第 6 章　递归类型

6.1 引　　言

在某些程序设计语言中，类型可以递归地定义。例如 ML，它有递归的 datatype 声明。递归类型是类型等式系统的解，就像递归函数是项等式系统的解一样。例如，自然数表的类型可以想象成类型等式 $t \cong unit + (nat \times t)$的一个解，二叉树的类型可以想象成类型等式 $t \cong unit + (t \times t)$的一个解。“≅”表示一个类型等式系统的解是要使等式两边的类型表达式同构，而不是要使它们相等。

归纳类型和余归纳类型是两类重要的递归类型。归纳类型对应到类型同构等式的最小解，也叫做初始解；余归纳类型对应到它们的最大解，也叫做终结解。直观地说，归纳类型被看成是包含它们引入形式的“最小”类型；而其消去形式是在这引入形式上的一种递归形式。对偶地，余归纳被看成是和它们的消去形式一致的“最大”类型；而其引入形式是一种在这消去形式需要时提供元素的方法。推动归纳类型研究的一个例子是自然数类型，而推动余归纳类型研究的一个例子是自然数流类型，它们在 6.3 和 6.4 节介绍。

归纳的概念比较熟悉，相应地理解归纳类型也比较容易。而要想理解余归纳类型，则必须首先了解余归纳概念，包括余归纳定义和余归纳证明原理等，以及它们和归纳的相应概念的对偶关系。对形式受到一定约束的递归类型等式，在不止一个解的情况下，某个初始代数对应它的最小解，而某个终结余代数对应它的最大解。因此还需要了解余代数以及代数和余代数的对偶性。

代数是数学中很好地确立了的一部分，涉及带有运算的集合，这些运算满足一定的性质，这样的集合有群、环、向量空间等等。泛代数则在更高一个抽象层次上研究代数结构，研究它们之间的同态、子代数和同余等。

概括地说，程序处理数据。在上世纪七八十年代就已经逐步清楚的是，这些数据的抽象描述有望保证一个程序不依赖于它所操作数据的具体表示。这样的抽象也方便了正确性证明。这种期望导致了代数方法在计算机科学中的使用，形成称为代数规范或抽象数据类型理论的一个分支。它用代数中熟知的概念和技术来研究计算机科学中使用的数据类型。第 2 章已经介绍了这方面的知识。

通常的代数技术已经展示出在捕捉计算机科学所用数据结构的本质方面是非常有用的。但是，在试图代数地描述计算过程中出现的天性地动态的结构时，却遇到了困难。这样的结构通常包括状态概念，状态可以按不同的方式进行变换。对于这样基于状态的动态系统，形式方法一般利用自动机或迁移系统。上世纪九十年代，对这样基于状态的系统的深刻认识逐步形成，它们不应该描述成代数，而应该是余代数。余代数是代数的对偶，这是本章需要介绍的概念。代数中初始性的对偶性质（即终结性）对余代数来说是关键的，并且这样的终结余代数所需的逻辑推理原理不是归纳，而是余归纳。在计算机科学中，代数方法是从“构造的”角度研究抽象数据类型的语义，而余代数方法是从“观察的”的角度描述诸如对象、自动机、进程、软件构件等基于状态的系统。

本章的主要内容有：

（1）直观地介绍余归纳的定义、余归纳的证明原理和余代数。

（2）形式地介绍递归类型。

（3）形式地介绍归纳类型和余归纳类型。

6.2 归纳和余归纳

6.2.1 余归纳现象

本节先通过例子直观地介绍余归纳的数据、余归纳的函数和余归纳的证明。先介绍余归纳的数据。

归纳法是构造主义的方法，用归纳法定义数据时，可以把它分解成几个基本步骤：基础情况、迭代规则和最小化条件。归纳法从本质上来讲是封闭的。归纳法是定义字符串集合、形式系统和可计算函数等的基础。

例6.1 数据集A上的有限表集可归纳地定义如下：

（1）基础情况：nil是有限表；

（2）迭代规则：如果$a \in A$，并且σ是有限表，则$cons(a, \sigma)$是有限表；

（3）最小化条件：除此之外，有限表集中不含其它元素。

最小化规则是指所定义的集合是满足（1）和（2）两条约束的最小集合，没有任何垃圾在其中。 □

余归纳也分成几个基本步骤：迭代规则和最大化条件。与归纳法相比，余归纳删去了基础情况，修改了迭代规则，称之为循环条件，并用最大化条件替换了最小化条件。

例6.2 数据集A上的无限表集（称为流）可余归纳地定义如下：

（1）迭代规则：如果$a \in A$，并且σ是无限表，则$cons(a, \sigma)$是无限表；

（2）最大化条件：数据集A上的无限表集是满足迭代规则的最大集合。

和归纳不一样，最大化条件表示所有未被（1）排除的元素都被包含在所定义的集合中，即该集合中的任何无限表都可以通过应用规则（1）若干次（可能是无限次）而得到。

有限表和无限表都有观察算子head和运算算子tail。它们满足的等式是

$$head(cons(a, \sigma)) = a$$

$$tail(cons(a, \sigma)) = \sigma$$

这两个算子合在一起又称为有限表和无限表的**解构子**（*destructor*）。 □

交互计算是在两个层次上余归纳的：余归纳定义的静态值域和余归纳定义的动态下一步动作。

现在考虑在余归纳定义的集合上定义函数。在归纳定义的数据上通常定义的是递归函数，在这样的函数中，需要定义它在所有构造子上的值。例如对于有限表集，计算表长的函数length的定义如下：

$$length(nil) = 0$$

$$length(cons(a, \sigma)) = 1 + length(\sigma)$$

在上述等式的左边，构造子出现在所定义的函数“里面”。

在一个函数f的余归纳定义中，需要定义所有解构子在每个$f(x)$上的值。仍以无限表集合为例，如果有函数$f: A \to A$，那么首先可以定义f的一个拓展函数$ext(f)$，它通过把f应用到无限表σ的每个元素而得到新的无限表$ext(f)(\sigma)$，然后需要定义解构子$head$和$tail$应用到无限表$ext(f)(\sigma)$的值：

$$head(ext(f)(\sigma)) = f(head(\sigma))$$

$$tail(ext(f)(\sigma)) = ext(f)(tail(\sigma))$$

在上述等式的左边，所定义的函数出现在解构子的“里面”。

例6.3 仍以无限表为例，假定想定义一个运算odd，它应用到一个无限表上，忽略其所有偶数位置上的元素，将其剩余元素按原来次序形成一个新的无限表。略加思考可得到如下

的定义：

$$head(odd(\sigma)) = head(\sigma)$$

$$tail(odd(\sigma)) = odd(tail(tail(\sigma)))$$

第一个等式说，$odd(\sigma)$的第一个元素是σ的第一个元素；$odd(\sigma)$的第二个元素是$head(tail(odd(\sigma)))$，用等式演算可得：

$$head(tail(odd(\sigma))) = head(odd(tail(tail(\sigma)))) = head(tail(tail(\sigma)))$$

因此$odd(\sigma)$的第二个元素是σ的第三个元素。这正是所需要的。不难证明，对所有的自然数n，$head(tail^{(n)}(odd(\sigma)))$和$head(tail^{(2n)}(\sigma))$一样。

这两个等式是纯数学规范。若要从计算的角度来看这两个等式，则必须用惰性计算的观点。 □

最后考虑余归纳的证明。余归纳定义的数据和函数的某些性质可以用归纳法来证明，例如例6.3中的对所有的自然数n，$head(tail^n(odd(\sigma)))$和$head(tail^{2n}(\sigma))$一样。余归纳证明的一种专用方法基于**互模拟**（*bisimulation*）的概念。互模拟从数学上刻画系统（如对象、进程等）行为等价这个直观概念，它是指两个系统从观测者角度看可以互相模拟对方的行为，即从观测者角度，两个系统对同样的输入会产生同样的输出，系统内部可能有的状态不同是观察不到的。还是以无限表为例，来解释基于互模拟的证明方法。

例6.4 先余归纳地定义函数*even*，它应用到一个无限表上，忽略其所有奇数位置上的元素，将其剩余元素按原来次序形成一个新的无限表。显然*even*可以这样定义：

$$even = odd \circ tail$$

再定义应用到两个无限表σ和τ的归并函数*merge*。*merge*依次从σ和τ轮流取元素，形成一个新的无限表。同样，*merge*的余归纳定义需要解构子*head*和tail在$merge(\sigma, \tau)$上的结果：

$$head(merge(\sigma, \tau)) = head(\sigma)$$

$$tail(merge(\sigma, \tau)) = merge(\tau, (tail(\sigma))$$

下面基于互模拟证明$merge(odd(\sigma), even(\sigma)) = \sigma$。无限表集合上的互模拟是满足下面条件的二元关系$R$：

若$\langle\sigma, \tau\rangle \in R$，则$head(\sigma) = head(\tau)$并且$\langle tail(\sigma), tail(\tau)\rangle \in R$。

在无限表集合上基于互模拟的余归纳证明原理是：

对互模拟关系R，若$\langle\sigma, \tau\rangle \in R$，则$\sigma = \tau$。

令

$$R = \{\langle merge(odd(\sigma), even(\sigma)), \sigma\rangle \mid \sigma\text{是一个无限表}\}$$

根据该余归纳证明原理，为了证明$merge(odd(\sigma), even(\sigma)) = \sigma$，只要证明$R$是互模拟关系就可以了，也就是要证明互模拟关系的两个必要条件。

对于第一个条件，有

$$head(merge(odd(\sigma), even(\sigma))) = head(odd(\sigma)) = head(\sigma)$$

对第二个条件，若有$\langle merge(odd(\sigma), even(\sigma)), \sigma\rangle \in R$，则把*tail*应用到该序对的两元时，产生的新序对组$\langle tail(merge(odd(\sigma), even(\sigma))), tail(\sigma)\rangle$也在$R$中，因为$tail(merge(odd(\sigma), even(\sigma))) = merge(odd(tail(\sigma)), even(tail(\sigma)))$。利用$even = odd \circ tail$，该等式的证明如下：

$$\begin{aligned} tail(merge(odd(\sigma), even(\sigma))) &= merge(even(\sigma), tail(odd(\sigma)) \\ &= merge(odd(tail(\sigma)), odd(tail(tail(\sigma)))) \\ &= merge(odd(tail(\sigma)), even(tail(\sigma))) \end{aligned}$$

利用归纳和等式演算，也可以证明$merge(odd(\sigma), even(\sigma)) = \sigma$，但是没有这么简单。可以对所有的自然数$n$，用归纳法先证明下面几个等式：

$$head(tail^{(n)}(odd(\sigma))) = head(tail^{(2n)}(\sigma))$$

$$head(tail^{(2n)}(merge(\sigma, \tau))) = head(tail^{(n)}(\sigma))$$

$$head(tail^{(2n+1)}(merge(\sigma, \tau))) = head(tail^{(n)}(\tau))$$

然后利用等式演算证明

$$head(tail^{(2n)}(merge(odd(\sigma), even(\sigma)))) = head(tail^{(n)}(\sigma))$$

显然这个证明比用余归纳原理的证明要复杂得多。

6.2.2 归纳和余归纳指南

本节从集合论角度介绍余归纳定义和余归纳证明原理。

令U是某个泛集（*universal set*），并且$F : P(U) \to P(U)$（$P(U)$表示U的幂集）是一个单调函数（即$X \subseteq Y$蕴涵$F(X) \subseteq F(Y)$）。归纳和余归纳是对偶的证明原理，它们分别衍生于作为形式为$X = F(X)$方程的最小解或最大解的集合的定义。

首先给出一些定义。一个集合$X \subseteq U$是F**封闭**的，当且仅当$F(X) \subseteq X$。对偶地，一个集合$X \subseteq U$是F**致密**（*dense*）的，当且仅当$X \subseteq F(X)$。F的不动点是方程$X = F(X)$的解。令$\mu X.F(X)$和$\nu X.F(X)$分别是满足下面条件的U的子集：

$$\mu X.F(X) \triangleq \bigcap \{X \mid F(X) \subseteq X\}$$

$$\nu X.F(X) \triangleq \bigcup \{X \mid X \subseteq F(X)\}$$

引理6.1

（1）$\mu X.F(X)$是最小的F封闭集合。

（2）$\nu X.F(X)$是最大的F致密集合。

证明 仅证明（2），（1）可以由对偶地论述得到。由$\nu X.F(X)$的定义知道，它包含每一个F致密集合，因此仅需要证明它本身是F致密的就可以了。这只要证明下面的引理就足够了：

如果每个X_i都是F致密的，那么它们的并$\bigcup_i X_i$也是。

因为对每个i有$X_i \subseteq F(X_i)$，那么$\bigcup_i X_i \subseteq \bigcup_i F(X_i)$。因为$F$是单调的，那么对每个$i$有$F(X_i) \subseteq F(\bigcup_i X_i)$。由此可得$\bigcup_i F(X_i) \subseteq F(\bigcup_i X_i)$。再由传递性，$\bigcup_i X_i \subseteq F(\bigcup_i X_i)$，即$\bigcup_i X_i$是$F$致密的。 □

定理6.2

（1）$\mu X.F(X)$是F的最小不动点。

（2）$\nu X.F(X)$是F的最大不动点。

证明 再次仅证明（2），（1）可以由对偶地论述得到。令$\nu = \nu X.F(X)$。由引理6.1，ν是致密的，因此$\nu \subseteq F(\nu)$。由F的单调性，$F(\nu) \subseteq F(F(\nu))$，因此$F(\nu)$也是致密的。由$\nu$的定义知道$F(\nu) \subseteq \nu$。由$\nu$和$F(\nu)$之间的这两个不等式得$\nu = F(\nu)$。$\nu$是最大不动点，因为任何其他不动点都是致密的，因而都包含在ν中。 □

$\mu X.F(X)$是$X = F(X)$的最小解，被称为是由F归纳地定义的集合；对偶地，$\nu X.F(X)$是$X = F(X)$的最大解，被称为是由F余归纳地定义的集合。这样就得到和这些定义相关联的两个对偶的证明原理：

（1）归纳： 如果X是F封闭的，那么$\mu X.F(X) \subseteq X$。

（2）余归纳：如果X是F致密的，那么$X \subseteq \nu X.F(X)$。

自然数归纳是该一般框架的一种特殊情况。假定存在一个元素$0 \in U$，并且存在一个内射函数$S : U \to U$。若单调函数$F : P(U) \to P(U)$由下式定义：

$$F(X) \triangleq \{0\} \cup \{S(x) \mid x \in X\}$$

令集合$\mathcal{N} \triangleq \mu X.F(X)$，那么上述归纳原理告知：

若$F(X) \subseteq X$，那么$\mathcal{N} \subseteq X$。

换句话说，

若$0 \in X$，并且只要$x \in X$，则$S(x) \in X$，那么$\mathcal{N} \subseteq X$。

这样，要想证明所有的自然数满足某个性质，可以假定X是满足该性质的所有元素集合，

然后证明X满足不等式

$$\{0\} \cup \{S(x) \mid x \in X\} \subseteq X$$

若能证明，则$\mathcal{N} \subseteq X$，即所有自然数都满足该性质。这就是自然数归纳。

计算机学科中熟悉的结构归纳和规则归纳等证明原理都是从某个特定的归纳定义得到的归纳原理。

对偶地，一个集合是余归纳定义的，若它是某种形式的不等式的最大解。下面以不终止的项重写系统为例来解释。

例6.5 用T表示某个项重写系统的项集。按某种确定的归约策略，不终止的项集可以如下定义：

$$T_{non} \triangleq \{\, M \mid \forall N{:}T.((M \twoheadrightarrow N) \Rightarrow N \text{ 不是范式}) \,\}$$

可以根据无界的归约来余归纳地刻画不终止性。令 $D : P(T) \to P(T)$并且 T_{gfp} 是如下定义的 T 的子集：

$$D(X) \triangleq \{\, M \mid \exists N{:}T.((M \to N) \wedge N \in X \,\}$$

$$T_{gfp} \triangleq \nu X.D(X)$$

很容易看出D是单调的，因此由余归纳定义知道，T_{gfp}是最大的D致密集合并且$T_{gfp} = D(T_{gfp})$。

下面证明 $T_{non} = T_{gfp}$。首先证明 $T_{gfp} \subseteq T_{non}$。假定 $M \in T_{gfp}$，则必须证明每当 $M \twoheadrightarrow N$，则 N 不是范式，即证明 $M \in T_{non}$。因为 $M \in T_{gfp}$，则存在 M'使得 $M \to M'$并且 $M' \in T_{gfp}$。显然由归纳可知，若 $M \twoheadrightarrow N$，则 N 不是范式。

再证明 $T_{non} \subseteq T_{gfp}$。由余归纳的定义，只要证明 T_{non} 是 D 致密的就可以了，因为 T_{gfp} 是最大的 D 致密集合。假定 $M \in T_{non}$。因为 $M \twoheadrightarrow M$，因此 M 不是范式，即存在 N 使得 $M \to N$。因为只要 $N \twoheadrightarrow N'$，则也有 $M \twoheadrightarrow N'$，并且 N'也不是范式，因为 $M \in T_{non}$。于是有 $N \in T_{non}$，即 $M \in D(T_{non})$。从 $M \in T_{non}$ 得到 $M \in D(T_{non})$就可以知道 T_{non} 是 D 致密的。 □

6.2.3 代数和余代数

从普通算法到交互计算的转变在数学上表现为一系列的扩展：从归纳到余归纳的扩展、从良基集到非良基集的扩展、从代数到余代数的扩展。从归纳到余归纳的扩展表达了从字符串到流的转变，这是从算法到交互的转变的基础；非良基集作为流的行为的形式模型被引进；余代数则为流的演算提供工具，它在交互计算模型中的地位相当于λ演算在图灵计算模型中的地位。本小节简要介绍余代数以及它和代数之间的对偶性，所用到的范畴论初步知识尽量用比较通俗的文字来阐述。

从第 3 章已经知道，对两个集合 X 和 Y，它们的笛卡儿积是如下的序对集合：

$$X \times Y \triangleq \{\langle x, y\rangle \mid x \in X \wedge y \in Y \,\}$$

并且射影函数 $\mathbf{Proj}_1 : X \times Y \to X$ 和 $\mathbf{Proj}_2 : X \times Y \to Y$ 满足等式 $\mathbf{Proj}_1\,(x, y) = x$ 和 $\mathbf{Proj}_2(x, y) = y$。还有，对函数 $f : Z \to X$ 和 $g : Z \to Y$，存在唯一的“配对”函数$\langle f, g\rangle : Z \to X \times Y$ 使得 $\mathbf{Proj}_1 \circ \langle f, g\rangle = f$ 并且 $\mathbf{Proj}_2 \circ \langle f, g\rangle = g$，即对 $z \in Z$，$\langle f, g\rangle(z) = \langle f(z), g(z)\rangle \in X \times Y$。

二元积的这些性质在范畴论中可以用交换图表表示，如图 6.1(a)所示。交换图表中每个节点都表示一个集合，有向边表示相应的两个集合之间的一个函数。若从交换图表一个节点到另一个节点有两条路径，则这两条路径上的函数分别复合的结果相等。图 6.1(a)表示的就是 $\mathbf{Proj}_1 \circ \langle f, g\rangle = f$ 并且 $\mathbf{Proj}_2 \circ \langle f, g\rangle = g$。

从第 3 章已经知道，对两个集合 X 和 Y，它们的和（又称可区分并、余积）是如下的序对集合：

$$X + Y \triangleq \{\langle 0, x\rangle \mid x \in X \,\} \cup \{\langle 1, y\rangle \mid y \in Y \,\}$$

其中序对中第一个成员 0 和 1 用来使并可区分。内射函数（又称余射影函数）$\mathbf{Inleft} : X \to X + Y$ 和 $\mathbf{Inright} : Y \to X + Y$ 以及它们满足的性质可以用 6.1(b)的交换图表表示。即具有性质[*f*,

$g]\circ$**Inleft** $= f$ 并且$[f, g]\circ$**Inright** $= g$，其中余配对函数$[f, g] : X + Y \to Z$ 的定义如下：

$$[f, g](w) \triangleq \begin{cases} f(x) & \text{如果} w = \langle 0, x\rangle \\ g(y) & \text{如果} w = \langle 1, y\rangle \end{cases}$$

从图 6.1 可以看出，(a)和(b)两个图的区别在于代表函数的所有有向边的方向相反，这正体现积与和的对偶性。上面有关二元积和二元和的性质很容易推广到多元的场合。

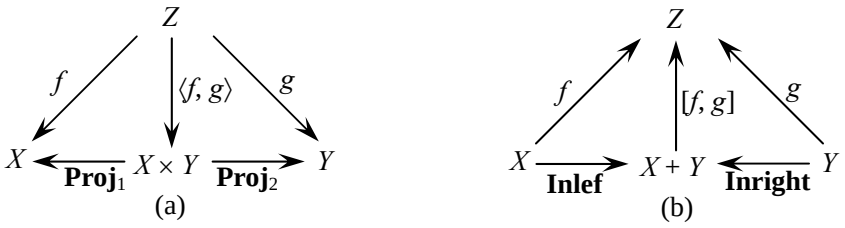

图 6.1 积与和的交换图表

所有的集合和它们之间的函数构成一个**范畴**，称为集合范畴。在这个范畴中，每个集合是范畴的一个**对象**（表示为图表上的一个节点），每个函数是相应两个对象之间的**射**（表示为图表上的一条有向边）。所有这些对象（集合）加上这些对象之间的所有射（函数）满足作为一个范畴的 4 个条件：

（1）射是可以合成的（函数可以复合）；

（2）射的合成满足结合律（函数复合有性质$(h \circ g) \circ f = h \circ (g \circ f)$）；

（3）每个对象都有一个恒等射（每个集合都有一个恒等函数）；

（4）如果$f : A \to B$ 是对象 A 到 B 的射，id_A 和 id_B 分别表示 A 和 B 的恒等射，$\circ$表示射的合成运算，那么$f \circ id_A = id_B \circ f = f$（显然恒等函数满足该性质）。

函子是范畴之间保结构的映射，在此只关心集合范畴到它自身的映射。集合范畴到它自身的映射 F 由 F_0 和 F_1 两部分构成：F_0 将集合映射到集合，F_1 将函数映射到函数。映射 F 若满足下面 3 个条件则称为函子：

（1）若$f : A \to B$ 在集合范畴中，则 $F_1(f) : F_0(A) \to F_0(B)$也在集合范畴中；

（2）对任何集合 A，$F_1(id_A) = id_{F_0(A)}$；

（3）若$f : A \to B$ 和 $g : B \to C$ 都在集合范畴中，则 $F_1(g \circ f) = F_1(g) \circ F_1(f)$。

下面不再使用 F_0 和 F_1，而是直接使用 F，若它的变元是集合或函数，则 F 分别代表 F_0 或 F_1。

下面可以基于函子来定义代数，只考虑单类代数的情况。令 F 是函子，F 的一个**代数**（简称 F 代数）是一个序对$\langle U, a\rangle$，其中 U 是集合，称为该代数的载体，a 是函数 $a : F(U) \to U$，称为该代数的**代数结构**（也称为运算）。

例 6.6 自然数上的零和后继函数 $0 : 1 \to \mathcal{N}$ 和 $S : \mathcal{N} \to \mathcal{N}$（其中 1 在这里代表对应 *unit* 类型的单元素集合）形成函子 $F(X) = 1 + X$ 的一个 F 代数$\langle 1+\mathcal{N}, [0, S] : 1+\mathcal{N} \to \mathcal{N}\rangle$。

以集合 A 的元素标记节点的二叉树的集合用 *tree*(A)表示，则空树 *nil* 可用函数 *nil* : $1 \to$ *tree*(A)表示，*node* : *tree*(A) × A × *tree*(A) → *tree*(A)表示从两棵子树和一个节点标记构造一棵树。*nil* 和 *node* 形成函子 $F(X) = 1 + (X \times A \times X)$的一个代数 $[nil, node] : 1 + (tree(A) \times A \times tree(A)) \to tree(A)$。 □

基于函子和交换图表可以给出代数**同态**的另一种表示。令 F 是函子，$a : F(U) \to U$ 和 $b : F(V) \to V$ 是两个函数。F 代数$\langle U, a\rangle$到$\langle V, b\rangle$的同态是一个函数 $f : U \to V$，满足 $f \circ a = b \circ F(f)$，其交换图表在图 6.2。

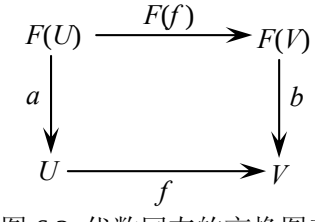

图 6.2 代数同态的交换图表

可以进一步定义**初始代数**。F 代数$\langle U, a\rangle$是初始的，如果对任意的 F 代数$\langle V, b\rangle$，都存在从$\langle U, a\rangle$到$\langle V, b\rangle$的唯一同态 f。

现在基于函子来定义余代数。令 F 是函子，F 的一个**余代数**（简称 F 余代数）是一个序对$\langle U, c\rangle$，其中 U 是集合，称为该余代数的载体，c 是函数 $c : U \to F(U)$，称为该余代数的**余代数结构**（也称为运算）。由于余代数经常描述动态系统，载体也叫做**状态空间**。

$F(U) \to U$ 的代数和 $U \to F(U)$的余代数有什么区别？本质上这是构造和观察之间的区别。一个代数由载体集合 U 和射入 U 的一个函数 $a : F(U) \to U$ 组成，它告知怎样构造 U 的元素。而一个余代数由载体集合 U 和一个逆向的函数 $c : U \to F(U)$组成。此时不知道怎样形成 U 的元素，仅有作用在 U 上的操作，它给出关于 U 的某些信息。

基于函子和交换图表也可以给出余代数同态的一种表示。令 F 是函子，$a : U \to F(U)$和 $b : V \to F(V)$是两个函数。F 余代数$\langle V, b\rangle$到$\langle U, a\rangle$的同态是一个函数$f : V \to U$，满足 $a \circ f = F(f) \circ b$，其交换图表在图 6.3。

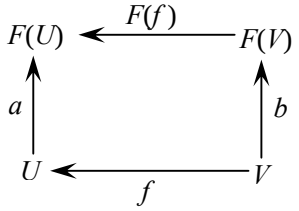

图 6.3 余代数同态的交换图表

也可以进一步定义终结代数。F 代数$\langle U, a\rangle$是终结的，如果对任意的 F 代数$\langle V, b\rangle$，都存在从$\langle V, b\rangle$到$\langle U, a\rangle$的唯一同态 f。

例 6.7 考虑有两个按键 *value* 和 *next* 的机器。按 *value* 键时，它不影响机器内部状态，并且产生数据集 A 的某个元素，该元素代表机器内部状态的某个可见属性。因此，连续按 *value* 键两次则产生同样的结果。按 *next* 键，则机器转移到另一个状态，该状态的性质也可以通过按 *value* 键来观察。抽象地说，该机器可以用状态空间 U 上的余代数

$$\langle value, next\rangle : U \to A \times U$$

来描述，该余代数的函数$\langle value, next\rangle$由两个函数 $value : U \to A$ 和 $next : U \to U$ 组成。在状态 $u \in U$ 下，连续地交替按 *next* 键和 *value* 键，可以产生无限序列$(a_1, a_2, \ldots)$，它可以看成 $\mathcal{N} \to A$ 上的一个函数，其中 $a_i = value(next^{(i)}(u)) \in A$。该序列就是状态 u 引发的可观察结果。若 $u_1, u_2 \in U$ 给出同样的可观察序列，则称 u_1 和 u_2 从可观察角度是不可区分的。 □

6.3 递归类型

6.3.1 递归类型总揽

在类型表达式的语法中增加类型变量 $r, s, t, \ldots$，再增加一种形式为$\mu t.\sigma$的类型表达式，由此可以把递归定义的类型加到 PCF，或加到其它基于λ演算的语言中。在类型定义 $t = \sigma$中，若 t 出现在σ中，则该类型等式相当于 t 的一个递归定义式。可以像为递归定义的项引入不动点算子那样，也为递归定义的类型引入不动点算子，$fix(\lambda t.\sigma)$ 用来表示等式 $t = \sigma$的一个解。递归类型是指类型等式系统的解，就像递归函数是项等式系统的解一样。第 7 章介绍多态类型时，类型表达式的抽象用记号Π而不是λ，以避免同项的抽象发生混淆，本章由于两种抽象表达式不会同时出现一个式子中，因而暂时用同一个记号。第 7 章还会讨论$\lambda t.\sigma$中 t 所属集合，目前暂时忽略它。习惯上用$\mu t.\sigma$作为 $fix(\lambda t.\sigma)$的语法美化，并且把μ看成基本符号。在$\mu t.\sigma$中μ 约束 t；类型表达式和项一样，也是既可以有约束变量又可以有自由变量。若在 PCF 中把单位类型、和类型及递归类型加入，PCF 的类型表达式则是由文法

$$\sigma ::= b \mid t \mid unit \mid \sigma + \sigma \mid \sigma \times \sigma \mid \sigma \to \sigma \mid \mu t.\sigma$$

产生的闭表达式，其中 b 代表基本类型，t 代表类型变量。一个项，若其类型表达式含自由变量，则会导致多态性，多态性要等到第 7 章讨论，因此这里将类型表达式限制到闭表达式。在此看到，合法的类型表达式已经难以用纯文法来表示了，这个问题也等到第 7 章解决。仅

约束变量的名字不同的类型表达式在此也被看成是等价的。

有递归类型时，需要仔细考虑类型相等问题。虽然从直观上讲，$\mu t.\sigma$代表等式 $t = \sigma$的一个解，但是对这个类型等式有两种可能的解释。第一种解释是，等式左右两边是真正不可区分的类型。在这种观点下，类型相等变得相当复杂，因为 $t = \sigma$意味着 $t = [\sigma/t]\sigma$。使用μ语法，则有等式

$$\mu t.\sigma = [\mu t.\sigma/t]\sigma$$

并且许多其它等式都可以由此得出。虽然类型相等的观点有些吸引力，但它使得确定项良类型与否变得更加困难。具体说，必须考虑项的类型相等问题，以便把某类型的一个项用到要求它具有相等类型（语法上可以有区别）的场合。本章不用这种观点，而用下面的观点。

第二种观点把等式 $t = \sigma$看成要找到类型 t，它和σ**同构**。这里用

$$\mu t.\sigma \cong [\mu t.\sigma/t]\sigma$$

表达这个意思。其中≅表示同构。在同构观点下，类型$\mu t.\sigma$并不等于类型$[\mu t.\sigma/t]\sigma$，但是存在把$\mu t.\sigma$的项“转换”成$[\mu t.\sigma/t]\sigma$的项的函数，反之亦然。对于递归类型，用同构代替相等后，可以继续用语法上的等式来表示类型相等（除了用于约束类型变量的改名以外）。对于类型同构情况，所要付的代价是，必须给出项在类型$\mu t.\sigma$和$[\mu t.\sigma/t]\sigma$之间变化的转换函数，使得可以准确知道每个项的类型的语法形式。

递归类型理论中最中心的问题是对类型等式 $t \cong \sigma$有解和无解的情况进行分类，即$\lambda t.\sigma$在什么情况下有不动点。这个问题在 6.4 节介绍。因此在形式为$\mu t.\sigma$的递归类型中，本节没有讨论对σ的限制。

下面在 PCF 语言中通过增加定型规则，来为递归类型同时给出新增项的形式及其类型。由于涉及递归类型的项的形式允许把类型$\mu t.\sigma$的项转换成类型$[\mu t.\sigma/t]\sigma$的项并且反之亦然，因此有下面两个定型规则

$$\frac{M : [\mu t.\sigma/t]\,\sigma}{\text{fold}\ M : \mu t.\,\sigma} \qquad (\mu\ Intro)$$

$$\frac{M : \mu t.\sigma}{\text{unfold}\ M : [\mu t.\sigma/t]\,\sigma} \qquad (\mu\ Elim)$$

其中函数 **fold** 和 **unfold** 互逆，满足下面的等式公理

unfold(**fold** M) = M　　**fold**(**unfold** M) = M　　(*fold*/*unfold*)

μ *Intro* 规则根据$\mu t.\sigma$的展开类型的元素来引入$\mu t.\sigma$类型的元素，而消去形式 **unfold** M 从该递归类型的一个元素给出其展开类型的相应元素。项 **fold** M 和 **unfold** M 处于递归类型$\mu t.\sigma$和它的展开类型$[\mu t.\sigma/t]\sigma$之间，是联系它们的桥梁。

基于急切归约策略，PCF 语言所增加的递归类型的归约规则见表 6.1。采用惰性归约策略的归约规则也很容易设计。

表 6.1　PCF 的急切归约

值

如果 V 是值，则 **fold** V 也是值。

公理

unfold(**fold** V) $\xrightarrow{\text{eager}} V$　　V 是值

子项规则

函数　$\dfrac{M \xrightarrow{\text{eager}} M'}{\text{fold}\ M \xrightarrow{\text{eager}} \text{fold}\ M'}$　$\dfrac{M \xrightarrow{\text{eager}} M'}{\text{unfold}\ M \xrightarrow{\text{eager}} \text{unfold}\ M'}$

通常仅把 *fold*/*unfold* 的第一个公理用作从左向右的归约公理。虽然运算 **fold** 和 **unfold** 能消去项的类型歧义，但写起来是非常麻烦的。在后面的章节中，有时会省略 **fold** 和 **unfold**，以增强可读性。

对增加递归类型的 PCF 语言，很容易证明下面的安全性定理。

定理 6.1 （1）保持性　若$\Gamma \triangleright M : \sigma$并且 $M \xrightarrow{\text{eager}} M'$，则$\Gamma \triangleright M' : \sigma$。

（2）前进性　若$\varnothing \triangleright M : \sigma$ 且σ是可观测类型，则 M 是闭范式，或者存在程序 M'，使得 $M \xrightarrow{\text{eager}} M'$。

6.3.2 递归的数据结构

递归类型在程序设计语言中有很多应用，包括：

（1）表示像表和树这样的无界数据结构；

（2）表示像循环图这样的带环数据结构；

（3）支持动态定型和动态类型派遣（*type dispatch*）；

（4）支持协同例程和类似的控制结构。

本小节举例说明在递归数据结构方面的应用。

递归类型的一个重要应用是表示像表和树这样的数据结构，它们的大小和内容在程序的执行过程中确定。例如自然数表的类型可以想象成类型等式 $t \cong unit + (nat \times t)$的一个解。该解是函数$\Phi_{list}$的一个不动点 $fix(\Phi_{list})$，Φ_{list} 的定义如下：

$$\Phi_{list} \triangleq \lambda t.unit + (nat \times t)$$

其不动点σ满足同构$\sigma \cong \Phi_{list}(\sigma)$。这时的 **fold** 和 **unfold** 函数的类型是

$$\textbf{fold} : unit + (nat \times t) \to t$$

和

$$\textbf{unfold} : t \to unit + (nat \times t)$$

它们是该类型等式的解和它的定义条件之间的见证。（删除）

类似地，对于二叉树，Φ_{tree} 的定义如下：

$$\Phi_{tree} \triangleq \lambda t.unit + (t \times t)$$

需要寻找$\sigma \cong \Phi_{tree}(\sigma)$的一个解，即$\Phi_{tree}$ 的一个不动点 $fix(\Phi_{tree})$。

下面详细介绍自然数类型的例子。在 PCF 语言中自然数类型作为一种基本类型。引入单位类型、和类型以及递归类型定义后，可以用它们来定义 *nat* 类型、数码 0，1，2，…、后继函数、前驱函数和判零函数。于是可以把任何 PCF 表达式翻译到只含递归类型定义、单位类型、积类型与和类型的表达式（因为另一个基本类型 *bool* 可以用单位类型与和类型来定义，见 3.2.4 节）。

自然数的一个显著特征是，如果加一个元素到自然数集合，所得集合和自然数集合一一对应，即同构。因为和 $unit+\tau$ 比 τ 仅多一个元素，因此期望 *nat* 满足下面的同构：

$$nat \cong unit + nat$$

这就导致下面把 *nat* 作为递归类型的一个定义：

$$nat \triangleq \mu t.unit + t$$

直观上，可以通过下面的同构来理解这个定义：

$$\begin{aligned} nat &\cong unit + nat \\ &\cong unit + (unit + nat) \\ &\cong unit + (unit + (unit + nat)) \\ &\cong \cdots \\ &\cong unit + (unit + (unit + (unit + (unit + \cdots + nat)\cdots)))\end{aligned}$$

这个展开过程可以按照需要一直继续，然后把 *nat* 想象成这无数个单位类型的可区分的并。其中自然数 0 由这个和的第一个 *unit* 类型的元素∗表示，若写成项 **Inleft**∗，则它成为上面和类型的元素，再写成项 **fold(Inleft**∗**)**，则被转换成该递归类型的一个元素；自然数 1 由这个和的第二个 *unit* 类型的元素∗表示，项 **fold(Inright fold(Inleft**∗**))**表示它被转换成该递归类型的一个元素；其余自然数依此类推。

为了避免自然数和用于代表自然数的项之间的混淆，在这里用⌈*n*⌉表示代表自然数 *n* 的数码（项）。根据上面的同构式，就有

⌈0⌉ ≜ **fold(Inleft**∗**)**

对任何自然数 $n > 0$，可以类似地定义如下：

⌈*n*⌉ ≜ **fold(Inright fold(Inright** … **fold(Inleft**∗**)** … **))**

其中 **Inright** 出现 *n* 次，并且 **hold** 应用有 *n*+1 次。

后继函数正好使用 **fold** 和 **Inright**：

succ ≜ λ*x*:*nat*. **fold(Inright** *x*)

该表达式有正确的类型，因为如果 *x*:*nat*，那么 **Inright** *x* 属于 *unit*+*nat*，**fold(Inright** *x*)属于 *nat*。读者可以检验，对任何自然数有 *succ*⌈*n*⌉ =⌈*n* +1⌉。

判零测试的定义如下：

zero? ≜ λ*x*:*nat*.**Case**$^{unit,\ nat,\ bool}$(**unfold** *x*) (λ*y*:*unit*.*true*) (λ*z*:*nat*.*false*)

读者很容易检查，*zero*? ⌈0⌉ = *true* 并且对 $n > 0$，*zero*? ⌈*n*⌉ = *false*。

最后一个操作是前驱算子。注意，0 虽然没有前驱，但是为了便利起见，取 *pred* 0 = 0。这样，前驱函数的定义如下，它类似于判零测试。

pred ≜ λ*x*:*nat*.**Case**$^{unit,\ nat,\ bool}$(**unfold** *x*) (λ*y*:*unit*.⌈0⌉) (λ*z*:*nat*.*z*)

读者可以检验，对任意 *x* : *nat*，有 *pred* (*succ x*) = *x*。

作为另一个例子，自然数表的类型可以由递归类型$\mu t.unit + (nat \times t)$来表示，即有同构

list ≅ *unit* + (*nat* × *list*)

表的两种构造形式，空表 *nil* 和非空表 *cons x l*，由下面的定义给出：

nil ≜ **fold(Inleft** ∗**)**

cons x l ≜ **fold(Inright** ⟨*x*, *l*⟩)

根据表的形式进行分支的函数 *listcase* 定义如下：

listcase≜ λ*x*:*list*.λ*y*:σ.λ*f*:*list*→σ.**Case**$^{unit,\ list,\ \sigma}$(**unfold** *x*) (λ*w*:*unit*.*y*) (*f*)

其中第 1 个参数 *x* 是 *list* 类型的项，若 *x* 是空表，该表达式的结果是σ类型的项 *y*，即等于第 2 个参数；若 *x* 是非空表，则该表达式的结果由类型为 *list*→σ的第三个参数 *f* 应用到该非空表得到。

6.4 归纳类型和余归纳类型

6.4.1 归纳类型和余归纳类型总揽

归纳类型和余归纳类型是两类重要的递归类型。归纳类型对应到某个类型同构等式的最小解，也叫初始解；余归纳类型对应到它们的最大解，也叫终结解。直观地说，归纳类型被看成是包含它们引入形式的“最小”类型；而其消去形式是在这引入形式上的一种递归形式。对偶地，余归纳被看成是和它们的消去形式一致的“最大”类型；而其引入形式是一种在这消去形式需要时提供元素的方法。

推动归纳类型研究的一个例子是自然数类型。它是包含引入形式 0 和 *succ*(*M*)的最小类型，其中 *M* 也是引入形式。为了进行以一个自然数为参数的计算，可以定义一个递归函数，

它对于 0 返回一个特别的值，而对于 $succ(M)$则返回一个基于以 M 为参数递归调用本身的结果来定义的值。归纳类型的其他例子有串、表、树和任何其他可以看成从它引入形式有限地生成的类型。

推动余归纳类型研究的一个例子是自然数流类型。每个流可以想象成处于由一个自然数（它的头）和另一个流（它的尾）构成的序对的生成过程中。为了创建一个流，需要定义一个生成子（它被调用时产生一个这样的自然数）和一个对该生成子的余递归调用。余归纳类型的其他例子有正则树类型（含有其后代也是其祖先的节点）和惰性自然数类型（包含一个由无数个后继算子堆成的“无穷远点”）。

为了便于理解归纳类型和余归纳类型，将它们同 6.2.3 节的代数和余代数联系起来介绍。在递归的类型同构式 $t \cong \sigma$中，σ中含自由类型变量 t 的类型表达式，$\lambda t.\sigma$被称为**类型抽象子**。如果类型表达式都能解释到集合，从而把类型表达式都看成集合表达式，那么类型抽象子就可以看成 6.2 节的集合范畴的函子 F。首先$\lambda t.\sigma$将类型表达式（集合）映射到类型表达式（集合），例如若$\sigma \equiv unit + t$，则$(\lambda t.\sigma)\tau = unit + \tau$，$(\lambda t.\sigma)\rho = unit + \rho$。扩展类型抽象子$\lambda t.\sigma$，使得它能将函数映射到函数（下面开始将$\lambda t.\sigma$缩写成 F），并且适当地定义 F，则 F 就相当于集合范畴的函子。例如，继续刚才的例子，对任意的 $M : \tau \to \rho$, 将 $F(M)$: $F(\tau) \to F(\rho)$（即 $F(M)$: $(unit + \tau) \to (unit + \rho)$）定义为：

$$F(M) = \lambda z{:}\ unit + \tau.\mathbf{Case}^{unit,\ \tau,\ unit+\rho}\ z\ (\lambda x.unit{:}\mathbf{Inleft}^{unit,\ \rho}(*))\ (\lambda y.\tau{:}\ \mathbf{Inright}^{unit,\ \rho}(My))$$

则可以看出 F 满足作为函子的条件。本例的结论可以推广到一般情况，即不论σ是什么形式，F 都能扩展为函子。

递归的类型同构式 $t \cong \sigma$的解是一个类型τ，使得τ和$F(\tau)$之间存在一个同构，即τ是 F 的不动点。对于 f: $F(\tau) \to \tau$，$\langle \tau, f \rangle$是一个 F 代数，并且 F 代数之间满足图 6.4 这样的交换图表，其中 h 是代数同态。若$\langle \tau, f \rangle$是初始 F 代数，即对任意 F 代数$\langle \rho, g \rangle$，正好存在唯一的代数同态 $h : \tau \to \rho$，那么τ是 F 最小不动点。F 的最小不动点用μF 表示。

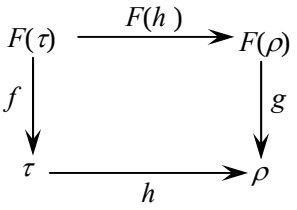

图 6.4 F 代数的交换图表

为什么$\langle \tau, f \rangle$是初始 F 代数，则τ就是最小不动点，需要回顾 2.4.2 节初始代数没有“垃圾”的特点，并对照 6.2.2 节有关最小不动点的讨论。

递归的类型同构式 $t \cong \sigma$的一个对偶的解则可以通过取终结 F 余代数得到，F 余代数是一个序对$\langle \tau, f \rangle$，其中 f: $\tau \to F(\tau)$。将图 6.4 的箭头全部逆向，就得到 F 余代数的交换图表。如果$\langle \tau, f \rangle$是终结代数，则τ是 F 的最大不动点。F 的最大不动点用νF 表示。

第 4 章在论域理论中讨论最小不动点时指出，若$\mathcal{D}$是有底元的 CPO，并且 f: $\mathcal{D} \to \mathcal{D}$ 连续，那么 f 有最小不动点。在这里也类似，需要对函子 F 的形式加以限制，才能保证它有最小不动点和最大不动点。由于 F 在这里是类型抽象子$\lambda t.\sigma$，因此也就是对σ的形式加以限制。在此介绍一种比较严格的限制，它对本章的例子来讲已经够用了。类型抽象子$\lambda t.\sigma$是**有把握的**（*positive*），如果 t 不会出现在σ的一个函数类型的定义域中。例如$\lambda t.t \to t$ 不是有把握的，而$\lambda t.nat \to t$ 和$\lambda t.s \to t$ 是有把握的。

若把μF 和νF 加入 PCF 语言，则 PCF 的类型表达式是由文法

$$\sigma ::= b \mid t \mid unit \mid \sigma + \sigma \mid \sigma \times \sigma \mid \sigma \to \sigma \mid \mu F \mid \nu F$$

产生的闭表达式，其中 b 代表基本类型，t 代表类型变量，F 被限定到有把握的类型抽象子。同样，合法的类型表达式用纯文法难以表示的问题等到第 7 章解决。

在 PCF 语言中为最小不动点和最大不动点类型（通常分别称它们为归纳类型和余归纳类型）新增项的形式及其类型，可以通过下面新增的定型规则来定义。这一次将 **fold** 和 **unfold** 定义成项常量。

$$\Gamma \rhd \mathbf{fold}^{\mu F}: F(\mu F) \to \mu F \qquad (\mu\, Intro)$$

$$\frac{\Gamma \rhd M : F(\tau) \to \tau}{\Gamma \rhd \mathrm{it}^{\mu F} M : \mu F \to \tau} \qquad (\mu\, Elim)$$

$$\frac{\Gamma \rhd M : \tau \to F(\tau)}{\Gamma \rhd \mathrm{gen}^{\nu F} M : \tau \to \nu F} \qquad (\nu\, Intro)$$

$$\Gamma \rhd \mathbf{unfold}^{\nu F}: \nu F \to F(\nu F) \qquad (\nu\, Elim)$$

$\langle \mu F, \mathbf{fold}^{\mu F} \rangle$是（同构到唯一的）初始 F 代数，$\mathbf{fold}^{\mu F}$ 是它的构造子；函数应用 $\mathbf{it}^{\mu F} M$ 得到从$\langle \mu F, \mathbf{fold}^{\mu F} \rangle$到$\langle \tau, M \rangle$的唯一 F 代数同态。对偶地，$\langle \nu F, \mathbf{unfold}^{\nu F} \rangle$是（同构到唯一的）终结 F 余代数，$\mathbf{unfold}^{\nu F}$ 是它的解构子；函数应用 $\mathbf{gen}^{\nu F} M$ 得到从$\langle \tau, M \rangle$到$\langle \nu F, \mathbf{unfold}^{\nu F} \rangle$的唯一 F 余代数同态。

新增的项满足下面的等式公理和推理规则。

$$\mathbf{it}^{\mu F} M(\mathbf{fold}^{\mu F} N) = M(F(\mathbf{it}^{\mu F} M)N) \qquad (\mu\beta)$$

$$\frac{P(\mathrm{fold}^{\mu F} N) = M(F(P)N)}{P = \mathrm{it}^{\mu F} M} \qquad (\mu\eta)$$

$$\mathbf{unfold}^{\nu F}(\mathbf{gen}^{\nu F} MN) = F(\mathbf{gen}^{\nu F} M)(MN) \qquad (\nu\beta)$$

$$\frac{\mathrm{unfold}^{\nu F}(PN) = F(P)(MN)}{P = \mathrm{gen}^{\nu F} M} \qquad (\nu\eta)$$

这些公理和推理规则很容易从 F 代数和余代数的交换图表，以及初始代数和终结代数的同态的唯一性得到。

将上面两条等式公理从左向右定向，就得到 PCF 语言为归纳类型和余归纳类型所增加的归约规则：

$$\mathbf{it}^{\mu F} M(\mathbf{fold}^{\mu F} N) \to M(F(\mathbf{it}^{\mu F} M)N) \qquad (\mu\beta)$$

$$\mathbf{unfold}^{\nu F}(\mathbf{gen}^{\nu F} MN) \to F(\mathbf{gen}^{\nu F} M)(MN) \qquad (\nu\beta)$$

规则$(\mu\beta)$在说，为了完成在归纳类型的一个值 $\mathbf{fold}^{\mu F} N$ 上的 $\mathbf{it}^{\mu F} M$ 计算，需要在类型构造子 F 的引导下归纳地将 $\mathbf{it}^{\mu F} M$ 应用到该值，然后在其结果上完成归纳步骤。**it** 是迭代子（*iterator*）的缩写，若按$(\mu\beta)$规则逐步进行重写，可以看出 $\mathbf{it}^{\mu F} M$ 的迭代计算。初始性表示迭代子是个一般工具，用来定义应用到归纳类型的值的函数。有关 F 的规则取决于相应的类型构造子的形式，先前已经给过 $F(M)$: $(unit + \tau) \to (unit + \rho)$的定义，可以把它从左向右定向为一个重写规则。

规则$(\nu\beta)$是规则$(\mu\beta)$在余归纳类型上的对偶。终结性表示生成子 **gen** 也是个一般工具，它用来创建余归纳类型的值。

对增加了归纳类型和余归纳类型的 PCF 语言，很容易证明下面的安全性定理。

定理 6.2 （1）保持性　若$\Gamma \rhd M : \sigma$并且 $M \to M'$，则$\Gamma \rhd M' : \sigma$。

（2）前进性　若$\varnothing \rhd M : \sigma$ 且σ是可观测类型，则 M 是闭范式，或者存在程序 M'，使得 $M \to M'$。

6.4.2 帮助理解的实例

和介绍递归类型相比，介绍归纳类型和余归纳类型时，主要多关注了两点。其一是考虑递归定义的类型等式在什么情况下有解；其二是关注两个特别的解，由此导致项的语法及定型规则、等式公理和操作语义等都不一样。

若不区分这两个特别解，而把它们都看成是某个解的话，则归纳类型和余归纳类型的定型规则(μ *Intro*)和(ν *Elim*)就相当于 6.3.1 节给出的递归类型的定型规则(μ *Intro*)和(μ *Elim*)。

在表 6.1 递归类型的归约规则中，只有化简 **fold** 和 **unfold** 参数的规则，而不像 6.4.1 节有($\mu\beta$)和($\nu\beta$)这样将 **fold** 和 **unfold** 同它们的参数一起操作的规则，这是因为 6.3.1 节在一般性地介绍递归类型时没有考虑选择哪个不动点，因而不便设计规则。正因为这个原因，在介绍递归类型时，主要采用的例子是在解释 *nat* 不必作为基本类型，它可以看成是基于 *unit* 类型与和类型的一种递归类型。

本节先介绍一个余归纳类型的实例。

例 6.8 先前已多次提到递归的类型定义 $list \cong unit + (nat \times list)$，自然数表和自然数流的类型分别是相应的归纳类型$\mu F$ 和余归纳类型νF，其中函子 F 就是相应的类型构造子$\lambda list.unit + (nat \times list)$，并且

$$F(M) = \lambda z{:}unit+(nat\times list).$$
$$\textbf{Case}\, z\ (\lambda x.unit{:}\textbf{Inleft}(*))\ (\lambda\langle y, w\rangle.nat\times list{:}\textbf{Inright}\langle y, Mw\rangle)$$

下面将把它当成一条从左向右的归约规则来使用。

Fibonacci 数列 1, 1, 2, 3, 5, …是一个自然数流，是类型νF 的一个项，其定义 *fibs* 如下：

$M \triangleq \lambda\langle m, n\rangle : list.\textbf{Inright}\langle m, \langle n, m + n\rangle\rangle$

$fibs \triangleq \textbf{gen}\ M\ \langle 1, 1\rangle$

其中形式为$\langle a, b\rangle$项是项$\langle a, \textbf{Inright}\langle b, \textbf{Inleft}*\rangle\rangle$的语法美化。很容易检查，$M$ 的类型是 $list \to F(list)$，*fibs* 的类型是 $list \to \nu F$。将该项展开一次就可以看出该项确实表示 Fibonacci 数列。

$$\begin{aligned}
&\textbf{unfold}\ (fibs)\\
\equiv\ &\textbf{unfold}(\textbf{gen}\ M\ \langle 1, 1\rangle)\\
\to\ &F(\textbf{gen}\ M)(M\ \langle 1, 1\rangle)\\
\equiv\ &F(\textbf{gen}\ M)((\lambda\langle m, n\rangle.\textbf{Inright}\langle m, \langle n, m + n\rangle\rangle)\langle 1, 1\rangle)\\
\to\ &F(\textbf{gen}\ M)(\textbf{Inright}\langle 1, \langle 1, 2\rangle\rangle)\\
\to\ &(\lambda z{:}unit+(nat\times list).\\
&\qquad \textbf{Case}\, z\ (\lambda x.unit{:}\textbf{Inleft}(*))(\lambda\langle y, w\rangle.nat\times list{:}\textbf{Inright}\langle y, \textbf{gen}\ M\ w\rangle)\\
&\qquad\qquad (\textbf{Inright}\langle 1, \langle 1, 2\rangle\rangle)\\
\to\ &(\lambda\langle y, w\rangle.nat\times list{:}\textbf{Inright}\langle y, \textbf{gen}\ M\ w\rangle)\langle 1, \langle 1, 2\rangle\rangle\\
\to\ &\textbf{Inright}\langle 1, \textbf{gen}\ M\ \langle 1, 2\rangle\rangle
\end{aligned}$$

□

下面的例子了解一点归纳类型和余归纳类型之间的联系。

例 6.9 再次考虑自然数类型的递归定义 $nat \cong unit + nat$。直观上，相应的归纳类型μF 是由下面两点决定的最小（限制最紧）类型：

（1）代表 0 的 **fold(Inleft**$*$**)**是μF 类型的项，并且

（2）如果 N 是μF 类型的项，则它的后继 **fold(Inright** N**)**也是μF 类型的项。

对偶地，余归纳类型νF 是项 N 的最大（最宽容）类型，满足 **unfold**(N)等价于：

（1）由 **Inleft**$*$定义的 0，或者

（2）由 **Inright**(N')定义的某个项 $N'{:}\nu F$ 的后继。

因此，不难将归纳定义的自然数嵌入余归纳定义的自然数集中，但是逆方向是不可能的。粗略地解释是，下面定义的项ω

$M \triangleq \lambda n : nat.\textbf{Inright}\ n$

$\omega \triangleq \textbf{gen}\ M\ N$　　（N: nat）

是一个余归纳定义的自然数，它大于任何嵌入的归纳定义的自然数。因为从下面的一次展开可以看出ω是有无数个后继（**Inright**）的项，因而大于任何只带有限后继的项，也就是说它大于任何有限的自然数。

unfold (ω)

$\equiv$ **unfold**(**gen** M N)

$\to$ F(**gen** M)(M N)

$\equiv$ F(**gen** M)((λn : nat.**Inright** n) N)

$\to$ F(**gen** M)(**Inright** N)

$\to$ (λz:$unit$+nat.**Case** z (λx.$unit$:**Inleft**(∗))(λy.nat:**Inright**(**gen** M y))) (**Inright** N)

$\to$ (λy.nat:**Inright**(**gen** M y)) N

$\to$ **Inright**(**gen** M N)

任何将余归纳定义的自然数嵌入归纳定义的自然数集都不得不把ω放在有限自然数之间，使它大于某些有限自然数，又小于另一些有限自然数，显然同上述的解释有矛盾。 □

习　　题

6.1 按例 6.4 所说的方法，利用归纳和等式演算，证明 $merge(odd(\sigma), even(\sigma)) = \sigma$。

6.2 证明在任何有函数、笛卡儿积和 *unit* 类型的λ演算中，对每个类型σ，存在一个同构$\sigma \cong \sigma \times unit$。具体说，给出闭项

$M : \sigma \to \sigma \times unit$

$N : \sigma \times unit \to \sigma$

并证明两个合成 $M \circ N$ 和 $N \circ M$ 都等于相应类型上的恒等函数。

6.3 证明在任何有函数、和与 *null* 类型的λ演算中，对每个类型σ，存在一个同构$\sigma \cong \sigma + null$。具体说，给出闭项

$M : \sigma \to \sigma + null$

$N : \sigma + null \to \sigma$

并证明两个合成 $M \circ N$ 和 $N \circ M$ 都等于相应类型上的恒等函数。本习题比上一题要复杂一些，你可能要用习题 3.8 的结果。

6.4 使用 6.3.2 节给出的定义，证明在自然数的项上有如下的归约。

(a) 对任何自然数 n，有 $succ\lceil n \rceil \twoheadrightarrow \lceil n+1 \rceil$。

(b) $zero?\lceil 0 \rceil \twoheadrightarrow true$ 并且对任何 $n > 0$，$zero?\lceil n \rceil \twoheadrightarrow false$。

(c) 如果 x 是一个自然数变量，那么 $pred(succ\ x) \twoheadrightarrow x$。

6.5 基于 6.3.2 节自然数表的递归类型：

$list \triangleq \mu t.unit + (nat \times t)$

和构造符 $nil : list$ 和 $cons : nat \times list \to list$，来定义函数 *car* 和 *cdr*。函数 *car* 返回表的第一个元素，函数 *cdr* 返回第一个元素以后的所有元素组成的表。这些函数的类型如下：

$car : list \to unit + nat$

$cdr : list \to unit + list$

并且约定当应用于空表时，这两个函数都返回 **Inleft**∗。这样，可以定义这两个函数的一个版本，使得它们应用于空表时都是有意义的。写出定义这两个函数的项，使得对 $x : nat$ 和 $l : list$，下列的等式是可证明的。

$car\ nil =$ **Inleft** ∗

$car\ (cons\ x\ l) = x$

$cdr\ nil =$ **Inleft** ∗

$cdr\ (cons\ x\ l) = l$

6.6 对于递归的类型定义 $list \cong unit + (nat \times list)$，能否解释为什么自然数表和自然数流

的类型分别是相应的归纳类型μF和余归纳类型νF，其中 F 是对应到类型构造子$\lambda list.unit + (nat \times list)$的函子。

6.7 对于二叉树的递归类型定义 $t \cong unit + (t \times t)$，考虑相应的余归纳类型。请参照例 6.8 和 6.9 写一个ω项，并参照该例，通过一次展开来保证你所给出的确实是一棵无穷树。

6.8 程序设计语言 ML 有一种形式的递归类型声明，叫做 datatype，它组合和类型与类型递归。数据类型声明的一般形式是

datatype $t = l_1$ of $\sigma_1 \mid \ldots \mid l_k$ of σ_k

其中语法标记 $l_1, \ldots, l_k$ 的使用方式同变体类型（带标记和，见习题 3.30）中的一样。直观上，该声明定义一个类型 t，它是$\sigma_1, \ldots, \sigma_k$的和，标记 $l_1, \ldots, l_k$ 如同入射函数。如果被声明的类型 t 出现在$\sigma_1, \ldots, \sigma_k$中，那么这就是类型 t 的一个递归声明。注意，竖线“|”是 ML 语法的一部分，不要把它看成描述 ML 的元语言。

使用 datatype，以自然数作为叶节点的二叉树类型可以声明为

datatype $tree = leaf$ of $nat \mid node$ of $tree \times tree$

使用习题 3.30 给出的变体记号，它声明了满足

$$tree \cong [leaf : nat, node : tree \times tree]$$

的类型 $tree$。

（a）请解释，怎样把 datatype 声明看成一个使用习题 3.30 变体的形式为$\mu t.[\ldots]$的递归类型表达式。用你的一般方法来解释上述树类型。

（b）ML 的一个方便之处是可以用模板匹配来定义应用到被声明数据类型的函数。应用到树的函数可以用两个“子句”来声明，它们分别对应到树的两种不同形式。用本书的 ML 语法方言，按模板匹配方式定义在 $tree$ 上的函数具有

$$\begin{array}{lll} \textbf{letrec fun} & f(leaf(x : nat)) & = M \\ \mid & f(node(t_1 : tree, t_2 : tree)) & = N \\ & \textbf{in } P & \end{array}$$

的形式。其中变量 $x : nat$ 和 $f : tree \to \sigma$在 M 中是受约束的，$t_1, t_2 : tree$ 和 f 在 N 中是受约束的，并且 f 在这个声明体 P 中是受约束的。编译器进行检查以保证对该数据类型的每个构造子有一个子句。例如，对所有叶子值求和的函数声明如下：

$$\begin{array}{lll} \textbf{letrec fun} & f(leaf(x : nat)) & = x \\ \mid & f(node(t_1 : tree, t_2 : tree)) & = f(t_1) + f(t_2) \end{array}$$

请解释怎样把带模板匹配的函数定义看成一个带 **Case** 的函数并且把你的一般方法用到上面对所有叶子求和的函数。