第 I 部分　语言模型

第 1 章

文本求值

在本书中，我们将学习如何以人们易于理解，同时也适合形式分析的方式定义编程语言——在此，形式分析也应使用人们易于理解的方式。

其中一种定义语言的方式是使用文章段落去解释语言中允许的表达式类型以及它们的求值方式。这种方法的优点是读者可以快速掌握语言的核心部分，但是语言的细节却显然难以通过文章段落理解。更糟糕的是，无法用文章段落的方式进行形式分析。

另一种定义语言的方式是使用一些元语言来编写这个语言的解释器。如果读者熟悉这种元语言，这种技术显然可以清楚且完整地表示语言细节。如果这种元语言存在形式定义，那么这个解释器同样存在该语言的形式意义，也可以做形式分析。

用于定义另一种语言的元语言不必可以高效执行，因为它的首要目的是向人们解释其他语言。元语言的基本数据结构也不必从字和字节的角度定义。实际上，对元语言我们可以直接使用逻辑和基于程序文本全集的集合论。同样，物理机器上的计算也是可以被描述的，最终，从在内存空间中重排列位的角度看，抽象计算也可以从文本关系的角度进行描述。

1.1 定义集合

当我们写出如下 BNF 文法时

$$
\begin{array}{rcl}
B & = & t \\
 & | & f \\
 & | & (B \bullet B)
\end{array}
$$

实际上则定义了文本串集合 B。上述的定义可以拓展为 B 的约束：

$$t \in B$$
$$f \in B$$
$$a \in B \text{ 且 } b \in B \Rightarrow (a \bullet b) \in B$$

严格的说，B 是遵循上述约束的最小集合。

注意，我们有时使用“B”来代替“集合 B”，因为通过上下文很容易弄懂其真实的含义。有时在集合名称上使用下标或者撇号表示集合中的任一元素，例如“B_1”或者“B'”。因此，上面的条件可以写成

$$\begin{array}{ll} t \in B & [a] \\ f \in B & [b] \\ (B_1 \bullet B_2) \in B & [c] \end{array}$$

无论是用 BNF 简略表达式定义，还是扩展成约束集进行复杂定义，集合 B 都是被递归定义的。在有限空间内枚举 B 的所有元素是不可能的：

$$B = \{t, f, (t \bullet t), (t \bullet f), \ldots\}$$

但是，对 B 中一段特定文本，通过展示定义 B 的约束是如果规定属于 B 的文本，我们可以证明该文本是属于 B 的。例如，$(t \bullet (f \bullet t))$ 是属于 B 的：

1.	$t \in B$	根据 $[a]$
2.	$f \in B$	根据 $[b]$
3.	$t \in B$	根据 $[a]$
4.	$(f \bullet t) \in B$	根据 $2, 3$, 且$[c]$
5.	$(t \bullet (f \bullet t)) \in B$	根据 $1, 4$, 且$[c]$

我们可以写成证明树形式：

$$\dfrac{t \in B\ [a] \qquad \dfrac{f \in B\ [b] \qquad t \in B\ [a]}{(f \bullet t) \in B \qquad [c]}}{(t \bullet (f \bullet t)) \in B \qquad [c]}$$

我们也可以省略证明树上被应用规则的名称，因为这通常是显而易见的：

$$\dfrac{t \in B \qquad \dfrac{f \in B \qquad t \in B}{(f \bullet t) \in B}}{(t \bullet (f \bullet t)) \in B}$$

▷ **练习 1.1.** 下面哪个是属于 B 的？为 B 中的每个元素都写出一个以证明该元素一定属于 B 的证明树。

1. t
2. •
3. $((f \bullet t) \bullet (f \bullet f))$
4. $((f) \bullet (t))$

1.2 关系

关系是由有序对[1]构成的集合。例如，我们可以定义关系 $\equiv$ 去让每个 B 中的元素和它自身匹配：

$$a \in B \quad \Rightarrow \quad \langle a, a\rangle \in \equiv$$

对类似 $\equiv$ 的二元关系，我们通常写成 $a \equiv a$ 而不是 $\langle a, a\rangle \in \equiv$。

$$a \in B \quad \Rightarrow \quad a \equiv a$$

或者更简单的

$$B_1 \equiv B_1$$

只要该表达式可以被理解为 $\equiv$ 的定义。事实证明，关系 $\equiv$ 是自反的、对称的和传递的。

关系 $\mathbf{R}$ 是**自反的**当且仅当 $a\ \mathbf{R}\ a$ (对任何a)
关系 $\mathbf{R}$ 是**对称的**当且仅当 $a\ \mathbf{R}\ b \Rightarrow b\ \mathbf{R}\ a$
关系 $\mathbf{R}$ 是**传递的**当且仅当 $a\ \mathbf{R}\ b$ 且 $b\ \mathbf{R}\ c \Rightarrow a\ \mathbf{R}\ c$

如果一个关系是自反的，对称的且传递的，那么这个关系也是**等价的**。

下面定义了关系 r，既不是自反的，传递的，也不是等价的。

$$\begin{array}{llll} (f \bullet B_1) & \mathbf{r} & B_1 & [a] \\ (t \bullet B_1) & \mathbf{r} & t & [b] \end{array}$$

根据 r，我们可以定义新的关系 $\asymp_{\mathbf{r}}$，且加入一个限制使 $\asymp_{\mathbf{r}}$ 自反。

$$\begin{array}{rlll} (f \bullet B_1) & \asymp_{\mathbf{r}} & B_1 & [a] \\ (t \bullet B_1) & \asymp_{\mathbf{r}} & t & [b] \\ B_1 & \asymp_{\mathbf{r}} & B_1 & [c] \end{array}$$

关系 $\asymp_{\mathbf{r}}$ 是 r 的**自反闭包**，我们可以加入对称和传递限制来构造一个新的关系：

$$\begin{array}{rlll} (f \bullet B_1) & \approx_{\mathbf{r}} & B_1 & [a] \\ (t \bullet B_1) & \approx_{\mathbf{r}} & t & [b] \\ B_1 & \approx_{\mathbf{r}} & B_1 & [c] \\ B_1 \approx_{\mathbf{r}} B_2 & \Rightarrow & B_2 \approx_{\mathbf{r}} B_1 & [d] \\ B_1 \approx_{\mathbf{r}} B_2 \text{ 且 } B_2 \approx_{\mathbf{r}} B_3 & \Rightarrow & B_1 \approx_{\mathbf{r}} B_3 & [e] \end{array}$$

$\approx_{\mathbf{r}}$ 是 $\asymp_{\mathbf{r}}$ 的**对称传递闭包**，也是 r 的**自反对称传递闭包**或者**等价闭包**。

[1]有序对 $\langle x, y\rangle$ 可以表示成 $\{\{x\}, \{x, y\}\}$ 的集合

1.3 求值关系

B 和 r 的实例应该能让你理解编程语言是如何通过文本和基于文本的关系定义的，或者更准确的说，是如何通过集合 (B) 和基于集合 (r) 的关系定义的。

事实上，你也许开始猜测 B 是布尔表达式的文法，其中 $\bullet$ 对应“or”，$\approx_{\mathbf{r}}$ 等同于含有相同布尔值的表达式。

实际上，基于上述约束，我们可以证明 $(f \bullet t) \approx_{\mathbf{r}} (t \bullet t)$，就像 $\mathbf{false} \vee \mathbf{true} = \mathbf{true} \vee \mathbf{true}$：

$$\dfrac{(f \bullet t) \approx_{\mathbf{r}} t \ [a] \qquad \dfrac{(t \bullet t) \approx_{\mathbf{r}} t \ [b]}{t \approx_{\mathbf{r}} (t \bullet t) \ [d]}}{(f \bullet t) \approx_{\mathbf{r}} (t \bullet t) \qquad [e]}$$

但是，这并不表明 $\bullet$ 和“or”完全一致。相反，我们必须证明关于 $\bullet$ 的一般推论，如证明对任何 B_1 有 $(B_1 \bullet t) \approx_{\mathbf{r}} t$（我们很快就会知道，事实上这是不可能被证明的）。

换言之，一个解释器定义的语言（即使解释器是用数学语言定义的）和我们可能想要保证的语言性质之间通常存在一定差距。对不同的目的而言，语言的性质和它所计算出的值同样重要。例如，如果 $\bullet$ 真的和“or”行为一致，那么编译器可以安全地将 $(B \bullet t)$ 优化成 t。类似的，如果某种语言的语义规则可以保证数值不能和任何非数值相加，那么该语言的实现就不必通过检查加法表达式的参数，来确保二者都是数值。

但是，我们首先必须消除定义语言的方式和定义一般求值的方式之间的“元差距”，才能开始处理这些差距。

1.4 有向求值

“求值”规则 $\approx_{\mathbf{r}}$ 并不完全具有求值的特征。它允许我们证明某些表达式是等价的，却不能告诉我们如何从任意一个 B_1 推导出 t 或 f。

从这个角度出发，更简单的关系 r 是更有效的。r 的定义中两种情况都可将一个表达式映射成一个更简洁的表达式。同样，对任意的表达式 B，无论 B 是 t 或 f，r 都可以使 B 关联到最多一个的其他表达式。因此，我们可以认为 r 是一步规约，对应着解释器对最终结果进行一次可能的单步求值的方式。

我们可以定义 $\rightsquigarrow_{\mathbf{r}}$ 为 r 的自反传递闭包，并且最终得到一个**多步规约**关系。多步关系 $\rightsquigarrow_{\mathbf{r}}$ 会将每个表达式映射成多个其他表达式。尽管如此，事实上，$\rightsquigarrow_{\mathbf{r}}$ 会将每个表达式映射成最多一个 t 或者 f。

> 这足以将 $\rightsquigarrow_{\mathbf{r}}$ 定义成“r 的自反传递闭包”。像上一节一样，一个交替的公式可以拓展成若干约束。第三种方式是部分拓展现有约

束，但需要使用 r 来定义 $\rightsquigarrow_{\mathbf{r}}$。

$$\begin{array}{rcl} B_1 & \rightsquigarrow_{\mathbf{r}} & B_1 \\ B_1 \ \mathbf{r}\ B_2 & \Rightarrow & B_1 \rightsquigarrow_{\mathbf{r}} B_2 \\ B_1 \rightsquigarrow_{\mathbf{r}} B_2 \text{ 且 } B_2 \rightsquigarrow_{\mathbf{r}} B_3 & \Rightarrow & B_1 \rightsquigarrow_{\mathbf{r}} B_3 \end{array}$$

关系 r 和 $\rightsquigarrow_{\mathbf{r}}$ 是有意不对称的，相当于需要对一个值向其特定的方向进行求值。例如对表达式 $(f \bullet (f \bullet (t \bullet f)))$，我们可以展示一个向 t 的**规约**：

$$\begin{array}{ll} (f \bullet (f \bullet (t \bullet f))) & \mathbf{r}\ \ (f \bullet (t \bullet f)) \\ & \mathbf{r}\ \ (t \bullet f) \\ & \mathbf{r}\ \ t \end{array}$$

左边列的每个空行默认可以被前一行的右侧列补充，然后每一行都是证明 $(f \bullet (f \bullet (t \bullet f))) \rightsquigarrow_{\mathbf{r}} t$ 的一步。

▷ **练习 1.2.** 证明 $(f \bullet (f \bullet (f \bullet f))) \rightsquigarrow_{\mathbf{r}} f$，通过 **r** 的一步关系证明其归约即可。

1.5 上下文求值

表达式 $((f \bullet t) \bullet f)$ 如何规约？根据 r 或者 $\rightsquigarrow_{\mathbf{r}}$ 的约束，它根本不会规约！

直觉上，通过 $(f \bullet t)\ r\ f$ 将第一个子表达式简化，$((f \bullet t) \bullet f)$ 应该规约成 $(t \bullet f)$。但是在 r 的定义中，并没有符合将 $((f \bullet t) \bullet f)$ 作为原表达式的约束。我们只能规约形式为 $(f \bullet B)$ 和 $(t \bullet B)$ 的表达式。换句话说，中间 $\bullet$ 右边列的表达式可以是任意的，但是左边列的表达式必须是 f 或者 t。

我们可以使用 $\rightarrow_{\mathbf{r}}$ 拓展关系 r 以支持子表达式的规约：

$$\begin{array}{rcll} B_1 \ \mathbf{r}\ B_2 & \Rightarrow & B_1 \rightarrow_{\mathbf{r}} B_2 & [a] \\ B_1 \rightarrow_{\mathbf{r}} B_1' & \Rightarrow & (B_1 \bullet B_2) \rightarrow_{\mathbf{r}} (B_1' \bullet B_2) & [b] \\ B_2 \rightarrow_{\mathbf{r}} B_2' & \Rightarrow & (B_1 \bullet B_2) \rightarrow_{\mathbf{r}} (B_1 \bullet B_2') & [c] \end{array}$$

关系 $\rightarrow_{\mathbf{r}}$ 是 r 的**合拍闭包**。类似 r，$\rightarrow_{\mathbf{r}}$ 也是一步关系，但是 $\rightarrow_{\mathbf{r}}$ 允许对整个表达式中任意子表达进行规约。r 规约的子表达式称作**可约项**，在**可约项**周围的文本称作**上下文**。

需要特别指出的是，$\rightarrow_{\mathbf{r}}$ 关系包含 $((f \bullet t) \bullet f) \rightarrow_{\mathbf{r}} (t \bullet f)$。我们通过下面的证明树证明这个包含：

$$\dfrac{\dfrac{(f \bullet t)\ \mathbf{r}\ t}{(f \bullet t) \rightarrow_{\mathbf{r}} t\ [a]}}{((f \bullet t) \bullet f) \rightarrow_{\mathbf{r}} (t \bullet f)\ [b]}$$

继续使用关系 $\to_{\mathbf{r}}$，我们可以规约 $((f \bullet t) \bullet f)$ 到 t：

$$\begin{array}{lll} ((f \bullet t) \bullet f) & \to_{\mathbf{r}} & (t \bullet f) \\ & \to_{\mathbf{r}} & t \end{array}$$

最后，如果我们定义 $\twoheadrightarrow_{\mathbf{r}}$ 作为 $\to_{\mathbf{r}}$ 的自反传递闭包，我们可以得到 $((f \bullet t) \bullet f) \twoheadrightarrow_{\mathbf{r}} t$。因此，$\twoheadrightarrow_{\mathbf{r}}$ 是根据 r 自然生成的**规约关系**。

总的来说，纯粹的 r 的自反闭包 $\asymp_{\mathbf{r}}$，等价闭包 $\approx_{\mathbf{r}}$ 或者自反传递闭包 $\rightsquigarrow_{\mathbf{r}}$ 是很乏味的。相反，我们通常会对合拍闭包 $\to_{\mathbf{r}}$ 以及它的自反传递闭包 $\twoheadrightarrow_{\mathbf{r}}$ 感兴趣，因为它们符合经典的求值概念。另外，$\to_{\mathbf{r}}$ 的等价闭包 $=_{\mathbf{r}}$ 也因关联了相同结果的表达式而耐人寻味。

▷ **练习 1.3.** 解释为什么 $(f \bullet ((t \bullet f) \bullet f)) \not\rightsquigarrow_{\mathbf{r}} t$。

▷ **练习 1.4.** 通过证明关于 $\to_{\mathbf{r}}$ 归约的命题来证明 $(f \bullet ((t \bullet f) \bullet f)) \twoheadrightarrow_{\mathbf{r}} t$。

1.6 求值函数

虽然 $\twoheadrightarrow_{\mathbf{r}}$ 让我们更加了解实用的求值概念，但我们仍没有真正的了解它。不仅存在 $((f \bullet t) \bullet f) \twoheadrightarrow_{\mathbf{r}} t$，同样也有 $((f \bullet t) \bullet f) \twoheadrightarrow_{\mathbf{r}} (t \bullet f)$ 和 $((f \bullet t) \bullet f) \twoheadrightarrow_{\mathbf{r}} ((f \bullet t) \bullet f)$ 两种情况。

对求值而言，我们关心的是 B 求值成 f 还是 t，任何使用 $\twoheadrightarrow_{\mathbf{r}}$ 或 $=_{\mathbf{r}}$ 的映射都与此无关。为了理解求值的概念，我们如下方式定义关系 $eval_{\mathbf{r}}$：

$$eval_{\mathbf{r}}(B) = \begin{cases} f & \text{如果 } B =_{\mathbf{r}} f \\ t & \text{如果 } B =_{\mathbf{r}} t \end{cases}$$

此处我们还是在用另一种概念去定义关系。这种概念让我们联想起**函数**，即将一个元素映射成另一个元素的关系。我们使用函数概念因为 $eval_{\mathbf{r}}$ 如果想要成为一个求值器则必须先成为一个函数。

▷ **练习 1.5.** 在 $\mathbf{r}, \asymp_{\mathbf{r}}, \approx_{\mathbf{r}}, \rightsquigarrow_{\mathbf{r}}, \to_{\mathbf{r}}, \twoheadrightarrow_{\mathbf{r}}, =_{\mathbf{r}}$ 和 $eval_{\mathbf{r}}$ 这些关系中，哪些是函数？为每一个非函数关系都找到一个表达式和它关联的两个表达式。

1.7 概念汇总

符号	定义	解释
$-$	表达式文法中的基本关系	没有上下文的一步“规约”
$\rightarrow\_$	表达式文法 _ 的合拍闭包	上下文的一步规约
$\twoheadrightarrow\_$	$\rightarrow\_$ 的自反传递闭包	多步规约（0 或者多步）
$=\_$	$\twoheadrightarrow\_$ 的对称传递闭包	产生相同结果的等价表达式
$eval\_$	限制结果的 $=\_$	完整的求值

第 2 章

求值的一致性

既然我们有了结构化指令，我们准备返回到 $eval_{\mathbf{r}}$ 作为求值器的一致性话题上。换言之，我们想证明 $eval_{\mathbf{r}}$ 是函数。更加形式化的说，给出结果集 R：

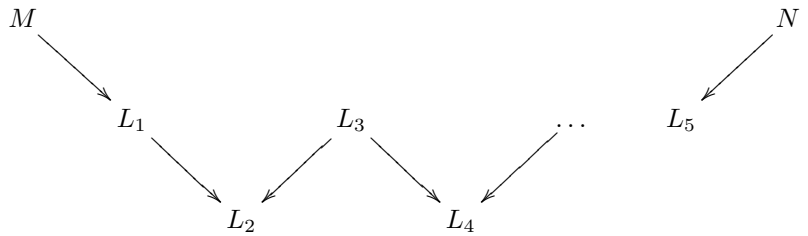
$$\begin{array}{rcl} R & = & t \\ & | & f \end{array}$$

我们想证明如下定理：

定理 2.1 如果 $eval_{\mathbf{r}}(B_0) = R_1$ 且 $eval_{\mathbf{r}}(B_0) = R_2$，那么 $R_1 = R_2$。

为了证明该定理，我们假设对某 B_0、R_1 和 R_2，有 $eval_{\mathbf{r}}(B_0) = R_1$ 且 $eval_{\mathbf{r}}(B_0) = R_2$，而想推出 $R_1 = R_2$。根据 $eval_{\mathbf{r}}$ 的定义，我们的假设意味着 $B_0 =_{\mathbf{r}} R_1$ 且 $B_0 =_{\mathbf{r}} R_2$（使用 $=_{\mathbf{r}}$ 而非 $=$）。因此，根据 $=_{\mathbf{r}}$ 是等价关系的定义可得 $R_1 =_{\mathbf{r}} R_2$。为了得出 $R_1 = R_2$ 的结论，我们必须探讨求值的本质，即当 $M, N \in B$ 时，$M =_{\mathbf{r}} N$ 证明的一般式。

因为 $=_{\mathbf{r}}$ 是单步规约 $\rightarrow_{\mathbf{r}}$ 的拓展形式，所以证明 $M =_{\mathbf{r}} N$ 的求值通常是基于 r 的一系列单步双向规约：



其中对每个 $L_{\mathbf{n}}$ 有 $L_{\mathbf{n}} \in B$。也许这些步骤可以重排，使得所有归约步骤从 M 到达某个 L 且从 N 到达同样的 L。换言之，如果 $M =_{\mathbf{r}} N$ 则可能存在一个表

达式 L，使 $M \twoheadrightarrow_L$ 且 $N \twoheadrightarrow_L$。

我们如果可以证明总是存在这样的 L，也就完成了一致性的证明。回想我们的断言

$$R_1 =_{\mathbf{r}} R_2$$

根据该断言（尚未证明），一定存在 L 使

$$R_1 \twoheadrightarrow_{\mathbf{r}} L \quad 且 \quad R_2 \twoheadrightarrow_{\mathbf{r}} L$$

但是 R 中的元素只有 t 和 f，显然不可能规约成除了他们自身的任何值。所以 $L = R_1$ 且 $L = R_2$，意味着 $R_1 = R_2$。

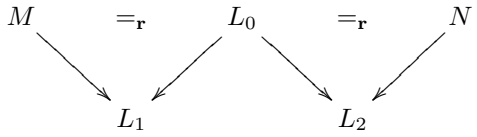
根据前面的论述, 我们已经将 $eval_{\mathbf{r}}$ 的一致性证明规约成证明 $M =_{\mathbf{r}} N$ 的论据形态的断言。这个关于形式等价系统的一致性证明和规约步骤重排之间的联系的重要见解归功于丘奇和罗瑟，二者将这个思路用于分析称为 λ 演算的语言（我们很快会学到）的一致性问题。一般而言，如果一个由基本规约概念按一定条件产生的等价关系满足重排性质，则它满足丘奇罗瑟性质。

定理 2.2 (基于 $=_{\mathbf{r}}$ 的丘奇罗瑟定理) 如果 $M =_{\mathbf{r}} N$，那么存在表达式 L 使 $M \twoheadrightarrow_{\mathbf{r}} L$ 且 $N \twoheadrightarrow_{\mathbf{r}} L$。

因为我们有特定的 $M =_{\mathbf{r}} N$，而 $=_{\mathbf{r}}$ 是递归定义的，所以可以通过对 $M =_{\mathbf{r}} N$ 的推导结构进行归纳来证明该定理。

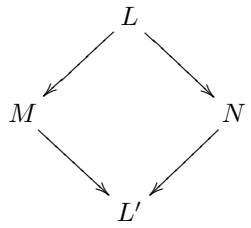
定理 2.2 的证明：对 $M =_{\mathbf{r}} N$ 证明结构的归约

- 奠基步骤
  - **当** $M \twoheadrightarrow_{\mathbf{r}} N$
    令 $L = N$，得证。
- 归纳步骤
  - **根据** $N =_{\mathbf{r}} M$，**有** $M =_{\mathbf{r}} N$
    根据归纳，L 存在于 $N =_{\mathbf{r}} M$，所以该 L 即为所求。
  - **根据** $M =_{\mathbf{r}} L_0$, $L_0 =_{\mathbf{r}} N$，**有** $M =_{\mathbf{r}} N$
    根据归纳，存在 L_1 使 $M \twoheadrightarrow_{\mathbf{r}} L_1$ 且 $L_0 \twoheadrightarrow_{\mathbf{r}} L_1$。再次根据归纳，存在 L_2 使 $N \twoheadrightarrow_{\mathbf{r}} L_2$ 且 $L_0 \twoheadrightarrow_{\mathbf{r}} L_2$。如图所示，我们有：



    现在我们假设，无论 L_0 何时同时规约到 $L1$ 和 L_2，都存在 L_3 使得 $L_1 \twoheadrightarrow_{\mathbf{r}} L_3$ 且 $L_2 \twoheadrightarrow_{\mathbf{r}} L_3$。然后我们想证明的断言即得证，因为 $M \twoheadrightarrow_{\mathbf{r}} L_3$ 且 $N \twoheadrightarrow_{\mathbf{r}} L_3$。

我们再一次完成了这个证明，但留下了另一个关于归约系统的断言有待证明。这个新性质叫做**菱形性质**，因为这个定理的形状需要规约能按照菱形排列：



定理 2.3 ($\twoheadrightarrow_{\mathbf{r}}$ 的菱形性质) 如果 $L \twoheadrightarrow_{\mathbf{r}} M$ 且 $L \twoheadrightarrow_{\mathbf{r}} N$，那么存在表达式 L' 使得 $M \twoheadrightarrow_{\mathbf{r}} L'$ 且 $N \twoheadrightarrow_{\mathbf{r}} L'$。

为了证明这个定理，我们可以先证明 $\to_{\mathbf{r}}$ 的类菱形定理。

引理 2.4 ($\to_{\mathbf{r}}$ 的类菱形定理) 如果 $L \to_{\mathbf{r}} M$ 且 $L \to_{\mathbf{r}} N$，那么要么有

- $M = N$，
- $M \to_{\mathbf{r}} N$，
- $N \to_{\mathbf{r}} M$，要么有
- 存在 L' 使 $M \to_{\mathbf{r}} L'$ 且 $N \to_{\mathbf{r}} L'$。

为了证明这个引理，我们考虑条件中的 $L \to_{\mathbf{r}} M$，注意 $\to_{\mathbf{r}}$ 是递归定义的：

引理 2.4 的证明：通过归纳 $L \to_{\mathbf{r}} M$ 的证明结构。

- 奠基步骤
  - **当** $L\ r\ M$ **时**
    根据 r 的定义，可分为以下两种情况。
    * **当** $L = (f \bullet B_0)$ **且** $M = B_0$ **时**
      表达式 L 可以以两种形式通过 $\to_{\mathbf{r}}$ 规约到 N：直接到 B_0 或者通过 $B_0 \to_{\mathbf{r}} B_1$ 到 $(f \bullet B_0')$。
      如果 $N = B_0$，那么 $M = N$ 得证。
      否则，$N = (f \bullet B_0')$。然后因为 $M = B_0, M \to_{\mathbf{r}} B_0$，那么根据 R 的定义，有 $N \to_{\mathbf{r}} B_0$，因此可证 $L' = B_0'$。
    * **当** $L = (t \bullet B_0)$ **且** $M = t$ **时**
      类似上一种情况，要么 $N = M$，要么 $N = (t \bullet B_0')$，所以可证 $N \to_{\mathbf{r}} M$
- 归纳步骤：不失一般性，假设 $L \not\to_{\mathbf{r}} N$（否则交换 M 和 N）
  - **当** $L = (B_1 \bullet B_2), B_1 \to_{\mathbf{r}} B_1'$ **且** $M = (B_1' \bullet B_2)$ **时**
    有两种子情况：

  * $N = (B_1'' \bullet B_2)$，其中 $(B_1 \to_{\mathbf{r}} B_1'')$。因为 $B_1 \to_{\mathbf{r}} B_1'$ 且 $(B_1 \to_{\mathbf{r}} B_1'')$，所以我们可以使用归纳。如果 $B_1' = B_1''$，那么 $M = N$，命题得证。如果 $B_1'' \to_{\mathbf{r}} B_1'$ 那么有 $M \to_{\mathbf{r}} N$，命题得证。类似的，如果 $B_1' \to_{\mathbf{r}} B_1''$，那么 $N \to_{\mathbf{r}} M$，命题得证。最后，如果 $B_1' \to_{\mathbf{r}} B_1'''$ 且 $B_1'' \to_{\mathbf{r}} B_1'''$，则 $M \to_{\mathbf{r}} (B_1''' \bullet B_2)$ 且 $N \to_{\mathbf{r}} (B_1''' \bullet B_2)$，那么由 $L' = (B_1''' \bullet B_2)$ 可知命题得证。

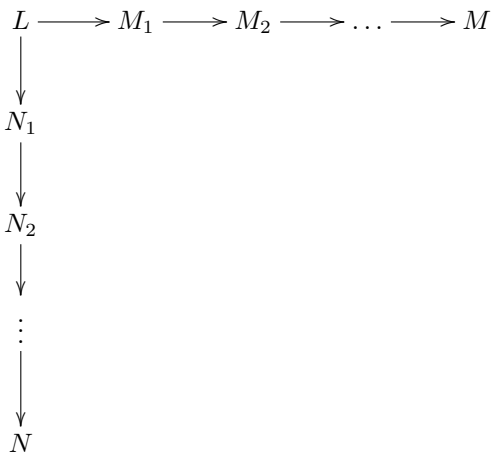
  * $N = (B_1 \bullet B_2')$，其中 $(B_2 \to_{\mathbf{r}} B_2')$。因为 $(B_1 \to_{\mathbf{r}} B_1')$，所以有 $N \to_{\mathbf{r}} (B_1' \bullet B_2')$。类似的，有 $M \to_{\mathbf{r}} (B_1' \bullet B_2')$。因此由 $L' = (B_1' \bullet B_2')$ 可知命题得证。

- **当 $L = (B_1 \bullet B_2), B_2 \to_{\mathbf{r}} B_2'$ 且 $M = (B_1 \bullet B_2')$ 时**
  证明过程同上。

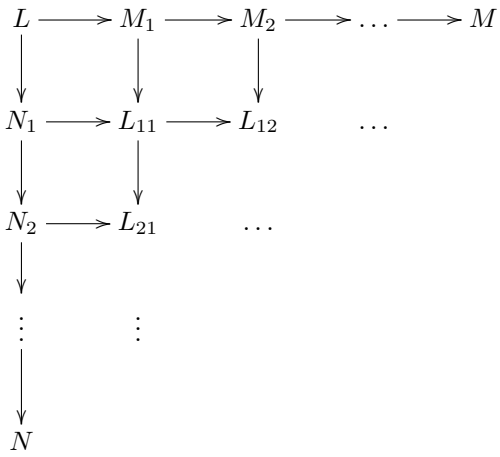
现在我们知道一步规约满足类菱形性质，我们就可以证明它的传递自反闭包也满足菱形性质。假设 $L \twoheadrightarrow_{\mathbf{r}} M$ 且 $L \twoheadrightarrow_{\mathbf{r}} N$。根据规约关系 $\twoheadrightarrow_{\mathbf{r}}$ 的归纳定义有：对 $m, n \in M$，

$$L \to_{\mathbf{r}}^{m} M \quad \text{且} \quad L \to_{\mathbf{r}}^{m} N$$

其中 $\to_{\mathbf{r}}^{m}$ 代表 $\to_{\mathbf{r}}$ 的 m 步规约。如图，我们有：



使用一步规约的类菱形规则，我们可以填充表达式 L_{11}, L_{21}, L_{12} 等。直到整个矩形被填满：



从形式化的角度看，这个思路也可以转换成归纳证明。

上面的论证也证明了当且仅当 $M \twoheadrightarrow_{\mathbf{r}} R$ 时有 $M =_{\mathbf{r}} R$。因此，我们可以不失一般性的通过规约定义演算。换言之，对称规则对 B 表达式的演算没有帮助。但是，在下一章中我们会介绍一种可使用明显的回退步骤，并真正意义上简化程序求值计算的编程语言。

定义了一个程序一定有唯一值之后，接下来的问题是，程序是否一定有值。

定理 2.5 对任意 B_0，存在 R_0 使得 $eval_{\mathbf{r}}(B_0) = R_0$。

这个定理可以得出，存在一种可以确定 B 表达式等式结果的算法：对等式两边求值并比较。实际的编程语言中的表达式是任意且无穷的，因此程序员不必通过这种方式确定表达式等式。

▷ **练习 2.1.** 证明定理 2.3（不使用图表，而是形式化的证明）。

▷ **练习 2.2.** 证明定理 2.5。

第 3 章

λ 演算

对于任何实际的编程任务而言，B 语言因限制太多而不实用。因为它没有任何抽象能力——即没有定义函数的能力——它远没有现实中的编程语言那么强大。

我们在本章中将学习一种叫做 λ 演算的语言，这个语言由丘奇发明。尽管 λ 演算的语义和规约关系都是最简的，但是它和实用语言（如 Scheme 和 ML）是紧密相关的。在这些语言中，函数不仅操作布尔值、整数和对，也可以操作函数。换言之，函数即为值。例如，map 函数接受一个函数和一个元素列表，并将该函数应用到每个列表中的元素里。类似的，derivation 函数接受一个（关于实数的）函数，并返回一个实现其导数的新的函数。

在 λ 演算中，函数是唯一的值，但是我们将看到如何使用函数定义各类值（包括布尔，整数和对）。

3.1 λ 演算中的函数

λ 演算的语义提供了一种简单、正规的写出应用函数的方法，它的输入、输出也是其他函数。λ 演算中的函数规则注重表达参数如何变成结果，而忽略函数的命名和它的域以及范围等问题。例如，数学家会用如下方式详述基于集合 A 的恒等函数，如

$$\forall x \in A, f(x) = x$$

或

$$f : \begin{cases} A & \longrightarrow & A \\ x & \mapsto & x \end{cases}$$

而在 λ 演算语义中，我们会写成

$$(\lambda x.x)$$

对表达式 $(\lambda x.x)$ 的信息解读是“如果参数被称为 x，那么函数的输出也是 x”。换句话说，这个函数只输出其输入的数据。

为了写出函数 f 对参数 a 的应用，λ 演算使用了常规的数学语义，并保留了括号：

$$(f\ a)$$

例如，表示恒等函数对 a 应用的表达式可写成

$$((\lambda x.x)\ a)$$

恒等函数的另一个可能的参数是它本身：

$$((\lambda x.x)\ (\lambda x.x))$$

下面是一个表示函数的表达式。它接受一个参数后会忽略之，同时返回该恒等函数：

$$(\lambda y.(\lambda x.x))$$

下面是一个表示函数的表达式，该函数接受一个参数并返回一个函数；且这个被返回的会函数忽略自己的参数，并返回原始函数的参数：

$$(\lambda y.(\lambda x.y))$$

λ 演算只支持单一参数的函数，但是上一个例子展示了一个函数如何能有效的接受两个函数：x 和 y，通过接受它的第一个参数，并返回另一个函数去获得第二个参数。这种技术叫做**柯里化**。

在传统的数学概念中，$f(a)$ 可以被“简化”，方式是通过使用表达式来定义 f，并将任何位置的 f 的形式参数替换掉。例如，给出表达式 $f(x) = x$，$f(a)$ 可以被化简成 a。λ 演算项的化简也与之类似：$((\lambda x.x)\ a)$ 可以被化简成 a，它接受的是 $(\lambda x.x)$ 的主体部分（即点之后的部分）之后的结果，并将形式参数 x（即点之前的部分）替换成实际参数 a。下面是一些例子：

$((\lambda x.x)\ a)$	$\rightarrow$	a	根据 x，使用 a 替换 x
$((\lambda x.x)\ (\lambda y.y))$	$\rightarrow$	$(\lambda y.y)$	根据 x，使用 $(\lambda y.y)$ 替换 x
$((\lambda y.(\lambda x.y))\ a)$	$\rightarrow$	$(\lambda x.a)$	根据 $\lambda x.y$，使用 a 替换 y

3.2 λ 演算的文法和规约

λ 演算中的一般表达式文法是由 M（别名 N 或 L）定义的:

$$\begin{array}{lcl} M, N, L & = & X \\ & | & (\lambda X.M) \\ & | & (M\ M) \\ X & = & \text{一个变量：}\ x, y, \ldots \end{array}$$

以下是 M 元素的例子：

$$\begin{array}{ccc} x & (x\ y) & ((x\ y)\ (z\ w)) \\ (\lambda x.x) & & (\lambda y.(\lambda z.y)) \\ (f(\lambda y.(y\ y))) & & ((\lambda y.(y\ y))\ (\lambda y.(y\ y))) \end{array}$$

在第一个例子中，由于 x 未被定义，所以没有特别的直观含义。类似地，$(x\ y)$ 表示“应用于 y 的 x”，但是没有其他含义，因为 x 和 y 都是未定义的。相反，例子 $(\lambda x.x)$ 对应恒等函数。它和前两个例子的区别在于：x 在前两个表达式中是**自由的**，但是在最后的例子中仅以**绑定的**形式出现。

关系 $\mathcal{FV}$，将一个表达式映射成该表达式中**自由变量**的集合。直观上说，如果 x 出现在 $(\lambda x.\_)$ 之外任意位置，那么它就是一个自由变量。更加形式化的说，我们可使用如下方式定义关系 $\mathcal{FV}$：

$$\begin{array}{lcl} \mathcal{FV}(\ X\ ) & = & \{X\} \\ \mathcal{FV}(\ (\lambda X.M)\ ) & = & \mathcal{FV}(M) \setminus X \\ \mathcal{FV}(\ (M_1\ M_2)\ ) & = & \mathcal{FV}(M_1) \cup \mathcal{FV}(M_2) \end{array}$$

$\mathcal{FV}$ 的例子：

$$\begin{array}{lcl} \mathcal{FV}(\ x\ ) & = & \{x\} \\ \mathcal{FV}(\ (x\ (y\ x))\ ) & = & \{x, y\} \\ \mathcal{FV}(\ (\lambda x.(x\ y))\ ) & = & \{y\} \\ \mathcal{FV}(\ (z\ (\lambda z.z))\ ) & = & \{z\} \end{array}$$

在定义 λ 演算表达式的规约关系之前，我们还须一个辅助关系去处理变量替换。关系 $\_[\_ \leftarrow \_]$ 映射一个源表达式，一个变量和一个参数表达式到一个目标表达式。目标表达式和源表达式一样，唯一不同的是自由变量都会被替换成参数表达式：

$$\begin{array}{lcl} X_1[X_1 \leftarrow M] & = & M \\ X_2[X_1 \leftarrow M] & = & X_2 \\ & & 其中 X_1 \neq X_2 \\ (\lambda X_1.M_1)[X_1 \leftarrow M_2] & = & (\lambda X_1.M_1) \\ (\lambda X_1.M_1)[X_2 \leftarrow M_2] & = & (\lambda X_3.M_1[X_1 \leftarrow X_3][X_2 \leftarrow M_2]) \\ & & 其中 X_1 \neq X_2, X_3 \notin \mathcal{FV}(M_2) \\ & & 且 X_3 \notin \mathcal{FV}(M_1) \backslash \{X_1\} \\ (M_1\ M_2)[X \leftarrow M_3] & = & (M_1[X \leftarrow M_3]\ M_2[X \leftarrow M_3]) \end{array}$$

关于 _[_ ← _] 的例子：

$$\begin{array}{lcl}
x[x \leftarrow (\lambda y.y)] & = & (\lambda y.y) \\
z[x \leftarrow (\lambda y.y)] & = & z \\
(\lambda x.x)[x \leftarrow (\lambda y.y)] & = & (\lambda x.x) \\
(\lambda y.(x\ y))[x \leftarrow (\lambda y.y)] & = & (\lambda z.((\lambda y.y)\ z)) \text{ 或 } (\lambda y.((\lambda y.y)\ y)) \\
(\lambda y.(x\ y))[x \leftarrow (\lambda x.y)] & = & (\lambda z.((\lambda x.y)\ z))
\end{array}$$

最后，为了定义 λ 演算的一般规约关系 n，我们首先定义三个简单的规约关系 α，β 和 η：

$$\begin{array}{llll}
(\lambda X_1.M) & \alpha & (\lambda X_2.M[X_1 \leftarrow X_2]) & \text{其中} X_2 \notin \mathcal{FV}(M) \\
((\lambda X_1.M_1)\ M_2) & \beta & M_1[X \leftarrow M_2] & \\
(\lambda X.(M\ X)) & \eta & M & \text{其中} X \notin \mathcal{FV}(M)
\end{array}$$

- α 关系将形式参数重命名。它对如下事实进行编码：类似 $(\lambda x.x)$ 和 $(\lambda y.y)$ 的函数是同一个函数，唯一的差别是参数使用了不同的命名。
- β 关系是主要的规约关系，用于编码函数应用。
- η 关系对如下事实进行编码：如果函数 f 接受一个参数，并且马上将该参数应用于 g，则使用函数 f 等于使用函数 g。

一般规约关系 n 是 α、β 和 η 的并集：

$$n = \alpha \cup \beta \cup \eta$$

通常，我们将 $\rightarrow_{\mathbf{n}}$ 定义为 n 的合拍闭包，将 $\twoheadrightarrow_{\mathbf{n}}$ 定义为 $\rightarrow_{\mathbf{n}}$ 的自反传递闭包，将 $=_{\mathbf{n}}$ 定义为 $\twoheadrightarrow_{\mathbf{n}}$ 的对称闭包。我们也将 $\rightarrow^{\alpha}_{\mathbf{n}}$，$\rightarrow^{\beta}_{\mathbf{n}}$，$\rightarrow^{\eta}_{\mathbf{n}}$ 分别定义为 α，β 和 η 的合拍闭包（α 的合拍闭包一般写成 $\rightarrow_{\alpha}$，但是我们使用 $\rightarrow^{\alpha}_{\mathbf{n}}$ 以强调 $\rightarrow_{\mathbf{n}}=\rightarrow^{\alpha}_{\mathbf{n}} \cup \rightarrow^{\beta}_{\mathbf{n}} \cup \rightarrow^{\eta}_{\mathbf{n}}$）。

下面是 $((\lambda x.((\lambda z.z)\ x))\ (\lambda x.x))$ 的多种可能的规约方式中的一种，其中表达式带下划线的部分是每一步的可约项（被 n 规约的部分）：

$$\begin{array}{lll}
((\lambda x.((\lambda z.z)\ x))\ \underline{(\lambda x.x)}) & \rightarrow^{\alpha}_{\mathbf{n}} & \underline{((\lambda x.((\lambda z.z)\ x))}\ (\lambda y.y) \\
& \rightarrow^{\eta}_{\mathbf{n}} & \underline{((\lambda z.z)\ (\lambda y.y))} \\
& \rightarrow^{\beta}_{\mathbf{n}} & (\lambda y.y)
\end{array}$$

以下是上述同一个表达式的另一种规约方法：

$$\begin{array}{lll}
((\lambda x.\underline{((\lambda z.z)\ x)})\ (\lambda x.x)) & \rightarrow^{\beta}_{\mathbf{n}} & \underline{((\lambda x.x)\ (\lambda x.x))} \\
& \rightarrow^{\beta}_{\mathbf{n}} & (\lambda x.x)
\end{array}$$

表达式中的括号通常是冗余的。它们会使原本就冗长的表达式变得更加难以理解。因此，我们使用一些规则以代替括号，另外还有一个规则可以消除 λ：

- 应用是左关联的：$M_1\ M_2\ M_3$ 意味着 $((M_1\ M_2)\ M_3)$
- 应用是强于抽象的：$\lambda X.M_1\ M_2$ 意味着 $(\lambda X.(M_1\ M_2))$
- 连续的 λ 可以被折叠：$\lambda X\ Y\ Z.M$ 意味着 $(\lambda X.(\lambda Y.(\lambda Z.M)))$

根据这些规则，$((\lambda x.((\lambda z.z)\ x))\ (\lambda x.x))$ 可以被缩写成 $(\lambda x.(\lambda z.z)x)\ \lambda x.x$，而上述第一个规约则可以被缩写成：

$$\begin{array}{lll}(\lambda x.(\lambda z.z)\ x)\ \underline{\lambda x.x} & \to^{\alpha}_{\mathbf{n}} & \underline{(\lambda x.(\lambda z.z)\ x)}\ \lambda y.y \\ & \to^{\eta}_{\mathbf{n}} & \underline{(\lambda z.z)\ \lambda y.y} \\ & \to^{\beta}_{\mathbf{n}} & \lambda y.y\end{array}$$

▷ **练习 3.1.** 使用 $\to_{\mathbf{n}}$ 归约下列表达式，直到再没有任何可能的 $\to^{\beta}_{\mathbf{n}}$ 归约为止。列出所有的步骤。

- $(\lambda x.x)$
- $(\lambda x.(\lambda y.y\ x))\ (\lambda y.y)\ (\lambda x.x\ x)$
- $(\lambda x.(\lambda y.y\ x))\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$

▷ **练习 3.2.** 通过列出归约来证明下面的等式。

- $(\lambda x.x) =_{\mathbf{n}} (\lambda y.y)$
- $(\lambda x.(\lambda y.(\lambda z.z\ z)\ y)\ x)\ (\lambda x.x\ x) =_{\mathbf{n}} (\lambda a.a((\lambda g.g)\ a))\ (\lambda b.b\ b)$
- $\lambda y.(\lambda x.\lambda y.x)\ (y\ y) =_{\mathbf{n}} \lambda a.\lambda b.(a\ a)$
- $(\lambda f.\lambda g.\lambda x.f\ x\ (g\ x))(\lambda x.\lambda y.x)(\lambda x.\lambda y.x) =_{\mathbf{n}} \lambda x.x$

3.3 编码布尔值

在 B 语言中，我们可以任意地将符号 f 和 t 分别对应为“假”和“真”。而在 λ 演算中，我们使用了不同的方法——一种虽然原则上依然是任意的，但事实上很方便的方法。

$$\begin{array}{lcl}\mathbf{true} & \doteq & \lambda x.\lambda y.x \\ \mathbf{false} & \doteq & \lambda x.\lambda y.y \\ \mathbf{if} & \doteq & \lambda v.\lambda t.\lambda f.v\ t\ f\end{array}$$

符号 $\doteq$ 表示我们定义了一个表达式的缩写或者是“宏”。如果方式得当，宏 **true**，**false** 和 **if** 是很实用的。例如，我们期待

$$\textbf{if true}\ M\ N =_{\mathbf{n}} M$$

对任何的 M 和 N 都成立。我们可以通过展开宏来证明等式成立：

$$\begin{array}{rcl}
\textbf{if true}\ M\ N & = & (\lambda v.\lambda t.\lambda f.v\ t\ f)\ (\lambda x.\lambda y.x)\ M\ N \\
& \to^{\beta}_{\mathbf{n}} & (\lambda t.\lambda f.(\lambda x.\lambda y.x)\ t\ f)\ M\ N \\
& \to^{\beta}_{\mathbf{n}} & (\lambda f.(\lambda x.\lambda y.x)\ M\ f)\ N \\
& \to^{\beta}_{\mathbf{n}} & (\lambda x.\lambda y.x)\ M\ N \\
& \to^{\beta}_{\mathbf{n}} & (\lambda y.M)\ N \\
& \to^{\beta}_{\mathbf{n}} & M
\end{array}$$

类似的，对 $\textbf{if false}\ M\ N =_{\mathbf{n}} N$：

$$\begin{array}{rcl}
\textbf{if false}\ M\ N & = & (\lambda v.\lambda t.\lambda f.v\ t\ f)\ (\lambda x.\lambda y.y)\ M\ N \\
& \to^{\beta}_{\mathbf{n}} & (\lambda t.\lambda f.(\lambda x.\lambda y.y)\ t\ f)\ M\ N \\
& \to^{\beta}_{\mathbf{n}} & (\lambda f.(\lambda x.\lambda y.y)\ M\ f)\ N \\
& \to^{\beta}_{\mathbf{n}} & (\lambda x.\lambda y.y)\ M\ N \\
& \to^{\beta}_{\mathbf{n}} & (\lambda y.y)\ N \\
& \to^{\beta}_{\mathbf{n}} & N
\end{array}$$

实际上，事实证明 $(\textbf{if true}) =_{\mathbf{n}} \textbf{true}$ 且 $(\textbf{if false}) =_{\mathbf{n}} \textbf{false}$。换言之，这里 **true** 相当于分支到第一个参数的条件句，而 **false** 的相当于分支到第二个参数的条件句。宏 **if** 仅仅是为了增加可读性。

▷ **练习 3.3.** 证明 $(\textbf{if true}) =_{\mathbf{n}} \textbf{true}$ 且 $(\textbf{if false}) =_{\mathbf{n}} \textbf{false}$。

▷ **练习 3.4.** 定义二元前缀操作符 **and** 和 **or**，使它们以自然的方式对 **true** 和 **false** 求值（使得 $\textbf{and true false} =_{\mathbf{n}} \textbf{false}$ 等）。

3.4 编码对

我们需要三种操作来编码对：一种是结合两个值；一种是提取第一个值；另一种是提取第二个值。换句话说，我们需要函数 **mkpair**,**fst** 和 **snd**，且遵循如下规则：

$$\begin{array}{rcl}
\textbf{fst}\ (\textbf{mkpair}\ M\ N) & =_{\mathbf{n}} & M \\
\textbf{snd}\ (\textbf{mkpair}\ M\ N) & =_{\mathbf{n}} & N
\end{array}$$

我们可以用标识 $\langle M, N\rangle$ 来简写第一个元素是 M 且第二个元素是 N 的对。一种为 **mkpair** 等找到定义的方法是考虑应该如何表示 $\langle M, N\rangle$。

因为我们唯一的值就是函数，所以 $\langle M, N\rangle$ 也必须是函数。这个函数应当内含表达式 M 和 N，且必须有某种方式用于为对的使用者返回其中一个或者另外一个元素，具体取决于使用者想要获取第一个还是第二个元素。这意味着对的

使用者应该像调用函数一样调用对——即提供 **true** 则获取第一个元素，提供 **false** 则获取第二个元素：

$$\langle M, N\rangle \doteq \lambda s.\mathbf{if}\ s\ M\ N$$

根据上一节所述，**if** 函数实际上不是必须的，所以可通过去掉 **if** 来化简上述公式。

根据这个编码，**fst** 函数接受一个对，然后把它当作函数应用到参数 **true**：

$$\mathbf{fst} \doteq \lambda p.p\ \mathbf{true}$$

类似地，**snd** 用它的对参数去应用 **false**。最后，为了定义 **mkpair**，我们将任意的 M 和 N 抽象成简写形式 $\langle M, N\rangle$

$$\begin{array}{lcl}\langle M, N\rangle & \doteq & \lambda s.\mathbf{if}\ s\ M\ N\\ \mathbf{mkpair} & \doteq & \lambda x.\lambda y.\lambda s.s\ x\ y\\ \mathbf{fst} & \doteq & \lambda p.p\ \mathbf{true}\\ \mathbf{snd} & \doteq & \lambda p.p\ \mathbf{false}\end{array}$$

▷ **练习 3.5.** 证明 **makepair**，**fst** 和 **snd** 遵循本节开头的表达式。

3.5 编码数字

在 λ 演算中，编码数字有很多种方式，但是最常用的编码方式是丘奇发明的，编码之后的数也因此而得名**丘奇数**。这个理念是将自然数 n 由一个接受双参数 f 和 x 的函数编码，其中这个函数用 f 应用于 n 次 x。因此 0 的函数接受 f 和 x 且返回 x（对应应用 f 零次），函数 1 应用 f 一次，以此类推。

$$\begin{array}{lcl}0 & \doteq & \lambda f.\lambda x\ x\\ 1 & \doteq & \lambda f.\lambda x\ f\ x\\ 2 & \doteq & \lambda f.\lambda x\ f\ (f\ x)\\ 3 & \doteq & \lambda f.\lambda x\ f\ (f\ (f\ x))\\ & \cdots & \end{array}$$

函数 **add1** 接受数字 n 的编码且输出数字 n+1 的编码。换句话说，它接受一个双参数函数并返回一个双参数函数。新的函数将其第一个参数应用于第二个参数 n+1 次。为了获得前 n 次应用，新的函数可以使用老函数。

$$\textbf{add1} \doteq \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$$

数字的编码类似 **true** 和 **false** 的编码过程，实际上是很方便的。为了使 n 和 m 这两个数相加，我们必须要做的是用 **add1** 应用 n 达到 m 次——且 m 恰好是一个会接受函数 **add1** 且将它应用 m 次的函数！

$$\textbf{add} \doteq \lambda n.\lambda m.m\ \textbf{add1}\ n$$

将数值当作函数使用的思想对定义 **iszero** 也很实用，要将其当做一个接受一个数字，并当这个数字为 0 时才返回真否则返回假的函数。我们可以通过使用一个忽略其本身参数并只返回 **false** 的函数实现 **iszero**，如果这个函数应用了 **true** 零次，则结果是 **true**，否则结果就是 **false**:。

$$\textbf{iszero} \doteq \lambda n.n\ (\lambda x.\textbf{false})\ \textbf{true}$$

为了将 **iszero** 应用到数字等式中，我们需要减法运算。和加法可基于 **add1** 的方式一样，减法也可基于 **sub1**。不过，尽管 **add1**,**add** 和 **iszero** 在丘奇数编码中的定义都很简单，但 **sub1** 的使用却要复杂得多。**sub1** 当做参数接受的数字函数应用了一个函数 n 次，但是 **sub1** 返回的函数应该少应用该函数一次。当然任意函数的逆是不能通过反转一次应用获得的。

实现 **sub1** 的函数需要分为两个部分：

- 将给出的参数 x 和值 **true** 组对，**true** 意味着应该忽略 f 的一次应用。
- 打包给定函数 f 使其可以接受对，当且仅当对包含 **false** 时应用 f。总是返回包含 **false** 的对，使得 f 在将来会被应用。

函数 wrap 打包一个给定的函数 f:

$$wrap \doteq \lambda f.\lambda p.\langle \textbf{false},\ \textbf{if}\ (\textbf{fst}\ p)\ (\textbf{snd}\ p)\ (f\ (\textbf{snd}\ p))\rangle$$

函数 **sub1** 接受一个 n 且返回一个新的函数。新的函数接受 f 和 x，将 f 用 wrap 打包，将 x 和 **true** 组对，用 n 应用 $(wrap\ f)$ 和 $\langle \textbf{true},\ x\rangle$ 并提取结果的第二部分——也就是 f 被 x 应用了 $n-1$ 次。

$$\textbf{sub1} \doteq \lambda n.\lambda f.\lambda x.\textbf{snd}\ (n\ (wrap\ f)\ \langle \textbf{true},\ x\rangle)$$

关于编码的提示：0 的编码和 **false** 的编码是一模一样的。因此，没有程序可以区分 0 和 **false**，程序员必须特别注意一些问题，比如 **true** 和 **false** 只会出现在布尔值上下文中。这可以类比在 C 语言使用相同模式的位以实现 0、**false** 和空指针。

▷ **练习 3.6.** 证明 $\textbf{add1}\ 1 =_{\mathbf{n}} 2$。

▷ **练习 3.7.** 证明 $\textbf{iszero1}\ 1 =_{\mathbf{n}} \textbf{false}$。

▷ **练习 3.8.** 证明 $\textbf{sub1}\ 1 =_{\mathbf{n}} 0$。

▷ **练习 3.9.** 使用允许我们定义 **add** 的技术定义 **mult**。换言之，利用 n 本身应用一个函数 n 次的事实，将 (**mult** $n\ m$) 实现成 n 次的 m 加上 0。提示：什么样的值是 (**add** m)？

▷ **练习 3.10.** λ 演算没有发送错误信号的机制。当 **sub1** 被 0 应用时会怎么样？当 **iszero** 被 **true** 应用时会怎么样？

3.6 递归

上一章的练习中要求你使用实现 **add** 的方式去实现 mult。这种实现方式利用了函数编码数字的方式。

如果使用上一节的函数 **iszero**，**add** 和 **sub1**，我们也可以在不了解任何数字编码的前提下来实现 mult（也就是程序员实现函数的一般做法）。我们必须定义一个递归程序以检查第一个参数是否是 0，如果不是，则将第二个参数加到缩减第一个参数的递归调用中。

$$mult \stackrel{?}{=} \lambda n.\lambda m.\textbf{if}\ (\textbf{iszero}\ n)\ 0\ (\textbf{add}\ m\ (mult\ (\textbf{sub1}\ n)\ m))$$

上面关于 mult 宏的定义的问题是，它指向的是其自身（即它是递归的），所以没法将 mult 拓展成一个纯粹的 λ 演算表达式。因此，这个缩写是非法的。

3.6.1 通过自应用实现递归

乘法函数如何能获得它自己的句柄？在定义 mult 宏的时候，我们无法获取乘法函数的定义，但是该定义在之后即可获得。尤其是当我们调用乘法函数的时候，我们需要乘法函数的句柄。

因此，乘法函数可以要求我们提供一个乘法函数 t（其自身）以及要相乘的参数，而不是直接应用其自身。更准确地说，使用这种策略，我们定义的宏不再是乘法函数，而是一个乘法函数生成器：它接受某个函数 t 然后产生可再接受两个相乘参数的乘法函数。

$$\mathbf{mkmult}_0 \doteq \lambda t.\lambda n.\lambda m.\mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ m\ (t\ (\mathbf{sub1}\ n)\ m))$$

如果宏 $\mathbf{mkmult}_0$ 被精确定义,同时假设 t 是一个乘法函数,那么 $(\mathbf{mkmult}_0\ t)$ 则生成了一个乘法函数！显然，我们仍没有获得一个乘法函数。我们尝试将原始的 mult 定义自我参数化，但是这样做我们就失去了 mult 的定义。

我们虽然不能提供一个像 t 一样的乘法函数，但可以换而提供像 t 一样的乘法函数生成器。这是否会有用，或者最终的结果只是一个有限的回退？事实证明，将一个生成器传给另一个生成器是正确的。

假设生成器应用生成器可以产生乘法函数。因此，传给生成器的初始参数 t 应该也是生成器。在生成器的实现主体中，不论何处需要乘法函数，我们都可以用 $(t\ t)$ 取代——因为 t 是一个生成器，且生成器应用自身可以产生一个生成器。这里生成器 $\mathbf{mkmult}_1$ 期待一个生成器作为其参数：

$$\mathbf{mkmult}_1 \doteq \lambda t.\lambda n.\lambda m.\mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ m\ ((t\ t)\ (\mathbf{sub1}\ n)\ m))$$

如果 $\mathbf{mkmult}_1$ 可以生效，那么我们可以通过让 $\mathbf{mkmult}_1$ 应用自身来获得 mult 函数：

$$\mathbf{mkmult} \doteq (\mathbf{mkmult}_1\ \mathbf{mkmult}_1)$$

让我们试着将这个可疑的函数应用 0 和 m（对于某任意 m），从而确保返回的确实是 0。我们将仅在需要的时候展开缩写，且假设诸如 **iszero** 和 0 这样的缩写都可以按预期的方式执行：

$$\begin{array}{lll}
mult & 0 & m \\
= & & (\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ 0\ m \\
\rightarrow_{\mathbf{n}} & & \lambda n.\lambda m.\mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ n)\ m)) \\
\rightarrow_{\mathbf{n}} & & \lambda m.\mathbf{if}\ (\mathbf{iszero}\ 0)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ 0)\ m)) \\
\rightarrow_{\mathbf{n}} & & \mathbf{if}\ (\mathbf{iszero}\ 0)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ 0)\ m)) \\
\rightarrow\!\!\!\!\rightarrow_{\mathbf{n}} & & \mathbf{if}\ \mathbf{true}\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ 0)\ m)) \\
\rightarrow\!\!\!\!\rightarrow_{\mathbf{n}} & & 0
\end{array}$$

到目前为止一直都还不错。如果我们将某个不等于 0 的 n 和 m 相乘，会怎么样？

$$\begin{array}{lll} mult & n & m \\ = & & (\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ n\ m \\ \rightarrow_{\mathbf{n}} & & \lambda n.\lambda m.\mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ n)\ m)) \\ \rightarrow_{\mathbf{n}} & & \lambda m.\mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ n)\ m)) \\ \rightarrow_{\mathbf{n}} & & \mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ n)\ m)) \\ \twoheadrightarrow_{\mathbf{n}} & & \mathbf{if\ false}\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ n)\ m)) \\ \twoheadrightarrow_{\mathbf{n}} & & (\mathbf{add}\ m\ ((\mathbf{mkmult}_1\ \mathbf{mkmult}_1)\ (\mathbf{sub1}\ n)\ m)) \end{array}$$

因为 $mult = (\mathbf{mkmult}_1\ \mathbf{mkmult}_1)$,上面的最后一步也可以写成 $(\mathbf{add}\ m\ (mult\ (\mathbf{sub1}\ n)\ m))$。因此

$$\begin{array}{lll} mult\ 0\ m & \twoheadrightarrow_{\mathbf{n}} & 0 \\ mult\ n\ m & \twoheadrightarrow_{\mathbf{n}} & (\mathbf{add}\ m\ (mult\ (\mathbf{sub1}\ n)\ m)) \quad 当 n \neq 0 \end{array}$$

这正是我们想要的 mult，**add**，**sub1** 和 0 之间的关系。

▷ **练习 3.11.** 定义宏 **mksum** 使得 (**mksum mksum**) 的行为和“sum”函数一致，即接受一个数值 n 且将 0 到 n 所有的数相加。

3.6.2 消除自应用

上一节介绍的技术可以让我们定义任何我们想要的递归函数。但是这个方法有些笨拙，因为我们不得不定义一个生成器函数，使其为每个递归调用都应用一个原始参数。为方便起见，我们想将自应用模式拉入其自身抽象中。

更具体的说,我们需要一个函数,暂称为 mk,它可以接受任何诸如 $\mathbf{mkmult}_0$ 的生成器且返回一个已生成的函数。例如 $(mk\ \mathbf{mkmult}_0)$ 应该返回一个乘法器

$$mk \stackrel{?}{=} \lambda t.t\ (mk\ t)$$

上面关于 mk 函数的定义不够完整，但是我们可以通过这个不完整的定义获得思路。mk 函数应该接受一个生成器 t，并输出目标函数。它通过调用函数要求的生成器 $(mk\ t)$ 实现这一点——该方式应该会产生目标函数。

我们可以用上一节介绍的技术修正 mk 函数的定义循环：

$$\begin{array}{c} \mathbf{mkmk} \doteq \lambda k.\lambda t.t\ ((k\ k)\ t) \\ mk \doteq (\mathbf{mkmk}\ \mathbf{mkmk}) \end{array}$$

我们可以验证 $(mk\ \mathbf{mkmult}_0)$ 和 mult 行为一一致：

$$
\begin{array}{ll}
& (mk\ \mathbf{mkmult}_0)\ 0\ m \\
= & ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ 0\ m \\
= & (((\lambda k.\lambda t.t\ ((k\ k)\ t))\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ 0\ m \\
\to_{\mathbf{n}} & ((\lambda t.t\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ t))\ \mathbf{mkmult}_0)\ 0\ m \\
\to_{\mathbf{n}} & (\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ 0\ m \\
\to_{\mathbf{n}} & (\lambda n.\lambda m.\mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ (\mathbf{sub1}\ n)\ m)))\ 0\ m \\
\to_{\mathbf{n}} & (\lambda m.\mathbf{if}\ (\mathbf{iszero}\ 0)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ (\mathbf{sub1}\ 0)\ m)))\ m \\
\to_{\mathbf{n}} & \mathbf{if}\ (\mathbf{iszero}\ 0)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ (\mathbf{sub1}\ 0)\ m)) \\
\to_{\mathbf{n}} & 0
\end{array}
$$

$$
\begin{array}{ll}
& (mk\ \mathbf{mkmult}_0)\ 0\ m \\
= & ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ 0\ m \\
= & (((\lambda k.\lambda t.t\ ((k\ k)\ t))\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ 0\ m \\
\to_{\mathbf{n}} & ((\lambda t.t\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ t))\ \mathbf{mkmult}_0)\ 0\ m \\
\to_{\mathbf{n}} & (\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ 0\ m \\
\to_{\mathbf{n}} & (\lambda n.\lambda m.\mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ (\mathbf{sub1}\ n)\ m)))\ 0\ m \\
\to_{\mathbf{n}} & (\lambda m.\mathbf{if}\ (\mathbf{iszero}\ n)\ n\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ (\mathbf{sub1}\ n)\ m)))\ m \\
\to_{\mathbf{n}} & \mathbf{if}\ (\mathbf{iszero}\ n)\ n\ (\mathbf{add}\ m\ ((\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ (\mathbf{sub1}\ n)\ m)) \\
\twoheadrightarrow_{\mathbf{n}} & (\mathbf{add}\ m\ ((\mathbf{mkmult}_0\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ \mathbf{mkmult}_0)\ (\mathbf{sub1}\ n)\ m)) \\
= & (\mathbf{add}\ m\ ((\mathbf{mkmult}_0\ (mk\ \mathbf{mkmult}_0)\ (\mathbf{sub1}\ n)\ m))
\end{array}
$$

3.6.3 不动点和 Y 组合子

从这个角度看，mk 函数无论如何都有些神秘。但不管怎么样，即使在 $\mathbf{mkmult}_0$ 被给予想要生成的乘法器时仅能生成该乘法器，它也能使 $\mathbf{mkmult}_0$ 变得实用！

一般来说，$\mathbf{mkmult}_0$ 可能被给予任何函数，而其最终返回的“乘法器”也会根据输入函数的不同而有相应不同的行为。但 mk 却为 $\mathbf{mkmult}_0$ 找到了它的输入函数，使得其输出函数和输入函数相同。换句话说，mk 找到了 $\mathbf{mkmult}_0$ 的**不动点**。

事实证明，mk 可以找到任何函数的不动点。换句话说，如果使用 mk 应用 M 会输出 N，那么用 M 应用 N 仅仅只会再次输出 N。

定理 3.1 对任何 M 都有 $M\ (mk\ M) =_{\mathbf{n}} (mk\ M)$

定理 3.1 的证明：因为 $=_{\mathbf{n}}$ 是 $\to_{\mathbf{n}}$ 的对称闭包，所以我们可以通过证明 $mk\ M \twoheadrightarrow_{\mathbf{n}} M\ (mk\ M)$ 来证明上述定理：

$$
\begin{array}{rcl}
mk\ M & = & (\mathbf{mkmk}\ \mathbf{mkmk})\ M \\
 & = & ((\lambda k.\lambda t.t\ ((k\ k)\ t))\ \mathbf{mkmk})\ M \\
 & \to_{\mathbf{n}} & (\lambda t.t\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ t))\ M \\
 & \to_{\mathbf{n}} & M\ ((\mathbf{mkmk}\ \mathbf{mkmk})\ M) \\
 & = & M\ (mk\ M)
\end{array}
$$

像 mk 这样的函数称为**不动点操作符**。mk 函数仅仅是 λ 演算中诸多不动点操作符中的一种。其中更广为人知的是 Y[1]：

$$Y \doteq \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

一般来说，Y 允许我们使用比 3.6.1 节介绍的手动技术更简单的方法定义递归函数。例如，我们可以这样定义 sum：

$$sum \doteq Y\ (\lambda s.\lambda n.\mathbf{if}\ (\mathbf{iszero}\ n)\ 0\ (\mathbf{add}\ n\ (s\ (\mathbf{sub1}\ n))))$$

既然我们不必去重复一个大型生成器表达式，我们就可以跳过用于过渡的生成器缩写 mksum，而是直接使用 Y 应用一个生成器函数。

此外，一个程序员看到了上述关于 sum 的定义会立即注意到其中的 Y，会知道 s 是 Y 的参数的参数，然后会读取 $\lambda n \ldots$ 当作名为 s 的递归函数。

▷ **练习 3.12.** 证明对任何 M 都有 $M\ (Y\ M) =_{\mathbf{n}} (Y\ M)$。

▷ **练习 3.13.** 使用下列宏定义 Lisp cons cells 的编码：

- **null**，一个常量
- **cons**，接受两个参数并返回一个 cons cells 的函数
- **isnull**，如果它的参数是 **null** 则返回 **true**，如果参数是一个 cons cell 则返回 **false**
- **car**，一个接受一个 cons cell 并返回其第一个元素的函数
- **crd**，一个接受一个 cons cell 并返回其第二个元素的函数

需要注意的是，你的编码必须具体满足下列等式：

$$\begin{array}{lll}(\mathbf{isnull}\ \mathbf{null}) & =_{\mathbf{n}} & \mathbf{true} \\ (\mathbf{isnull}\ (\mathbf{cons}\ M\ N)) & =_{\mathbf{n}} & \mathbf{false} \\ (\mathbf{car}\ (\mathbf{cons}\ M\ N)) & =_{\mathbf{n}} & M \\ (\mathbf{cdr}\ (\mathbf{cons}\ M\ N)) & =_{\mathbf{n}} & N\end{array}$$

你的编码不必为诸如 (**car null**) 或 (**car cons**) 这样的表达式赋予任何特殊意义。

▷ **练习 3.14.** 使用你在上一个练习中的编码定义 **length**，该函数接受一个布尔值列表且返回列表中 cons cell 的个数。布尔值列表中要么是 **null**，要么是 (**cons** $b\ l$)，其中 b 是 **true** 或 **false** 而 l 是一个布尔值列表。

[1] 术语 **Y 组合子**的概念实际上指代整个不动点算子族，稍后会详述

3.7 归约策略和一般式

表达式何时可以算是完全归约？几乎任何表达式都可以被 $\rightarrow_{\mathbf{n}}^{\alpha}$ 归约，只需重命名变量即可，所以它不能列入该定义中。也就是说，完全归约表达式是不能被 $\rightarrow_{\mathbf{n}}^{\beta}$ 和 $\rightarrow_{\mathbf{n}}^{\eta}$ 归约的表达式。

如果一个表达式不能被 $\rightarrow_{\mathbf{n}}^{\beta}$ 或 $\rightarrow_{\mathbf{n}}^{\eta}$ 归约，则是**一般式**。
如果 $M =_{\mathbf{n}} N$ 且 N 是一般式，则说 M **存在一般式** N。

一般式类似 λ 演算表达式的结果。如果表达式存在一般式，那么它仅有一个一般式（可能通过不同的途径归约）。更详细的说，这是一个以 $\rightarrow_{\mathbf{n}}^{\alpha}$ 为模的一般式。

定理 3.2 (一般式) 如果 $L =_{n} M$ 且 $L =_{N}$，且 M 和 N 都是一般式，那么 $M =_{\alpha} N$。

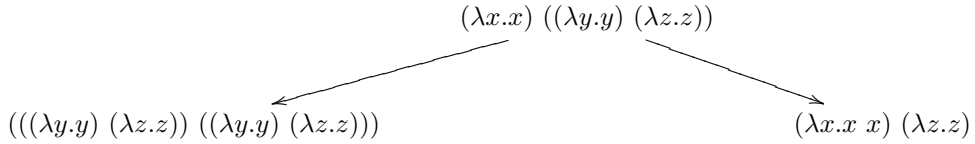
通常，$=_{\alpha}$ 是 α 合拍闭包产生的等价类。如果 λ 演算具有丘奇罗瑟性质，定理 3.2 是很容易证明的：

定理 3.3 ($=_{\mathbf{n}}$ 的丘奇罗瑟定理) 如果 $M =_{n} N$，那么存在 L' 使 $M \twoheadrightarrow_{n} L'$ 且 $N \twoheadrightarrow_{n} L'$ 。

从 $=_{\mathbf{r}}$ 的角度看，这个定理的证明依赖于 $\twoheadrightarrow_{\mathbf{n}}$ 的菱形定理

定理 3.4 ($\twoheadrightarrow_{\mathbf{n}}$ 的菱形定理) 如果 $L \twoheadrightarrow_{n} M$ 且 $L \twoheadrightarrow =_{N}$，那么存在 L' 使 $M \twoheadrightarrow_{n} L'$ 且 $N \twoheadrightarrow_{n} L'$ 。

单步关系 $\rightarrow_{\mathbf{n}}$ 并不遵循菱形定理，即使是 $\rightarrow_{\mathbf{r}}$ 这样略微改变过的类菱形定理也不遵守。究其原因是因为 $\rightarrow_{\mathbf{n}}^{\beta}$ 能够复制归约表达式。例如：



不存在某个可以将两个位于底层的表达式同时回退的单步。下一章我们会学到，解决该问题的关键在于定义并行归约概念上的独立子表达式，这使得左侧的两

个子表达式 $((\lambda y.y)\ (\lambda z.z))$ 都可以被同时一次性归约。但是，我们目前不想证明定理 3.4，而是在下一章证明相关语言的菱形定理。

和 B 语言表达式中每个都有固定的结果 f 或 t 不同的是，并不是每个 λ 表达式都有一般式。例如，Ω 没有一般式：

$$\Omega \doteq ((\lambda x.x\ x)\ (\lambda x.x\ x))$$

即使一个表达式存在一般式，也可能存在让这个表达式永远无法变成一般式的无限归约序列。例如：

$$\begin{array}{ll}
(\lambda y.\lambda z.z)\ ((\lambda x.x\ x)\ (\lambda w.w\ w)) & \\
\to_{\mathbf{n}} \quad \lambda z.z & \text{一般式}
\end{array}$$

$$\begin{array}{ll}
(\lambda y.\lambda z.z)\ ((\lambda x.x\ x)\ (\lambda w.w\ w)) & \\
\to_{\mathbf{n}} \quad (\lambda y.\lambda z.z)\ ((\lambda w.w\ w)\ (\lambda w.w\ w)) & \\
\to_{\mathbf{n}} \quad (\lambda y.\lambda z.z)\ ((\lambda w.w\ w)\ (\lambda w.w\ w)) & \\
\to_{\mathbf{n}} \quad \cdots & \text{永远是相同的表达式}
\end{array}$$

因此，虽然定理 3.2 保证最多只存在一个一般式，但是我们目前没有还没有办法找到它。直观上来看，上述无限递归的问题在于，我们在对并没有被函数体用到的参数求值。这暗示我们应寻找一个极左 β，η 归约的策略以获得一般式:

$$\begin{array}{lll}
M \to_{\overline{\mathbf{n}}} & \text{当} & M\ \beta\ N \\
M \to_{\overline{\mathbf{n}}} & \text{当} & M\ \eta\ N \\
(\lambda X.M) \to_{\overline{\mathbf{n}}} (\lambda X.N) & \text{当} & M \to_{\overline{n}} N \\
(M\ N) \to_{\overline{\mathbf{n}}} (M'\ N) & \text{当} & M \to_{\overline{n}} M' \\
 & & \forall L,\ (M\ N)\ \not\beta\ L\ [\text{且}(M\ N)\ \not\eta\ L] \\
(M\ N) \to_{\overline{\mathbf{n}}} (M\ N') & \text{当} & N \to_{\overline{n}} N' \\
 & & M\text{是一般式} \\
 & & \forall L,\ (M\ N)\ \not\beta\ L\ [\text{且}(M\ N)\ \not\eta\ L]
\end{array}$$

如果存在一般式，$\to_{\overline{\mathbf{n}}}$ 关系保证一定能找到。

定理 3.5 (正则序归约) 当且仅当 $M \to_{\overline{\mathbf{n}}} N$ 时，M 存在一般式 N。

虽然只要存在一般式，使用正则序归约一定可以找到，但实际上没有编程语言会采用这种归约方式。原因在于该策略虽然很强大，但通常很低效。例如，在之前展示的 $\to_{\mathbf{n}}$ 的非菱形性质图表中，左侧归约对应正则序，而右侧的归约用更少的步骤生成了恒等函数。另一个问题是 $\to_{\overline{\mathbf{n}}}$ 会在函数被应用之前对函数内部进行求值。

在任何情况下，既然我们有了一个唯一一般式的概念，自然而然要问的问题就是我们是否可以用定义 $eval_{\mathbf{r}}$ 的方式定义 $eval_{\mathbf{n}}$：

$$eval_{\mathbf{n}}(M) \stackrel{?}{=} N \text{ 如果} M =_{\mathbf{n}} N \text{且 N 是一般式}$$

从 α 重命名的角度说，上述定义是不完整的，但这是一个更深的问题。正如我们所见，存在 mk 和 Y 这样函数，它们行为一致，但是不可以互相归约。下一章中，我们会探讨解决这个问题的方式。

▷ **练习 3.15.** 证明 $((\lambda x.x\ x)\ (\lambda x.x\ x))$ 没有一般式。

3.8 历史

丘奇提出 λ 演算的时间比图灵发明图灵机的时间稍微早一点。

Barendregt[2] 将丘奇的 λ 演算作为一个逻辑系统进行了综合研究，许多描述术语的传统都始于 Barendregt。他的书提供了大量演算相关的技术，虽然该书没有将 λ 演算作为程序设计语言的一种算子。

第 II 部分　真实语言的模型

第 4 章

ISWIM

丘奇用一种全数学的方式研究 λ 演算，在具体方式上和机械计算背道而驰。在 20 世纪 60 年代 Landin 证明了丘奇的演算实际上并不是大部分编程语言的恰当的模型。例如，λ 演算表达式

$$(\lambda x.1)\ (\mathbf{sub1}\ \lambda y.y)$$

会归约到 1（的编码），即使大多数编程语言都会对 (**sub1** $\lambda y.y$) 报错。问题不仅在 **sub1** 和 $\lambda y.y$ 的编码上，而是整个表达式的 β 归约规则会完全忽略参数 (**sub1** $\lambda y.y$)。

这种按名调用的行为是否可取是一个有待讨论的话题。不管怎样，很多语言是不支持按名调用的。反之，它们支持按值调用，也就是在函数被应用之前，该函数的参数必须被完全求值。

在本章，我们介绍 Landin 的 **ISWIM**。它更进一步的模型化诸如 Scheme 和 ML 等按值调用语言的本质。ISWIM 的基本语义和 λ 演算相同，且自由变量和替换的概念也相同。ISWIM 处理基本常量和基本操作符的集合，这使它 λ 演算不同而更接近真实编程语言。但是更本质的不同在于 ISWIM 的按值调用归约规则。

4.1 ISWIM 表达式

ISWIM 的文法拓展了 λ 演算文法：

$$
\begin{array}{lcl}
M, N, L, K & = & X \\
 & | & (\lambda X.M) \\
 & | & (M\ M) \\
 & | & b \\
 & | & (o^n\ M\ \ldots\ M) \\
X & = & \text{一个变量：} x,\ y, \ldots \\
b & = & \text{一个基本常量} \\
o^n & = & \text{一个 n 元基本操作符}
\end{array}
$$

其中仅当 $n = m$ 时，表达式 $(o^n\ M_1\ \ldots\ M_{\mathbf{n}})$ 合法。我们可以用多种不同的方式定义集合 b 和 o^n，语言的高级特性仍然保持不变。具体的说，我们用如下方式定义 b 和 o^n：

$$
\begin{array}{rcl}
b & = & \{\ulcorner n \urcorner | n \in \mathbb{Z}\} \\
o^1 & = & \{\mathbf{add1}, \mathbf{sub1}, \mathbf{iszero}\} \\
o^2 & = & \{+, -, *, \uparrow\}
\end{array}
$$

词法对象 $\ulcorner 1 \urcorner$ 代表整数 1。词法对象 + 代表加法操作符，↑ 是幂操作符等等。另外，我们定义一个实用的宏 **if0**：

$$
(\mathbf{if0}\ K\ M\ N) \doteq (((\mathbf{iszero}\ K)\ (\lambda X.M)\ (\lambda X.N))\ \ulcorner 0 \urcorner)
$$
其中$X \notin \mathcal{FV}(M) \bigcup \mathcal{FV}(N)$

关系 $\mathcal{FV}$ 和 $\_[\_ \leftarrow \_]$ 可以用显而易见的方式拓展成新的文法：

$$
\begin{array}{lcl}
\mathcal{FV}(\ b\ ) & = & \{\ \} \\
\mathcal{FV}(\ X\ ) & = & \{X\} \\
\mathcal{FV}(\ (\lambda X.M)\ ) & = & \mathcal{FV}(M) \setminus \{X\} \\
\mathcal{FV}(\ (M_1\ M_2)\ ) & = & \mathcal{FV}(M_1) \bigcup \mathcal{FV}(M_2) \\
\mathcal{FV}(\ (o^n\ M_1\ \ldots M_{\mathbf{n}})\ ) & = & \mathcal{FV}(M_1) \bigcup \ldots \mathcal{FV}(M_{\mathbf{n}}) \\
b[X \leftarrow M] & = & b \\
X_1[X_1 \leftarrow M] & = & M \\
X_2[X_2 \leftarrow M] & = & X_2 \quad \text{其中} X_1 \neq X_2 \\
(\lambda X_1.M_1)[X_1 \leftarrow M_2] & = & (\lambda X_1.M_1) \\
(\lambda X_1.M_1)[X_2 \leftarrow M_2] & = & (\lambda X_3.M_1[X_1 \leftarrow X_3][X_2 \leftarrow M_2]) \\
 & & \text{其中} X_1 \neq X_2, X_3 \notin \mathcal{FV}(M_2) \\
 & & \text{且} X_3 \notin \mathcal{FV}(M_1) \setminus \{X\} \\
(M_1\ M_2)[X \leftarrow M_3] & = & (M_1[X \leftarrow M_3]\ M_2[X \leftarrow M_3]) \\
(o^n\ M_1\ \ldots M_{\mathbf{n}})[X \leftarrow M] & = & (o^n\ M_1[X \leftarrow M]\ \ldots\ M_{\mathbf{n}}[X \leftarrow M])
\end{array}
$$

4.2 ISWIM 归约

因为 ISWIM 中的函数仅接受被完全求值的参数，我们必须先定义**值**的集合。值的集合 V（也称 U 或 W）是表达式集合的一个子集：

$$\begin{array}{rcl} U, V, W & = & b \\ & | & X \\ & | & (\lambda X.M) \end{array}$$

需要特别指出的是，一个应用表达式永远不是一个值，然而一个抽象（比如函数）却总是一个值，无论它主体的形态是怎样的。

ISWIM 的核心归约关系是 $\beta_{\mathbf{v}}$，$\beta_{\mathbf{v}}$ 除了参数必须属于 V 而非 M，其他的都类似 β：

$$((\lambda X.M)\ V) \quad \beta_{\mathbf{v}} \quad M[X \leftarrow V]$$

在某种程度上，限制参数必须是 V 的元素也强制了求值的顺序。例如 $((\lambda x.1)\ ((sub1)\ \lambda y.y))$ 不可能通过 $\beta_{\mathbf{v}}$ 归约到 1，因为 $((sub1)\ \lambda y.y)$ 不是 V 的元素。类似的，$((\lambda x.1)\ ((sub1)\ 1))$ 按顺序必须先归约到 $((\lambda x.1)\ 0)$ 然后才是 1。

ISWIM 没有类似 η 的关系，这意味着真实的编程语言的实现通常不会提取某个函数在另一个函数体中的应用。此外，从现在开始，我们通常用 α 等价（$=_\alpha$）来比较表达式，而不是把 α 视为归约。

除了函数应用，ISWIM 归约规则还需考虑基本运算符。同样地，b 和 o^n 本质上也是抽象的（即使我们在例子中定义了具体的集合），和基本运算符相关的归约记为抽象 σ。关系 σ 将每个 o^n 加上 n 个基本常量映射成一个值。具体地说，我们如下选择 σ：

$$\begin{array}{lll} (\mathbf{add1}\ \ulcorner m \urcorner) & \mathbf{b}^1 & \ulcorner m+1 \urcorner \\ (\mathbf{sub1}\ \ulcorner m \urcorner) & \mathbf{b}^1 & \ulcorner m-1 \urcorner \\ (\mathbf{iszero}\ \ulcorner 0 \urcorner) & \mathbf{b}^1 & \lambda xy.x \\ (\mathbf{iszero}\ \ulcorner n \urcorner) & \mathbf{b}^1 & \lambda xy.y \quad n \neq 0 \\ (+\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m+n \urcorner \\ (-\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m-n \urcorner \\ (\cdot\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m \cdot n \urcorner \\ (\uparrow\ \ulcorner m \urcorner\ \ulcorner n \urcorner) & \mathbf{b}^2 & \ulcorner m^n \urcorner \end{array}$$

$$\sigma \quad = \quad \mathbf{b}^1 \cup \mathbf{b}^2$$

通过合并 β 和 σ，我们得到了完整的关系 $\mathbf{v}$：

$$\mathbf{v} = \beta_{\mathbf{v}} \cup \sigma$$

通常，$\rightarrow_{\mathbf{v}}$ 是 $\mathbf{v}$ 的合拍闭包，$\twoheadrightarrow_{\mathbf{v}}$ 是 $\rightarrow_{\mathbf{v}}$ 的传递自反闭包，而 $=_{\mathbf{v}}$ 是 $\twoheadrightarrow_{\mathbf{v}}$ 的对称闭包。

▷ **练习 4.1.** 使用 $\rightarrow_{\mathbf{v}}$ 将表达式

$$(\lambda w.(- \ (w \ \ulcorner 1 \urcorner) \ \ulcorner 5 \urcorner)) \ ((\lambda x.x \ \ulcorner 10 \urcorner) \ \lambda yz.(+ \ z \ y))$$

归约成一个值。

4.3 $Y_{\mathbf{v}}$ 组合子

对纯粹的 λ 演算，我们定义函数 Y 以寻找任意表达式的不动点，因此它可用于定义任何递归函数。虽然在 ISWIM 中，对于任何 f，$(f \ (Y \ f))$ 和 $(Y \ f)$ 相等仍然成立，但事实上这是无用的：

$$\begin{array}{rcl} Y \ f & = & (\lambda f.(\lambda x.f \ (x \ x)) \ (\lambda x.f \ (x \ x))) \ f \\ & \rightarrow_{\mathbf{v}} & (\lambda x.f \ (x \ x)) \ (\lambda x.f \ (x \ x)) \\ & \rightarrow_{\mathbf{v}} & f \ ((\lambda x.f \ (x \ x)) \ (\lambda x.f \ (x \ x))) \end{array}$$

问题是上面 f 最外层应用的参数并不是一个值——也不能归约到一个值——所以 $(Y \ f)$ 永远不能生成期待的不动点函数。

我们可以通过将 Y 中的每个应用 M 变成 $(\lambda X.M \ X)$ 以避免无限归约，因为无论如何，该应用应该返回一个函数。这个逆 η 变换将无限归约应用替换成一个值。最终得到的将是 $Y_{\mathbf{v}}$ 组合子：

$$\begin{array}{rcl} Y_{\mathbf{v}} & = & (\lambda f. \\ & & \quad (\lambda x. \\ & & \quad \ (\ (\lambda g.(f \ (\lambda x.((g \ g) \ x)))) \\ & & \quad \ \ (\lambda g.(f \ (\lambda x.((g \ g) \ x)))) \ x))) \end{array}$$

$Y_{\mathbf{v}}$ 组合子被一个接受函数且返回另一个函数的函数应用时生效。

定理 4.1 ($Y_{\mathbf{v}}$ 的不动点原理) 如果 $K = \lambda gx.L$,那么 $(K\ (Y_{\mathbf{v}}K)) =_{\mathbf{v}} (Y_{\mathbf{v}}\ K)$

定理 4.1 的证明：证明的方式是直接计算，其中所有的证明步骤都是 $\beta_{\mathbf{v}}$ 步骤：

$$
\begin{array}{ll}
& (Y_{\mathbf{v}}\ K) \\
= & ((\lambda f.\lambda x.((\lambda g.(f\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(f\ (\lambda x.((g\ g)\ x))))\ x))\ K) \\
\to_{\mathbf{v}} & \lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x) \\
\to_{\mathbf{v}} & \lambda x.((K\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\
\to_{\mathbf{v}} & \lambda x.((\lambda gx.L)\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\
\to_{\mathbf{v}} & \lambda x.((\lambda x.L[g \leftarrow (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x))])\ x) \\
\to_{\mathbf{v}} & \lambda x.L[g \leftarrow (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))] \\
\leftarrow_{\mathbf{v}} & ((\lambda gx.L)\ (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))) \\
= & (K\ (\lambda x.((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ x))) \\
\leftarrow_{\mathbf{v}} & (K\ (Y_{\mathbf{v}}\ K))
\end{array}
$$

定理 4.1 的证明看起来过于复杂，因为步骤中的参数必须是让 $\beta_{\mathbf{v}}$ 应用的值。因此我们需要传递由 $(Y_{\mathbf{v}}\ K)$ 计算出的值，而不是计算

$$
\begin{array}{ll}
& \lambda x.((K\ (\lambda x.(((\lambda g.(K\ (\lambda x.((g\ g)\ x))))\ (\lambda g.(K\ (\lambda x.((g\ g)\ x)))))\ x)))\ x) \\
= & \lambda x.((K\ (Y_{\mathbf{v}}\ K))\ x) \\
= & \lambda x.(((\lambda gx.L)\ (Y_{\mathbf{v}}\ K))\ x)
\end{array}
$$

为了接下来避免这样的复杂计算，我们证明对于可以证实是一个值的参数（但是无需已经成为一个值）已经可以被应用，就像它是 $=_{\mathbf{v}}$ 的目标值一。

定理 4.2 如果 $M =_{\mathbf{v}} V$ 那么对所有的 $(\lambda X.N)$，都有 $((\lambda X.N)\ M) =_{\mathbf{v}} N[X \leftarrow M]$。

定理 4.2 的证明：证明的初始部分是一个简单计算：

$$((\lambda X.N)\ M)\ =_{\mathbf{v}}\ ((\lambda X.N)\ V)\ =_{\mathbf{v}}\ N[X \leftarrow V] =_{\mathbf{v}}\ N[X \leftarrow M]$$

然而这个计算中的最后一步，需要根据 N 的归纳分解证明，证明如果 $M =_{\mathbf{v}} L$，$N[X \leftarrow M]\ =_{\mathbf{v}}\ N[X \leftarrow L]$：

- 奠基步骤：
  - **当** $N = b$ **时**
    $b[X \leftarrow M] = b =_{\mathbf{v}} b = b[X \leftarrow L]$。
  - **当** $N = Y$ **时**
    如果 $Y = X$，那么 $X[X \leftarrow M] = M =_{\mathbf{v}} L = X[X \leftarrow L]$。否则，$Y[X \leftarrow M] = Y =_{\mathbf{v}} Y = Y[X \leftarrow L]$。
- 归纳步骤：
  - **当** $N = \lambda Y.N'$ **时**
    如果 $Y = X$，那么 $N[M \leftarrow X] = N =_{\mathbf{v}} N = N[L \leftarrow X]$。否

则，根据归纳，$N'[X \leftarrow M] =_{\mathbf{v}} N'[X \leftarrow L]$。那么

$$\begin{array}{ll} & (\lambda Y.N')[X \leftarrow M] \\ = & (\lambda Y.N'[X \leftarrow M]) \\ =_{\mathbf{v}} & (\lambda Y.N'[X \leftarrow L]) \\ = & (\lambda Y.N')[X \leftarrow L] \end{array}$$

- 当 $N = (N_1\ N_2)$ 时
  根据归纳，对 $i \in [1,2]$ 有 $N_i[X \leftarrow M] =_{\mathbf{v}} N_i[X \leftarrow L]$，则

$$\begin{array}{ll} & (N_1\ N_2)[X \leftarrow M] \\ = & (N_1[X \leftarrow M]\ N_2[X \leftarrow M]) \\ =_{\mathbf{v}} & (N_1[X \leftarrow L]\ N_2[X \leftarrow L]) \\ = & (N_1\ N_2)[X \leftarrow L] \end{array}$$

- 当 $N = (o^n\ N_1 \ldots\ N_{\mathbf{n}})$ 时
  和上一种情况相同。

4.4 求值

从定义求值器的角度说，作为值的函数在 ISWIM 中产生了和 λ 演算中同样多的问题。例如，表达式

$$(\lambda x.x)\ (\lambda y(\lambda x.x)\ \ulcorner 0 \urcorner)$$

显然等价于

$$\lambda y.(\lambda x.x)\ \ulcorner 0 \urcorner$$

以及

$$\lambda y.\ulcorner 0 \urcorner$$

我们应该选择哪一个作为求值器的结果呢？虽然这些结果很可能都在表达同一个函数，但也显然存在无限多种方式可以表达同一个函数。因此我们采用一个本质上所有实际编程系统实现的解决方案：当一个程序归约到一个函数值的时候，$eval_{\mathbf{v}}$ 函数仅仅返回一个表示该值为函数的词素。

我们定义 ISWIM 函数求值的结果集 A：

$$A = b \cup \{\mathbf{function}\}$$

这个部分函数 $eval_{\mathbf{v}}$ 的定义如下：

$$eval_{\mathbf{v}}(M) = \begin{cases} b & \text{当 } M =_{\mathbf{v}} b \text{ 时} \\ \mathbf{function} & \text{当 } M =_{\mathbf{v}} \lambda X.N \text{ 时} \end{cases}$$

如果 $eval_{\mathbf{v}}(M)$ 不存在，我们则说 M 是**发散**的。例如，Ω 是发散的。

▷ **练习 4.2.** 假设我们尝试将求值函数强化成如下形式：

$$eval_1(M) = \begin{cases} b & \text{当 } M =_{\mathbf{v}} b \text{ 时} \\ \mathbf{function1} & \text{当 } M =_{\mathbf{v}} \lambda X.N \text{ 时} \\ \mathbf{function+} & \text{当 } M =_{\mathbf{v}} \lambda X.\lambda Y.N \text{ 时} \end{cases}$$

$eval_1$ 还是一个函数吗？如果是，证明该结论。如果不是，举出一个反例。

4.5 一致性

ISWIM 求值器的定义依赖于 ISWIM 的等价演算。虽然该演算是基于直观的参数，并且几乎可以像代数系统一样易于使用，但求值器是否是对一个程序总是返回唯一结果的函数的结论还不明显。但是对于要求确定和可靠的编程语言的程序员而言，这个结论是重要的，而且是需要被形式化确立的。简单地说，我们需要为 ISWIM 的 $eval_{\mathbf{v}}$ 修改 $eval_{\mathbf{r}}$ 的一致性定理。事实上，虽然它们都采用了同样的思路，但是这个定理的证明远比 $eval_{\mathbf{r}}$ 的证明要复杂得多。

我们开始证明断定 IWSIM 演算的丘奇罗瑟性质的基本定理。

定理 4.3 ($eval_{\mathbf{v}}$ 的一致性) 关系 $eval_{\mathbf{v}}$ 是部分函数。

定理 4.3 的证明：假设丘奇罗瑟性质成立：如果 $M =_{\mathbf{v}} N$，那么存在表达式 L 使 $M \twoheadrightarrow_{\mathbf{v}} L$ 且 $N \twoheadrightarrow_{\mathbf{v}} L$。
对结果 A_1 和 A_2，令 $eval_{\mathbf{v}}(M) = A_1$ 且 $eval_{\mathbf{v}}(M) = A_2$。我们需要证明 $A_1 = A_2$。基于对 ISWIM 结果的定义，我们将其分为两种情况：

- **当 $A_1 \in b$ 且 $A_2 \in b$ 时**
  根据丘奇罗瑟性质可得，两个基本常量可被证明是等价的当且仅当它们是恒等的，因为二者都不会归约。
- **当 $A_1 = \mathbf{function}$ 且 $A_2 \in b$ 时**
  根据 $eval_{\mathbf{v}}$ 的定义，$M =_{\mathbf{v}} A_2$ 并且对某些 N 有 $M =_{\mathbf{v}} \lambda x.N$，所以 $A_2 =_{\mathbf{v}} \lambda x.N$。再根据丘奇罗瑟性质和常量的不可归约性可得 $\lambda x.N \twoheadrightarrow_{\mathbf{v}} A_2$。但是根据归约关系 $\twoheadrightarrow_{\mathbf{v}}$ 的定义有，$\lambda x.N \twoheadrightarrow_{\mathbf{v}} K$ 可推知 $K = \lambda x.K'$。因此 $\lambda x.N$ 不可能归约到 A_2，这意味着这种情况的假设是矛盾的。
- **当 $A_1 \in b$ 且 $A_2 = \mathbf{function}$ 时**
  和上面的情况相似。
- **当 $A_1 = \mathbf{function}$ 且 $A_2 = \mathbf{function}$ 时**
  则有 $A_1 = A_2$。

这就是所有可能的情况，因此 $A_1 = A_2$ 总是为真，所以 $eval_{\mathbf{v}}$ 是一个函数。

根据上述证明，我们已经将一致性性质归约到 ISWIM 的丘奇罗瑟性质上。

定理 4.4 (ISWIM 的丘奇罗瑟性质) 如果 $M =_{\mathbf{v}} N$，则存在表达式 L 使 $M \twoheadrightarrow_{\mathbf{v}} L$ 且 $M \twoheadrightarrow_{\mathbf{v}} L$。

定理 4.4 的证明： 这个证明实际上就是 $=_{\mathbf{r}}$（定理 2.2）的证明的翻版。它假设了归约关系 $\twoheadrightarrow_{\mathbf{v}}$ 的菱形性质。

在将一致性和丘奇罗瑟定理分解以后，我们已经到了问题的核心部分。现在我们需要证明 ISWIM 的菱形性质。

定理 4.5 ($\twoheadrightarrow_{\mathbf{v}}$ 的菱形性质) 如果 $L \twoheadrightarrow_{\mathbf{v}} M$ 且 $L \twoheadrightarrow_{\mathbf{v}} N$，那么存在表达式 K 使得 $M \twoheadrightarrow_{\mathbf{v}} K$ 且 $M \twoheadrightarrow_{\mathbf{v}} K$。

对于 $\twoheadrightarrow_{\mathbf{r}}$，类菱形性质对其单步关系 $\rightarrow_{\mathbf{r}}$ 成立，通过这个结论易得其传递闭包的菱形性质亦成立。但是，和上一章的例子类似，类菱形性质显然不适用于 $\rightarrow_{\mathbf{v}}$。$\beta_{\mathbf{v}}$ 归约可以复制函数参数中的可约项，以阻止在特定的菱形中出现相同的缩略子[1]。以下面的表达式为例

$$\underline{((\lambda x.(x\ x))\ (\lambda y.\underline{((\lambda x.x)\ (\lambda x.x))}))}$$

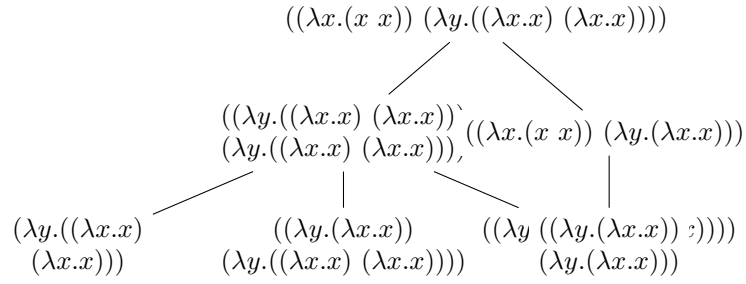
该表达式包含两个重叠的带下划线的 $\beta_{\mathbf{v}}$ 可约项。根据其中一种或另一种规则，我们可以分别得到表达式

$$\underline{((\lambda x.(x\ x))\ (\lambda y.(\lambda x.x)))}$$

以及

$$((\lambda y.\underline{((\lambda x.x)\ (\lambda x.x))})\ (\lambda y.\underline{((\lambda x.x)\ (\lambda x.x))}))$$

这两个表达式都含有可约项，但是按下图所示，始于这些表达式的一步归约并不能得到共同的表达式。



因此，一步关系 $\rightarrow_{\mathbf{v}}$ 不满足菱形性质

[1]译者注：原文为 contractum，拉丁文，原文中的意思是“经 β 归约缩略后的元素”。

因为基于 $\mathbf{v}$ 的一步关系的问题是由复制可约项的归约引起的，因此我们考虑一步关系 $\rightarrow_{\mathbf{v}}$ 的拓展 $\hookrightarrow_{\mathbf{v}}$，它可以并行收缩好几个不重叠的可约项。如果该拓展满足菱形性质，且它的自反传递闭包和 $\rightarrow_{\mathbf{v}}$ 的相同，那么我们可以证明 $\twoheadrightarrow_{\mathbf{v}}$ 归约的菱形性质。

并行拓展 $\hookrightarrow_{\mathbf{v}}$ 的定义如下：

M	$\hookrightarrow_{\mathbf{v}}$	N	如果$M =_\alpha N$
$(o^n\ b_1\ \ldots b_{\mathbf{n}})$	$\hookrightarrow_{\mathbf{v}}$	$\sigma(o^n,\ b_1,\ldots b_{\mathbf{n}})$	如果定义了 $\sigma(o^n,\ b_1,\ldots b_{\mathbf{n}})$
$((\lambda X.M)\ N)$	$\hookrightarrow_{\mathbf{v}}$	$M'[X \leftarrow V]$	如果 $M \hookrightarrow_{\mathbf{v}} M'$ 且 $N \hookrightarrow_{\mathbf{v}} V$
$(M\ N)$	$\hookrightarrow_{\mathbf{v}}$	$(M'\ N')$	如果 $M \hookrightarrow_{\mathbf{v}} M'$ 且 $N \hookrightarrow_{\mathbf{v}} N'$
$(\lambda X.M)$	$\hookrightarrow_{\mathbf{v}}$	$(\lambda x.M')$	如果 $M \hookrightarrow_{\mathbf{v}} M'$
$(o^n\ M_1\ \ldots M_{\mathbf{n}})$	$\hookrightarrow_{\mathbf{v}}$	$(o^n\ M'_1\ \ldots M'_{\mathbf{n}})$	如果对 $i \in [1,n]$ 有 $M_i \hookrightarrow_{\mathbf{v}} M'_i$

这个关系满足菱形性质。

定理 4.6 ($\hookrightarrow_{\mathbf{v}}$ 的菱形性质) 如果 $L \hookrightarrow_{\mathbf{v}} M$ 且 $L \hookrightarrow_{\mathbf{v}} N$，那么存在表达式 K 使 $M \hookrightarrow_{\mathbf{v}} K$ 且 $M \hookrightarrow_{\mathbf{v}} K$。

定理 4.6 的证明：基于 $L \hookrightarrow_{\mathbf{v}}$ 证明树结构的归约进行证明。

- 奠基步骤
  - **当** $L =_\alpha M$ **时**
    令 $K = N$，命题得证。
  - **当** $L = (o^n\ b_1 \ldots\ b_{\mathbf{n}}), M = \sigma(o^n, b_1, \ldots b_{\mathbf{n}})$ **时**
    因为 σ 是函数，所以没有其他可能对 L 的归约。令 $K = N = M$ 即可。
- 归纳步骤
  - **当** $L = ((\lambda X.M_0)\ L_1), M = M'_0[X \leftarrow V], M_0 \hookrightarrow_{\mathbf{v}} M'_0, L_1 \hookrightarrow_{\mathbf{v}} V$ **时**
    $L \hookrightarrow_{\mathbf{v}} N$ 有两种可能的归约路径：要么应用的两部分各自完全分开归约，要么该应用使用 $\beta_{\mathbf{v}}$ 归约：
    * **当** $L = ((\lambda X.M_0\ L_1)), N = ((\lambda X.M''_0)\ L''_1), M_0 \hookrightarrow_{\mathbf{v}} M''_0, L_1 \hookrightarrow_{\mathbf{v}} L''_1$ **时**
      在这种情况下 M_0，M'_0 和 M''_0 满足归约假设的前项，L_1，V 和 L''_1 也满足。因此，存在补全两个菱形上半部分的表达式 K_0 和 K_1。我们如果知道替换和并行归约是可交换的，就可以根据 $((\lambda X.M''_0)\ L''_1) \hookrightarrow_{\mathbf{v}} K_0[X \leftarrow K_1]$ 和 $M = M'_0[X \leftarrow V] \hookrightarrow_{\mathbf{v}} K'[X \leftarrow K_1]$ 得到 $K = K_0[X \leftarrow K_1]$。
    * **当** $L = ((\lambda X.M_0)\ L_1), M = M''_0[X \leftarrow V'], M_0 \hookrightarrow_{\mathbf{v}} M''_0, L_1 \hookrightarrow_{\mathbf{v}} V'$ **时**

就像在第一个子情况中，M_0，M_0'' 和 M_0''' 以及 L_1，V 和 L_1'' 决定的菱形的上半部分。根据归纳假设，每个都产生了两个菱形的下半部分和相关项 K_0 和 K_1。再者，如果替换和并行归约是可交换的，则令 $K = K_0[X \leftarrow V]$ 即可得出该子情况。

为了补全这个情况，我们必须证明其交换性质。不过我们推迟这个证明，因为还有其他的情况依赖它，见引理 4.7。

- **当** $L = (L_1\ L_2), M = (M_1\ M_2), L_i \hookrightarrow_{\mathbf{v}} M_i$ **时**和前一个情况相同。
- **当** $L = (o^n\ L_1 \ldots L_{\mathbf{n}})$，$M = (o^n\ M_1 \ldots M_{\mathbf{n}})$，$L_i \hookrightarrow_{\mathbf{v}} M_i$ **时**有两种情况：
  * **当** $L_1 = b_1, \ldots L_{\mathbf{n}} = b_{\mathbf{n}}$，$N = \sigma(o^n, b_1, \ldots b_{\mathbf{n}})$ **时**
    那么 $M = L$ 且 $K = N$
  * **当** $N = (o^n\ N_1 \ldots N_{\mathbf{n}})$，$L_i \hookrightarrow_{\mathbf{v}} N_i$ **时**
    一个 n 折的归纳假设可以得出结论。
- **当** $L = (\lambda X.L_0)$，$M = (\lambda X.M_0)$，$L_0 \hookrightarrow_{\mathbf{v}} M_0$ **时**
  N 唯一的可能是 $N = (\lambda X, N_0)$ 且 $L_0 \hookrightarrow_{\mathbf{v}} N_0$。那么归纳假设适用于 L_0，M_0 和 N_0 且产生表达式 K_0，有 $K = (\lambda X.K_0)$。

为了完成上述证明，我们需要证明表达式关于并行归约相关的表达式中的替换不影响归约步骤。

引理 4.7 如果 $M \hookrightarrow_{\mathbf{v}} M'$ 且 $N \hookrightarrow_{\mathbf{v}} N'$ 那么 $M[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M'[X \leftarrow N']$。

引理 4.7 的证明：根据 $M \hookrightarrow_{\mathbf{v}} M'$ 证明树的归约进行证明：

- 奠基步骤
  - **当** $M =_\alpha M'$ **时**
    在这种特殊情况下，这个命题实际上等同于如果 $N \hookrightarrow_{\mathbf{v}} N'$，那么 $M[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M[X \leftarrow N']$。这个特殊结论的证明是基于 M 结构的归约。我们将这个证明留作练习。
  - **当** $M = (o^n b_1 \ldots b_{\mathbf{n}})$，$M' = \sigma(o^n, b_1, \ldots b_{\mathbf{n}})$ **时**
    M 和 M' 是封闭的，所以 $M[X \leftarrow N] = M$ 且 $M'[X \leftarrow N] = M'$，而且我们已经知道 $M \hookrightarrow_{\mathbf{v}} M'$。
- 归纳步骤
  - **当** $M = ((\lambda X_0.M_0)\ L_0)$，$M' = M_0'[X_0 \leftarrow V]$，$M_0 \hookrightarrow_{\mathbf{v}} M_0'$，$L_0 \hookrightarrow_{\mathbf{v}} V$ **时**。
    根据归纳，$M_0[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M_0'[X \leftarrow N']$ 且 $L_0[X \leftarrow N] \hookrightarrow_{\mathbf{v}} V[X \leftarrow N']$。因此：

$$\begin{array}{rcll} M[X \leftarrow N] & = & ((\lambda X_0.M_0[X \leftarrow N])\ L_0[X \leftarrow N]) & \\ & \hookrightarrow_{\mathbf{v}} & M_0'[X \leftarrow N'][X_0 \leftarrow V[X \leftarrow N']] & \\ & = & M_0'[X_0 \leftarrow V][X \leftarrow N'] & (\dagger) \\ & =_\alpha & M'[X \leftarrow N'] & \end{array}$$

等式 (†) 是替换函数的基本性质。我们将这一步骤留作练习。

- **当** $M = (M_1\ M_2)$，$M' = (M_1'\ M_2')$，$M_i \hookrightarrow_{\mathbf{v}} M_i'$ **时**
  根据归约，对 $i = 1$ 和 $i = 2$ 都有 $M_i[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M_i'[X \leftarrow N']$，因此有

$$\begin{array}{rcl} M[X \leftarrow N] & = & (M_1[X \leftarrow N]\ M_2[X \leftarrow N]) \\ & \hookrightarrow_{\mathbf{v}} & (M_1'[X \leftarrow N]\ M_2'[X \leftarrow N]) \\ & \hookrightarrow_{\mathbf{v}} & (M_1'[X \leftarrow N']\ M_2'[X \leftarrow N']) \\ & = & (M_1'\ M_2')[X \leftarrow N'] \\ & = & M'[X \leftarrow N'] \end{array}$$

  所以论题得证。
- **当** $M = (o^n\ M_1 \ldots\ M_{\mathbf{n}})$，$M' = (o^n\ M_1' \ldots\ M_{\mathbf{n}}')$，$M_i \hookrightarrow_{\mathbf{v}} M_i'$ **时**
  和上面的情况相同。
- **当** $M = (\lambda X.M_0)$，$M = (\lambda X.M_0')$，$M_0 \hookrightarrow_{\mathbf{v}} M_0'$ **时**
  和上面的情况相同。

▷ **练习 4.3.** 证明如果 $N \hookrightarrow_{\mathbf{v}} N'$，那么 $M[X \leftarrow N] \hookrightarrow_{\mathbf{v}} M[X \leftarrow N']$。

▷ **练习 4.4.** 证明如果 $X \notin \mathcal{FV}(L)$，那么

$$K[X \leftarrow L][X' \leftarrow M[X \leftarrow L]] =_\alpha K[X' \leftarrow M][X \leftarrow L]$$

▷ **练习 4.5.** 证明并行关系 $\hookrightarrow_{\mathbf{v}}$ 的传递自反闭包和 $\twoheadrightarrow_{\mathbf{v}}$ 相同。这个结论连同定理 4.6 的证明，可支持定理 4.5 的证明。

4.6 观察等价性

除了程序求值器，程序转换在编程实践和编程语言实现时都发挥了很重要的作用。例如，如果 M 是一个程序，如果我们认为程序 N 的计算行为和 M 相同但是计算得更快，那么我们就要证明 M 是否真的可以和 N 互换。ISWIM 为这个问题的一个特殊案例提供了一个清晰的方案：如果两个表达式都是完整的程序且我们知道 $M =_{\mathbf{v}} N$，那么我们可得 $eval_{\mathbf{v}}(M) = eval_{\mathbf{v}}(N)$ 且可知能够使用 M 替换 N。

但是一般来说，M 和 N 是一些程序中的子表达式，而且我们可能不能将它们独立求值。因此，我们真正需要的是一个可以解释什么时候表达式是“功能性等价”的一般性关系。

为了理解“功能性等价”的概念，我们必须回想程序的使用者仅能“观察”到输出，也对输出最感兴趣。程序员通常不知道程序文本或任何关于文本的概念。换句话说，这样的观察者将程序当作使用某种未知规则而产生某个值的黑箱。因此，表达式的影响自然是其作为程序一部分所产生的影响。甚至更进一

步地说，两个开放表达式的比较可以归约成是否存在一种辨别表达式影响的方法的问题。或者更正式地说，如果两个表达式具有相同的影响，那么从外部的观察者的视角看来，它们是可以交换的。

在继续之前，我们需要表达式上下文的概念。下面的文法 C 定义了一种贴近表达式的集合。他们不是严格的表达式，因为 C 的每个元素都在其中一个子表达式上存在一个**洞**，记为 []。

$$
\begin{array}{rcl}
C & = & [] \\
 & | & (\lambda X.C) \\
 & | & (C\ M) \\
 & | & (M\ C) \\
 & | & (o^n\ M\ \ldots\ M\ C\ M\ \ldots\ M)
\end{array}
$$

$C[M]$意味着“将 C 中的 [] 替换为 M”

例如，$(\lambda x.[])$ 和 $(+\ 1\ ((\lambda x.\ulcorner 0\urcorner)\ []))$ 是 C 的元素，且

$$
\begin{array}{lcl}
(\lambda x.[])[(\lambda y.x)] & = & (\lambda x.(\lambda y.x)) \\
(+\ 1\ ((\lambda x.\ulcorner 0\urcorner)\ []))[(z\ \ulcorner 12\urcorner)] & = & (+\ 1\ ((\lambda x.\ulcorner 0\urcorner)\ (z\ \ulcorner 12\urcorner)))
\end{array}
$$

与替换不同的是，填补一个上下文洞可以捕捉变量。

根据这个上下文定义，我们可以表示观察等价性的概念，两个表达式 M 和 N 是**观察等价**的，当且仅当它们在所有上下文 C 中都没有差别，记为 $M \simeq_{\mathbf{v}} N$。

$M \simeq_{\mathbf{v}} N$ 如果对所有的 C，有$eval_{\mathbf{v}}(C[M]) = eval_{\mathbf{v}}(C[N])$

这个定义暗示了如果 $M \simeq_{\mathbf{v}} N$，那么 $eval_{\mathbf{v}}(C[M])$ 和 $eval_{\mathbf{v}}(C[N])$ 对任何 C，要么都被定义了，要么都没被定义。

观察等价性明显的拓展了在 $eval_{\mathbf{v}}$ 意义上的程序等价性。如果程序 M 和 N 是观察等价的，那么他们空集上下文中就是程序，所以 $eval_{\mathbf{v}}(M) = eval_{\mathbf{v}}(N)$。但是至少我们也希望在等价功能表达式理论中获得这些信息了。如其名所示，观察等价性是一个等价关系。最后，如果两个表达式是观察等价的，那么将它们嵌入到上下文中，也会产生等价表达式。毕竟被加入的部分是恒等的且不应该影响表达式在程序输出上可能产生的效果。简而言之，观察等价性是一个**同余关系**，也就是说，它基于上下文的闭包也是一个等价关系。

定理 4.8 对任何表达式 M、N 和 L 都有

1. $M \simeq_{\mathbf{v}} M$

2. 如果 $L \simeq_{\mathbf{v}} M$ 且 $M \simeq_{\mathbf{v}} N$，那么 $L \simeq_{\mathbf{v}} N$

3. 如果 $L \simeq_{\mathbf{v}} M$，那么 $M \simeq_{\mathbf{v}} L$

4. 如果 $M \simeq_{\mathbf{v}} N$，那么对于所有的上下文 C 都有 $C[M] \simeq_{\mathbf{v}} C[N]$

定理 4.8 的证明：通过 3 可知第 1 点是平凡的。至于第四点，假设 $M \simeq_{\mathbf{v}} N$ 且令 C' 为一个任意的上下文。那么 $C'[C]$ 是 M 和 N 的上下文。因此根据假设 $M \simeq_{\mathbf{v}} N$ 可得

$$eval_{\mathbf{v}}(C'[C[M]]) = eval_{\mathbf{v}}(C'[C[N]])$$

该等式是我们必须证明的。

实际上观察等价性是满足我们直观标准的表达式的最大同余关系。

定理 4.9 令 $\mathbf{R}$ 是同余关系，且对表达式 M、N 有，通过 $M\ \mathbf{R}\ N$ 可得 $eval_{\mathbf{v}}(M) = eval_{\mathbf{v}}(N)$。如果 $M\ \mathbf{R}\ N$，那么 $M \simeq_{\mathbf{v}} N$。

定理 4.9 的证明：我们假设 $M \not\simeq_{\mathbf{v}} N$ 且显示 $M\ \not\mathbf{R}\ N$。根据假设存在一个分离的上下文 C。也就是说，$eval_{\mathbf{v}}(C[M]) \neq eval_{\mathbf{v}}(C[N])$。因此，$C[M]\ \not\mathbf{R}\ C[N]$。因为 $\mathbf{R}$ 被假设是一个同余关系，我们不能得出 $M \equiv N$，因为这是矛盾的。

这个命题证明了观察等价性是表达式最基本、最基础的同余关系。所有其他的关系仅仅接近观察等价性定义表达式是所需要的精确性。就观察定价性而言，它也遵循 ISWIM 的稳固性命题。但遗憾的是，它也是不完整的，这个演算不能证明所有的观察等价关系。

定理 4.10 (稳固性，不完整性) 如果我们能证明 $M =_{\mathbf{v}} N$，那么 $M \simeq_{\mathbf{v}} N$。但是 $M \simeq_{\mathbf{v}} N$ 并不意味着能推出 $M =_{\mathbf{v}} N$。

定理 4.10 的证明：根据定义，$=_{\mathbf{v}}$ 是同余关系。此外 $eval_{\mathbf{v}}$ 是基于 $=_{\mathbf{v}}$ 定义的，也就是说如果 $M =_{\mathbf{v}} N$ 则 $eval_{\mathbf{v}}(M) = eval_{\mathbf{v}}(N)$。
对于相反的情况，我们可以给出一个反例并简述该反例的证明。例如表达式 $(\Omega\ (\lambda x.x))$ 和 Ω，其中 $\Omega = ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$，这两个表达式都是发散的且观察等价的。但是二者都仅归约到自身，因此不能被证明是关系 $=_{\mathbf{v}}$。

我们仍然缺少一些工具去完成不完全性定理的证明。我们在下一章拓展了关于 ISWIM 的一些拔高于基本知识的内容，同时也将继续这个证明。

▷ **练习 4.6.** 考虑下列求值函数 $eval_0$，以及其相关的观察等价关系 $\simeq_0$：

$eval_0(M) = \mathbf{value}$	如果对某些 V 有$M =_{\mathbf{v}} V$
$M \simeq_0 N$	如果对所有的 C，有$eval_0(C[M]) = eval_0(C[N])$

$M \simeq_0 N$ 能推导出任何关于 $M \simeq_{\mathbf{v}} N$ 的结论吗？简述你答案的理由。

4.7 历史

在二十世纪 60 年代中期的一系列论文中 [5]，Landin 详述了两个关于编程语言的重要发现。第一，他认为所有的编程语言都有一套用于详细描述计算的共享基元集合[2]，但是对数据和数据元的选择有所不同。基本的基元集合包括命名、过程、应用、异常机制、可变数据结构以及可能的其他形式的非局部控制。数值应用语言通常包含多种形式的数值常量和大量的数值元集合，然而字符串操作语言一般提供高效的字符串匹配和操作元。

第二，他极力主张程序员和语言实现者一样，都应把编程语言当成算数与代数的一种高级的符号化形式。因为我们都习惯于使用数值、布尔，甚至我们从幼儿园到高中期间学习的更为复杂的数据结构来进行计算，这些计算也应该易于被程序计算。程序求值、多种形式的程序编辑、程序转换和优化都只是不同的、更为复杂的计算形式。这类计算处理关于程序或程序片段的问题，而不是简单的算数表达式。

Landin 定义了编程语言 ISWIM。他的设计基于丘奇的 λ 演算——从 Landin 认为程序过程的核心作用是作为所有语言的共享基元的观点看，这是一个自然会想到的着手点。但是，为了支持基本数据以及赋值和控制结构的相关元，Landin 通过恰当的结构拓展了 λ 演算。他通过一种抽象机器指定了这个拓展语言的语义，因为他不知道如何将 λ 演算的等价理论拓展到完整的编程语言理论中。实际上，事实证明 λ 演算甚至不能解释纯函数式子语言的语义，因为 ISWIM 总是对程序过程的参数进程求值。因此，Landin 没有完成自己着手去做的事情，也就是没有定义理想化的所有编程语言的核心以及定义其语义的等价演算。

自二十世纪 70 年代中叶 Plotkin 关于抽象机器和等价演算的关系的工作开始后，就有大量的研究者，包括 Felleisen，Mason，Talcott 以及他们的合作者，补充 Landin 研究中的不足。Plotkin 的工作涵盖了 ISWIM 的基本函数式子语言，这需要 λ 演算的按值调用变量的定义。Felleisen 和他的合作者根据若干种可解释不同类型的必要语言基元的公理，拓展了等价性理论。Mason 和 Talcott 研究了在完全类 ISWIM 语言作为一种程序验证和程序转换的工具的等价性理论的应用。

虽然 ISWIM 没有成为广泛使用的语言，但是 ISWIM 的哲学仍存在于现代编程语言中，尤其是 Scheme 和 ML 语言中，而其语言分析和设计方法基本应用于所有编程语言中。本书的另一个目标是说明等价性理论的设计、分析和应用，例如在编程语言的设计和分析背景下的 λ 演算。

[2]译者注：原文为“a basic set of facilities”