的；减少或消除非确定性，特别是初始化和操作的非确定性

精化的证明义务

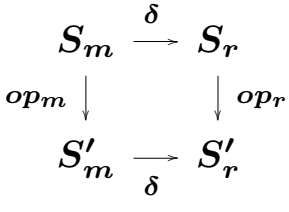
定义好所需的抽象机，用不变式描述抽象机的不变性质，证明其初始化能建立不变式，所有操作都维持不变式，这是软件形式化开发的一步

随后精化阶段，针对已有的抽象机定义精化抽象机。要保证定义出的机器确实是原机器的精化，也要通过严格的形式化证明

给定一个抽象机 M，如希望针对 M 定义的精化抽象机 R 确实是 M 的精化，也必须确认一组证明义务。证明了的义务被称为 “精化定理"

先直观分析需要证明什么。显然要考虑

- 精化机器的状态与原机器的状态正确对应
- 精化机器的初始化动作产生的状态对应于原机器的初始状态
- 精化机器的操作产生的状态转移，对应于原机器的状态转移



两边的状态靠连接不变式 “δ" 描述的状态关系相互关联，见图

抽象机 Color

下面用一个简单的颜色抽象机的开发作为实例，讨论精化的证明义务

这一抽象机非常简单：

MACHINE $Colors$
SETS $COLOR = \{red, green, blue\}$
VARIABLES $colors$
INVARIANT $colors \subseteq COLOR$
INITIALISATION $colors :\in \mathbb{P}(COLOR - \{blue\})$
OPERATIONS
$add(cl) =$ **PRE** $cl \in COLOR$ **THEN** $colors := colors \cup \{cl\}$ **END;**
$cc \longleftarrow query =$ **PRE** $colors \neq \varnothing$ **THEN** $cc :\in colors$ **END;**
$change =$
LET $clss$ **BE** $clss = \mathbb{P}(COLOR) - \{colors\}$
IN $colors :\in clss$ **END**
END

抽象机 Color

抽象机的状态变量 $colors$ 是颜色集合 $COLOR$ 的子集

抽象机将 $colors$ 初始化为 $\{red, green\}$

操作 add 给颜色集合变量 $colors$ "加" 一个颜色

操作 $query$ 任意取出 $colors$ 里的一个颜色

注意在 $change$ 操作的定义里，**LET** 的局部约束

$$clss = \mathbb{P}(COLOR) - \{colors\}$$

使 $clss$ 成为 $COLOR$ 的（除 $colors$ 之外的）所有子集的集合，因此 $change$ 操作改变了 $colors$ 的值（换了一个颜色集合）

精化这个抽象机，可以有许多不同的方式。下面的想法是做数据精化

- 考虑用一个颜色变量取代这里的颜色集合
- 在颜色变量里保存的必须是原抽象机器的颜色集 $colors$ 里的一个元素，调用 $query$ 取元素时就可以直接取这个变量的值

抽象机 Color 的精化

如果做出的数据精化是正确的，就能保证 $\boldsymbol{change}$ 操作和 $\boldsymbol{add}$ 操作之后都能正确维持这个颜色变量的值。也就是说，保证在这些操作后，通过 $\boldsymbol{query}$ 取出的值（该颜色变量的值）一定是原集合 $colors$ 的元素

下面是根据上面想法做出的精化：

REFINEMENT $\boldsymbol{Colors\_R}$
REFINES $\boldsymbol{Colors}$
VARIABLES $color$
INVARIANT $color \in COLOR \wedge color \in colors$
INITIALISATION $color :\in COLOR - \{\boldsymbol{blue}\}$
OPERATIONS
$\boldsymbol{add(cl)} =$ **BEGIN** $color :\in \{color, \boldsymbol{cl}\}$ **END;**
$\boldsymbol{cc} \longleftarrow \boldsymbol{query} = \boldsymbol{cc} := color$;
$\boldsymbol{change} =$ **LET** $\boldsymbol{cs}$ **BE** $\boldsymbol{cs} = COLOR - \{color\}$
IN $color :\in \boldsymbol{cs}$ **END**
END

抽象机 Color 的精化

先直观地分析一下两个修改状态的操作

- 考虑操作 $\boldsymbol{add}$

  $\boldsymbol{add(cl)} =$ **BEGIN** $color :\in \{color, \boldsymbol{cl}\}$ **END**

  如果在操作 $\boldsymbol{add}$ 前变量 $color$ 的值属于原抽象机的集合 $colors$，经过原来的 $\boldsymbol{add}$ 操作后要加入的元素和元素 $color$ 都属于 $colors$，新操作修改后的状态能保证 关系 $color : colors$ 成立

- 再看操作 $\boldsymbol{change}$

  $\boldsymbol{change} =$ **LET** $\boldsymbol{cs}$ **BE** $\boldsymbol{cs} = COLOR - \{color\}$
  IN $color :\in \boldsymbol{cs}$ **END**

  如果操作 $\boldsymbol{change}$ 前 $color$ 属于原抽象机的集合 $colors$，那么 $\boldsymbol{cs}$ 就是一个与 $colors$ 不同的集合，$color$ 就是一个与 $colors$ 不同的集合的元素

精化机器的初始化非常简单，直观看起来它能满足要求

下面的工作是要把这些直观的非形式化的看法严格说清楚

精化关系

在下面的讨论中（关于初始化和关于操作），我们都先抽象地讨论问题，然后再用前面机器的精化作为具体示例

首先讨论初始化的问题，这里考虑：

- 抽象机器 M 的初始化代换是 $\boldsymbol{T}$，它建立不变式 $\boldsymbol{I}$
- 精化机器 R 的初始化代换是 $\boldsymbol{T_1}$，而连接不变式是 $\boldsymbol{J}$（这里把 R 的整个不变式看作连接不变式，说起来简单，也合理）
- 根据精化的要求，我们希望新的初始化代换 $\boldsymbol{T_1}$ 能以某种与抽象机器的初始化 $\boldsymbol{T}$ 协调的方式建立不变式 $\boldsymbol{J}$

一般而言，抽象机器和精化机器的代换 $\boldsymbol{T}$ 和 $\boldsymbol{T_1}$ 可能都有非确定性，因此可能允许多种不同的执行，给出多个不同的状态

要保证 R 是 M 的精化，就要求 $\boldsymbol{T_1}$ 的每个可能执行都与 $\boldsymbol{T}$ 的某个执行匹配，也就是说，由 $\boldsymbol{T_1}$ 的任何一个执行产生的状态，都（通过连接不变式 $\boldsymbol{J}$）对应于 $\boldsymbol{T}$ 能建立的某个状态（没有超出允许的范围）

精化关系和初始化

现在考虑如何严格地形式化地描述上面的认识

考虑用下面方式描述抽象机器 M 与其精化机器 R 之间的关系

- 做一个组合 “机器”，其状态由 M 的状态 $\boldsymbol{s}$ 和 R 的状态 $\boldsymbol{s'}$ 两个独立部分组成。用 $(\boldsymbol{s}, \boldsymbol{s'})$ 表示这种组合状态
- M 和 R 在这种状态上一起执行，R 先执行 M 后执行（初始化/动作）
- 可以认为连接不变式 $\boldsymbol{J}$ 联系起这一组合机器的两部分状态，定义了两个状态空间之间的一个关系 $\boldsymbol{J}(\boldsymbol{x}, \boldsymbol{x'}) \in \mathsf{M}_{\boldsymbol{s}} \leftrightarrow \mathsf{R}_{\boldsymbol{s}}$
- 对于这个 “机器” 而言，合法的状态就是即满足 $\boldsymbol{I}$（只与状态中的 M 部分有关），又满足 $\boldsymbol{J}$（与两个部分都有关）的状态

要使精化机器 R 是抽象机器 M 的正确精化，要求：如果 R 的初始化 $\boldsymbol{T_1}$（只与组合状态的 R 部分有关） 建立起 R 的某个状态 $\boldsymbol{s'}$，那么一定存在 M 的 $\boldsymbol{T}$ 可以建立的状态 $\boldsymbol{s}$，使连接不变式 $\boldsymbol{J}(\boldsymbol{s}, \boldsymbol{s'})$ 成立

连接不变式 $\boldsymbol{J}$ 是有关抽象状态和精化状态的谓词，说存在 $\boldsymbol{T}$ 的变换使 $\boldsymbol{J}$ 成立，也就是说，“不是 $\boldsymbol{T}$ 的所有变换都使 $\boldsymbol{J}$ 为假”

初始化的精化

把 "不是 $\boldsymbol{T}$ 的所有变换都使 $\boldsymbol{J}$ 为假" 严格表述出来，就是

$$\neg[\boldsymbol{T}]\neg\boldsymbol{J}$$

这里

- $\neg\boldsymbol{J}$ 说 $\boldsymbol{J}$ 为假
- $[\boldsymbol{T}]\neg\boldsymbol{J}$ 说 $\boldsymbol{T}$ 表示的每个转换之后都保证 $\boldsymbol{J}$ 一定假
- $\neg[\boldsymbol{T}]\neg\boldsymbol{J}$ 说不是上面情况，即存在 $\boldsymbol{T}$ 表示的某些转换，使转换后 $\boldsymbol{J}$ 真

（还可能出现 $\boldsymbol{T}$ 表示的某些转换不终止的情况，后面讨论）

作为谓词，$\neg[\boldsymbol{T}]\neg\boldsymbol{J}$ 对一个状态为真，当且仅当 $\boldsymbol{T}$ 表示的某些转换能建立这个满足 $\boldsymbol{J}$ 的状态，这并不要求 $\boldsymbol{T}$ 建立的所有状态都满足 $\boldsymbol{J}$

要使 R 的初始化代换 $\boldsymbol{T}_1$ 是 $\boldsymbol{T}$ 的精化，就要求对 $\boldsymbol{T}_1$ 建立的每个状态，谓词 $\neg[\boldsymbol{T}]\neg\boldsymbol{J}$ 都为真，即是说，$\boldsymbol{T}_1$ 创建的状态 $\neg[\boldsymbol{T}]\neg\boldsymbol{J}$ 都成立。即要求验证

$$[\boldsymbol{T}_1]\neg[\boldsymbol{T}]\neg\boldsymbol{J}$$

初始化的精化

考虑前面的抽象机器 $\boldsymbol{Colors}$ 和精化机器 $\boldsymbol{ColorsR}$，连接不变式 $\boldsymbol{J}$ 是

$$color \in colors$$

（去掉类型部分是为简单）显然它只对一些抽象状态和精化状态 "对" 成立

抽象机器的初始化代换是

$$\boldsymbol{T} \quad = \quad colors :\in \mathbb{P}(COLOR - \{\boldsymbol{blue}\})$$

前面说，满足 $[\boldsymbol{T}]\neg\boldsymbol{J}$ 的状态，也就是那些无法通过 $\boldsymbol{J}$ 关联于 $\boldsymbol{T}$ 的可达状态的状态。看这个具体例子里的情况

如 $color = \boldsymbol{blue}$ 就是这样的状态，$[\boldsymbol{T}]\neg\boldsymbol{J}$ 对它为真，也就是说，对于 $\boldsymbol{T}$ 可能建立起来的任何 $colors$ 集合，都不可能使 $color = \boldsymbol{blue}$ 成立

另一方面，如果 $color = \boldsymbol{red}$ 或者 $color = \boldsymbol{green}$，都总存在 $\boldsymbol{T}$ 建立的某些状态使 $[\boldsymbol{T}]\neg\boldsymbol{J}$ 不成立，因此 $\neg[\boldsymbol{T}]\neg\boldsymbol{J}$ 成立

初始化的精化

下面是对应于上面非形式化讨论的严格推导：

$$\begin{aligned}
& \neg[colors :\in \mathbb{P}(COLOR - \{\boldsymbol{blue}\})]\neg(color \in colors) \\
= \;& \neg(\forall colors \cdot (colors \in \mathbb{P}(COLOR - \{\boldsymbol{blue}\}) \Rightarrow \neg(color \in colors))) \\
= \;& \exists colors \cdot \neg(colors \in \mathbb{P}(COLOR - \{\boldsymbol{blue}\}) \Rightarrow \neg(color \in colors)) \\
= \;& \exists colors \cdot (colors \subseteq (COLOR - \{\boldsymbol{blue}\}) \wedge color \in colors)
\end{aligned}$$

最后公式说：存在 $COLOR - \{\boldsymbol{blue}\}$ 的子集 $colors$ 使 $color \in colors$，显然，当且仅当 $color \neq \boldsymbol{blue}$ 时这个谓词成立

也就是说，上述谓词等价于

$$color \neq \boldsymbol{blue}$$

现在考虑初始化的精化关系的证明义务

$$[\boldsymbol{T}_1]\neg[\boldsymbol{T}]\neg\boldsymbol{J}$$

在这个例子里的具体情况（下页）

初始化的精化

精化机器 $\boldsymbol{ColorsR}$ 的初始化代换是

$$\boldsymbol{T}_1 \;=\; color :\in COLOR - \{\boldsymbol{blue}\}$$

因此有下面推导：

$$\begin{aligned}
& [\boldsymbol{T}_1]\neg[\boldsymbol{T}]\neg\boldsymbol{J} \\
= \;& [\boldsymbol{T}_1](color \neq \boldsymbol{blue}) \\
= \;& [color :\in COLOR - \{\boldsymbol{blue}\}](color \neq \boldsymbol{blue}) \\
= \;& \forall color \cdot (color \in (COLOR - \{\boldsymbol{blue}\}) \Rightarrow color \neq \boldsymbol{blue}) \\
= \;& \boldsymbol{true}
\end{aligned}$$

这证明了精化机器 $\boldsymbol{ColorsR}$ 的初始化代换确实是抽象机器 $\boldsymbol{Colors}$ 的精化

有人可能会问：为什么这一证明义务里没有提到抽象机器的不变式 $\boldsymbol{I}$?

回答是：在构造该抽象机器时已证明过它的初始化代换 $\boldsymbol{T}$ 建立的状态都满足不变式 $\boldsymbol{I}$，因此 $\boldsymbol{I}$ 的信息已经蕴涵在代换 $\boldsymbol{T}$ 里了

精化与终止性

前面说 $\neg[T]\neg J$ 为真当且仅当 T 不保证总能建立 $\neg J$。出现这种情况，实际上有两种可能性

- 存在 T 的某个（某些）执行，当其结束时 J 成立；或者
- 存在 T 的某个（某些）执行，它（它们）根本不终止（不结束）

从实践的角度看，我们只会去做定义良好的抽象机的精化，那么其中的 T 也应该是定义良好的。抽象机定义良好的一个证明义务是

$$[T]\ I$$

只要验证了这一证明义务，就保证了 T 一定终止

在上述前提下，证明了 $\neg[T]\neg J$，也就保证了精化机器的初始化代换确实是抽象机器的初始化代换的精化

对于下面要考虑的操作的精化问题，情况也与此类似，就不再重复讨论了

操作的精化

现在考虑操作的精化。先考虑没有输出的操作。考虑抽象机器 M 和精化机器 R 的同一个操作，假定其定义分别是：

$$\textsf{PRE } P \textsf{ THEN } S \textsf{ END}$$

$$\textsf{PRE } P_1 \textsf{ THEN } S_1 \textsf{ END}$$

实际中精化机器里的操作经常没有 P_1 部分，因为抽象操作已将执行操作前条件完全描述清楚了。后面会讨论一下出现 P_1 的情况

与初始化的情况类似，这里也是要证明，对于 S_1 的每个执行，都有 S 的某个执行与之匹配。也就是说，要证明 $[S_1]\neg[S]\neg J$

但操作与初始化有不同的地方，操作是从合法的机器状态出发进行的，这一已有的合法状态需要作为证明的前提

现在可以依赖的情况是抽象机器的状态和精化机器的状态满足连接不变式 J，而且抽象机器的状态本身满足不变式 I，组合 “机器” 的状态满足 $I \wedge J$。在加上代换是在前条件成立的情况下进行，就得到证明义务

$$I \wedge J \wedge P \Rightarrow [S_1]\neg[S]\neg J$$

操作的精化

在证明义务 $\boldsymbol{I} \wedge \boldsymbol{J} \wedge \boldsymbol{P} \Rightarrow [\boldsymbol{S}_1]\neg[\boldsymbol{S}]\neg\boldsymbol{J}$ 里，谓词 $[\boldsymbol{S}_1]\neg[\boldsymbol{S}]\neg\boldsymbol{J}$

- 关注抽象机器和精化机器的状态，它对一些状态对成立，对另一些不成立
- 说的更清楚一些：
  - 对于那样的状态对，从它们出发的每个 $\boldsymbol{S}_1$ 执行，都存在某个 $\boldsymbol{S}$ 的执行与之匹配，$[\boldsymbol{S}_1]\neg[\boldsymbol{S}]\neg\boldsymbol{J}$ 对这种状态对一定为真
  - 对于那样的状态对，存在一个或一些从它们出发 $\boldsymbol{S}_1$ 的执行，所有 $\boldsymbol{S}$ 的执行都不与之匹配，$[\boldsymbol{S}_1]\neg[\boldsymbol{S}]\neg\boldsymbol{J}$ 对这种状态对一定为假

考虑抽象机器 $\boldsymbol{Colors}$ 和精化机器 $\boldsymbol{ColorsR}$，其 $\boldsymbol{I} \wedge \boldsymbol{J}$ 是

$$colors \subseteq COLOR \wedge color \in COLOR \wedge color \in colors$$

考虑操作 $\boldsymbol{add}$，其在抽象机器里的定义是

$$\boldsymbol{add}(cl) = \textbf{PRE } cl \in COLOR \textbf{ THEN } colors := colors \cup \{cl\} \textbf{ END}$$

在精化机器里的定义是

$$\boldsymbol{add}(cl) = \textbf{BEGIN } color :\in \{color, cl\} \textbf{ END}$$

操作的精化

这时 $\boldsymbol{I} \wedge \boldsymbol{J} \wedge \boldsymbol{P}$ 是

$$colors \subseteq COLOR \wedge color \in COLOR \wedge color \in colors \wedge cl \in COLOR$$

下面是 $\boldsymbol{add}$ 操作精化的证明：

$$\begin{aligned}
& [\boldsymbol{S}_1]\neg[\boldsymbol{S}]\neg\boldsymbol{J} \\
= \;& [\boldsymbol{S}_1]\neg[\boldsymbol{S}]\neg(color \in colors) \\
= \;& [\boldsymbol{S}_1]\neg[colors := colors \cup \{cl\}]\neg(color \in colors) \\
= \;& [\boldsymbol{S}_1]\neg(\neg(color \in colors \cup \{cl\})) \\
= \;& [\boldsymbol{S}_1](color \in colors \cup \{cl\}) \\
= \;& [color :\in \{color, cl\}](color \in colors \cup \{cl\}) \\
= \;& \forall \boldsymbol{x} \cdot (\boldsymbol{x} \in \{color, cl\} \Rightarrow color \in colors \cup \{cl\}) \\
= \;& (color \in colors \cup \{cl\}) \wedge (cl \in colors \cup \{cl\}) \\
= \;& color \in colors \cup \{cl\}
\end{aligned}$$

显然 $\boldsymbol{I} \wedge \boldsymbol{J} \wedge \boldsymbol{P} \Rightarrow [\boldsymbol{S}_1]\neg[\boldsymbol{S}]\neg\boldsymbol{J}$，因为 $\boldsymbol{J} = color \in colors$ 且

$$color \in colors \Rightarrow color \in colors \cup \{cl\}$$

操作的精化

现在考虑抽象机器 M 和精化机器 R 的同一个操作的一般性情况，假定两个定义分别如下，而且精化机器的操作有明确写出的 P_1：

PRE P **THEN** S **END**

PRE P_1 **THEN** S_1 **END**

有时写出与 P 不同的 P_1 也是有用的。例如这一精化机器还要进一步精化，如果有了适当的 P_1，下一步精化时就用它，而不会继续回溯到 P

但这里的 P_1 不能随便写，由于抽象机器 M 里已明确写出了操作的前条件 P，而 R 的相应操作是在该前条件下执行的

因此 P_1 不能排除任何满足 P 的操作调用情况（满足 P 的调用，都必须允许）。也就是说，如果 P 成立则 P_1 也必须成立

这里还要考虑抽象机器的不变式和连接不变式，对 P_1 的要求是：

$$I \wedge J \wedge P \Rightarrow P_1$$

有输出的操作的精化

现在考虑操作带有输出的情况，先分析其精化会提出什么要求

没有输出的精化操作要保证所产生的状态 “正确”，即，它产生的状态能与原操作所在机器的状态正确 “连接”。当操作产生输出时，还需进一步保证给出的输出满足抽象操作的规范的要求

在抽象机器 $Colors$ 和精化机器 $ColorsR$ 的实例里，只有操作 $query$ 产生输出。假如在某种情况下调用精化机器 $ColorsR$ 的 $query$ 操作产生的输出是 $green$，那么就要保证 $green$ 也是抽象机器 $Colors$ 在相应情况下可能产生的输出（注意两操作都可能有非确定性）

要考虑这种操作精化问题的形式化定义，要注意

- 两个操作的输出参数具有同样的名字
- 输出参数不是机器状态的组成部分，连接不变式 J 不会说到它们
- 在这种精化产生的证明义务里，必须说明两操作的输出之间的关系

下面就是要把 “精化操作的输出要满足抽象操作规范” 这件事严格说清楚

有输出的操作的精化

所谓满足抽象操作的规范，也就是说，精化操作的输出参数得到的任何可能的值，都必须对应于抽象操作可能得到的某个值

如果在某种状态下精化操作产生的一个可能的输出是 o，那就要求在抽象机器的对应状态下，抽象操作也存在某个执行，产生的输出也恰好是 o

为了清晰地描述两个操作的不同输出，要设法区分其输出参数的名字。假定要考虑的抽象操作的输出参数是 out，下面要把相应的精化操作的输出改用 out' 表示（这个讨论好像只说明了一个输出参数的情况，实际上完全可以把 out 和 out' 看作两个参数列表，下面的讨论都适用）

为此，需要修改精化操作的输出参数名（原来也是 out），还要把操作体 S_1 里的 out 都换成 out'。用 $S_1[out'/out]$ 表示修改后的代换体

要求输出匹配，实际上就是要求 $out' = out$。对于多个输出参数的情况，就是要求 $out'_1 = out_1 \wedge \ldots \wedge out'_n = out_n$

对输出的要求和对状态的要求一样：对 out' 的某个可能赋值都必须有对应的对 out 的赋值，也就是说，要证明 $[S_1[out'/out]]\neg[S]\neg(out' = out)$

有输出的操作的精化

结合操作的状态维持性和输出的匹配，同样考虑加上可能的前条件 P_1，对于带输出的操作，精化的证明义务如下：

$$I \wedge J \wedge P \quad \Rightarrow \quad [S_1[out'/out]]\neg[S]\neg(J \wedge out' = out)$$
$$I \wedge J \wedge P \quad \Rightarrow \quad P_1$$

考虑抽象机器 $Colors$ 和精化机器 $ColorsR$，其 J 是

$$color \in colors$$

唯一带输出的的操作是 $query$，抽象机器里的定义是：

$$cc \longleftarrow query = \ \textbf{PRE}\ colors \neq \varnothing\ \textbf{THEN}\ cc :\in colors\ \textbf{END}$$

精化机器里的定义是：

$$cc \longleftarrow query = cc := color$$

下面考虑由此产生的证明义务的证明

有输出的操作的精化

有如下推导：

$$
\begin{array}{rl}
 & [\boldsymbol{S_1}[cc'/cc]]\neg[\boldsymbol{S}]\neg(\boldsymbol{J} \wedge cc' = cc) \\
= & [\boldsymbol{S_1}[cc'/cc]]\neg[cc :\in colors]\neg(\boldsymbol{J} \wedge cc' = cc) \\
= & [\boldsymbol{S_1}[cc'/cc]]\neg(\forall cc \cdot (cc \in colors \Rightarrow \neg(\boldsymbol{J} \wedge cc' = cc))) \\
= & [\boldsymbol{S_1}[cc'/cc]]\neg(\forall cc \cdot (\neg(cc \in colors) \vee \neg(\boldsymbol{J} \wedge cc' = cc))) \\
= & [\boldsymbol{S_1}[cc'/cc]](\exists cc \cdot \neg(\neg(cc \in colors) \vee \neg(\boldsymbol{J} \wedge cc' = cc))) \\
= & [\boldsymbol{S_1}[cc'/cc]](\exists cc \cdot (cc \in colors \wedge \boldsymbol{J} \wedge cc' = cc)) \\
= & [cc' := color](\exists cc \cdot (cc \in colors \wedge \boldsymbol{J} \wedge cc' = cc)) \\
= & \boldsymbol{J} \wedge \exists cc \cdot (cc \in colors \wedge color = cc)) \\
\Leftrightarrow & \boldsymbol{J} \wedge color \in colors \\
= & \boldsymbol{J} \wedge \boldsymbol{I}
\end{array}
$$

由此显然有

$$\boldsymbol{J} \wedge \boldsymbol{I} \wedge \boldsymbol{P} \Rightarrow [\boldsymbol{S_1}[cc'/cc]]\neg[\boldsymbol{S}]\neg(\boldsymbol{J} \wedge cc' = cc)$$

精化的证明义务

上面讨论了精化的基本情况，但还有些问题要说明。在精化里

- 可以定义用 **SETS** 和 **CONSTANTS** 定义新的集合和常量
- 可以通过 **PROPERTIES** 描述新的集合和常量的性质
- 但不能引进新的参数，因此精化机器不会有 **CONSTRAINTS**

抽象机器有针对集合和常量的证明义务，集合和常量也出现在初始化和操作描述里，这些东西也是精化机器里的各种描述的上下文信息

总而言之，在解决精化机器的证明义务时，下面信息都可以使用

- 抽象机器里有关参数的 **CONSTRAINTS** 子句
- 抽象机器里有关集合和常量的 **PROPERTIES** 子句
- 精化机器里有关自己定义的集合和常量的 **PROPERTIES** 子句

另一方面，精化机器引入的集合和常量及 **PROPERTIES** 也会带来新的证明义务。下面统一地总结一下精化的证明义务

精化的证明义务

集合和常量 设 St_1, C_1, B_1 是精化机器的集合、常量和 PROPERTIES，而 St, C, B, Cn 是它精化的原机器的集合、常量、 PROPERTIES 和 CONSTRAINTS，相关的证明义务是

$$Cn \Rightarrow \exists St_1, C_1, St, C \cdot B_1 \wedge B$$

初始化 设 T_1 和 J 是精化机器的初始化代换和不变式，T 是被它精化的原机器的初始化代换，相应的证明义务是

$$Cn \wedge B \wedge B_1 \Rightarrow [T_1]\neg[T]\neg J$$

操作 设精化机器和被它精化的原机器的一对对应操作是

$$out \longleftarrow op(in) = \text{PRE } P \text{ THEN } S \text{ END}$$

$$out \longleftarrow op(in) = \text{PRE } P_1 \text{ THEN } S_1 \text{ END}$$

与之相关的证明义务是

$$Cn \wedge B \wedge B_1 \wedge I \wedge J \wedge P \Rightarrow [S_1[out'/out]]\neg[S]\neg(out' = out)$$

$$Cn \wedge B \wedge B_1 \wedge I \wedge J \wedge P \Rightarrow P_1$$

总结

本章讨论了精化问题的进一步问题，以及精化的理论，包括：

- 精化机器与被它精化的抽象机器的关系，这里的最关键结构是连接不变式
- 通过精化减少或者消除非确定性。应关注：
  - 非确定性的一些情况
  - 如何在精化中消除操作的非确定性
  - 数据精化和操作精化的相互关系
- 精化带来的证明义务，基本考虑是保证精化机器的行为满足原抽象机器规范，状态有正确的对应，产生的输出符合要求，为此要验证一批证明义务
  - 精化机器的初始化和各种操作都带来一些证明义务
  - 带输出的操作，其输出要特别处理，因为连接不变式并不涉及输出
- 精化机器本身的常量和集合定义也可能带来一些证明义务