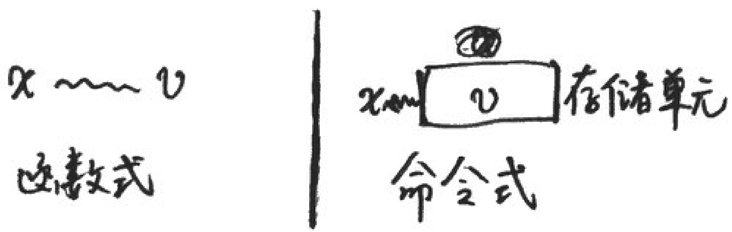
# 存储器和指针

「函数式」的特性已经几近被我们网罗尽了，下面是时候向「命令式」进军了。二者的不同在哪儿呢？一言以蔽之：函数式编程更高层，不关心内存的问题，变元符号仅仅是指代抽象值的符号；命令式编程更底层，涉及内存的问题，变量名与内存地址绑定在一起，指代内存中的值。



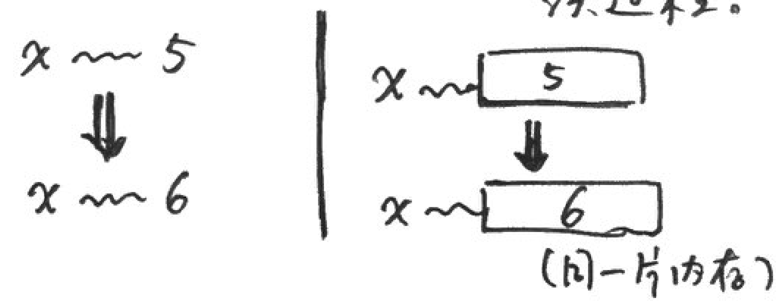

因而，用λ演算写

```
let x = 5 in
  let x = x+1 in
    x
```

和用C语言写

```
int x=5;
x = x+1;
```

的效果是不同的。前者只是重新绑定了x与值；后者却改变了存储单元中的内容，且该动永久地影响后续过程。




于是，在命令式语言中，语句的执行会产生「副作用」，并通过存储器保持下来，对今后持续产生影响。这也是为什么命令式语言可以用for循环对一个变量反复迭代，并计算出所需结果，而函数式语言则须通过递归方可实现。

你或许要问：命令式又有什么好呢？

从纯粹计算的角度讲，函数式语言更高级、简洁，但是，如今的计算机不只能胜任「计算」，还能输入、产出甚至交互。在纯计算以外的领域，「副作用」是有必要的。试想一个最简单的记帐系统，人们用它来管理、保存帐目，并且要求它能「随调随用」甚至具备「共享」功能。凭借纯粹的函数式语言，我们根本无法把帐目信息记录到存储器上，或是通过网络传输给别人。待我们重新运行之时，一切又打回原形，前功尽弃。

事实上，从另一个角度想，也能证明「命令式」的必要性。计算机的架构完全是基于存储器及状态转移的，因此，计算机软件无法完全脱离存储的概念。（至少，编译器和操作系统无法用纯函数式实现。）

既然要引入存储器，那么就要有如下两点考虑：

1° 程序运行不再是单纯的「代码变换」，而应为「代码＋存储器内容的变换」。单步运行的判断形式势必得改成

$$\langle M, t\rangle \longrightarrow \langle M', t'\rangle$$

其中M, M'是存储器内容，t与t'则为代码。

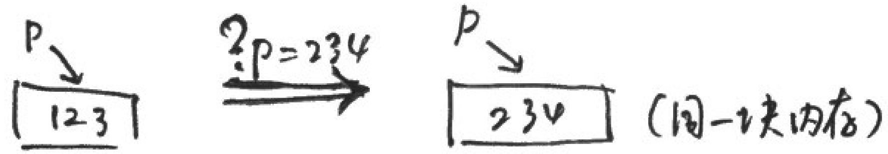
2° 必须提供读、写存储器的接口，以便程序员操纵存储器。

其中，1°是语义层面的，2°是语法层面的。我们从2°说起。我们可设计如下的原型：

- "alloc 123" 将开辟一块内存单元，往里面放入值123，并返回指向

该内存区域的指针。(功能类似于 C 的 malloc，只不过多了一个初始化操作。)

- 设 $p$ 是一个指针，那么"?p"将取出 $p$ 所指的内容。(类似于 C 的 *p)
- 设 $p$ 是一个指针，那么"?p=234"将把 $p$ 所指的内容改写为 234。(类似于 C 的 *p=234)




## 顺序执行的

- 利用";"间隔语句，比如

```
let p = alloc 1 in
  ?p := ?p+10;
  ?p := ?p/2;
  ?p
```

之所以要引入这一特性，是因为我们现在正「面向副作用编程」，顺序执行的逻辑是最常见的。

## def 表达式

$t ::= \ldots \mid \text{alloc } t \mid ?t \mid ?t=t \mid l \mid t;t \mid \text{null}$

其中 $l$ 比较特殊，用于代指「内存地址」，其取值范围应预先定义好，比如 $\mathbb{Z}$。程序员书写程序时是用不着它的；它仅为解释器所用。null 是一个占位符似的无用之物，用途以后再说。

这里，我们采取了 ML 和 Java 中的处理，不允许用户手动操纵地址，因而灵活性稍逊于 C/C++；可持久化数据始终得凭借 C/C++ 的类库来获取。但扩展之并不困难。

## def 值

$v ::= \ldots \mid l \mid \text{null}$

## def 类型

$T ::= \ldots \mid T\ \text{Ptr} \mid \text{Null}$

一眼可看出 Null 类型是与 null 匹配的。简单说，null/Null 即「空」；即「没什么好说的」；即「结果不重要，重点看过程」。

下面来看1°（语义）。因为仅有「alloc …」和「?t := …」两种语句可能修改存储器，所以，在下面的定义中，这两种情形值得重点关注。

**def.** 单步执行

判断形式：$\langle M, t\rangle \to \langle M', t'\rangle$

判断规则：

$$\frac{\langle M, t\rangle \to \langle M', t'\rangle}{\langle M, \mathrm{alloc}\ t\rangle \to \langle M', \mathrm{alloc}\ t'\rangle}\ [\text{E-ALLOC}]$$

★ $$\frac{l \text{ 是存储器 } M \text{ 中空闲的单元} \rightsquigarrow \text{简写为 } l \notin \mathrm{dom}\, M}{\langle M, \mathrm{alloc}\ v\rangle \to \langle M[l \mapsto v], l\rangle}\ [\text{E-ALLOC-VAL}]$$

$$\frac{\langle M, t\rangle \to \langle M', t'\rangle}{\langle M, ?t\rangle \to \langle M', ?t'\rangle}\ [\text{E-RETRIEVE}]$$

$$\frac{l \in \mathrm{dom}\, M \quad M(l) = v}{\langle M, ?l\rangle \to \langle M, v\rangle}\ [\text{E-RETRIEVE-VAL}]$$

$$\frac{\langle M, t_1\rangle \to \langle M', t_1'\rangle}{\langle M, ?t_1 := t_2\rangle \to \langle M', ?t_1' = t_2\rangle}\ [\text{E-ASSIGN-L}]$$

$$\frac{\langle M, t_2\rangle \to \langle M', t_2'\rangle}{\langle M, ?v := t_2\rangle \to \langle M', ?v := t_2'\rangle}\ [\text{E-ASSIGN-R}]$$

★ $$\frac{l \in \mathrm{dom}\, M}{\langle M, ?l := v\rangle \to \langle M[l \mapsto v], \mathrm{null}\rangle}\ [\text{E-ASSIGN}]$$

$$\frac{\langle M, t_1\rangle \to \langle M', t_1'\rangle}{\langle M, t_1 t_2\rangle \to \langle M', t_1' t_2\rangle}\ [\text{E-APPLY-L}]$$

$$\frac{\langle M, t_2\rangle \to \langle M', t_2'\rangle}{\langle M, v_1 t_2\rangle \to \langle M', v_1 t_2'\rangle}\ [\text{E-APPLY-R}]$$

$$\frac{}{\langle M, (\lambda x.t) v_2\rangle \to \langle M, t[x/v_2]\rangle}\ [\text{E-APPLY}]$$

$$\frac{\langle M, t_1\rangle \to \langle M', t_1'\rangle}{\langle M, \mathrm{let}\ x = t_1\ \mathrm{in}\ t_2\rangle \to \langle M', \mathrm{let}\ x = t_1'\ \mathrm{in}\ t_2\rangle}\ [\text{E-LET-L}]$$

$$\frac{}{\langle M, \mathrm{let}\ x = v_1\ \mathrm{in}\ t_2\rangle \to \langle M, t_2[x/v_1]\rangle}\ [\text{E-LET}]$$

$$\frac{\langle M, t_1\rangle \to \langle M', t_1'\rangle}{\langle M, t_1; t_2\rangle \to \langle M', t_1'; t_2\rangle}\ [\text{E-SEQ-L}]$$

$$\frac{}{\langle M, \mathrm{null}; t_2\rangle \to \langle M, t_2\rangle}\ [\text{E-SEQ-R}]$$

先讲[E-ALLOC-VAL]。当解释器拿到程序 alloc $v$ 时，它会立即辟出一块新的内存单元存放值 $v$，并返回指针 $l$。这里，我们把存储器 $M$ 视作一张由地址到值的映射表：

| location | value |
|---|---|
| $l_1$ | $v_1$ |
| $l_2$ | $v_2$ |
| $\vdots$ | $\vdots$ |
| $l_n$ | $v_n$ |
| 暂未使用 | |

而 $M[l \mapsto v]$ 的意思即把地址 $l$ 映到 $v$。因为 $l$ 原来不在 $M$ 中，故这相当于为 $M$ 新增了一个条目。

remark. 为了使单步运行结果唯一确定，我们可限定 $l$ 只能取 $M$ 中最靠前的单元。

再讲[E-ASSIGN]。当解释器拿到程序 $?l := v$ 时，它会尝试把 $l$ 所指的存储单元内容改动为 $v$，不过，为防非法访问，前提是 $M$ 中的确开辟过 $l$ 这块空间。如果没有，则程序就此卡住。[E-RETRIEVE]也是一样的。

余下的规则均不涉及存储器的读写，因此，$M \to M'$ 的转移全是从子过程继承下来的。

remark. 为什么[E-ASSIGN]的执行结果是 null 呢？这是因为「$?l := v$」目的是「改变存储器」这一副作用，人们并不在乎其结果。该语句与[E-SEQ-R]放在一起看，你就会明白顺序执行的含义。

有了单步执行的语义，类型推导规则就不难书写了。由于我们的语言不允许用户手动操作地址，所以，程序员所取得的地址只有一个源头：他们自己

通过alloc申请得来。于是，在程序运行以前存储器上有什么内容，程序都是无法读或写的。那么，开始运行程序的时候，存储器是空亦或非空，是完全无所谓的；程序本身的类型，也完全不依赖于存储器里的内容。因此，类型推导无需考虑M。

def 类型推导关系 $\vdash$.（原有规则省略）

$$\frac{\Gamma \vdash t:T}{\Gamma \vdash (\text{alloc}\ t):T\,Ptr}\ [\text{T-ALLOC}] \qquad \frac{\Gamma \vdash t:T\,Ptr}{\Gamma \vdash\ ?t:T}\ [\text{T-RETRIEVE}]$$

$$\frac{\Gamma \vdash t_1:T\,Ptr \quad \Gamma \vdash t_2:T}{\Gamma \vdash (?t_1:=t_2):Null}\ [\text{T-ASSIGN}]$$

保证内存单元中数据类型不变

$$\frac{\Gamma \vdash t_1:Null \quad \Gamma \vdash t_2:T}{\Gamma \vdash (t_1;t_2):T}\ [\text{T-SEQ}]$$

remark. 但若我们考虑的是诸如C/C++这种能掌控硬件、读取任意地址数据的语言时，情况就不同了。这时候，存储器上原有的数据的确能被访问、修改，因而类型推导关系必须涉及M中的内容。

经过几章的训练，你或许已感觉到类型系统实际上就是个弱化了的运行系统。T规则，是把若干相关的E规则整合后得到的。T规则排除掉了单步运行的琐碎细节，直接一眼望穿结果。

但当我们考察类型安全性的时候，「一眼望穿」的缺陷就体现出来了。须知，类型安全性的两大定理(Preservation & Progress)讲述的都是与单步运行相关的性质：

1° Preservation讲的是，类型为T的程序在单步运行后，类型保持为T

2° Progress讲的是，一个有类型的程序，在化为值以前决不卡住，总能单步运行下去。

那么好，请看以下程序：

$$\begin{array}{l|l} \text{let } p = \text{alloc } 123 \text{ in} & \\ \quad ?p := ?p + 2 & M_0 \end{array}$$

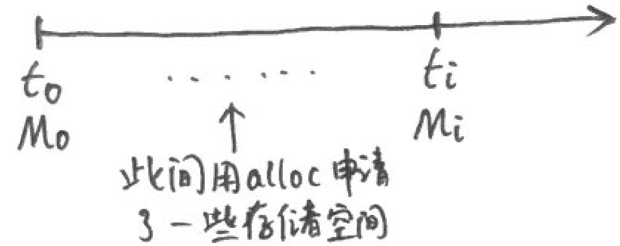
根据前面定义的类型规则，易推导出其类型为 Null。运行一步，得

$$\begin{array}{l|l} \to \text{let } p = l \text{ in} & M_0 \cup \{l \mapsto 1\} \\ \quad ?p := ?p + 2 & (\text{或写成 } M_0[l \mapsto 1]) \end{array}$$

这下子麻烦了。根据 Preservation 的要求，程序应当保持类型为 Null，但是，根据前面定义的类型规则，却推不出当前程序的类型。这么一来，系统岂不是就不安全了？

事实上并非如此。如果我们把 M 也考虑进来，全面地分析一下，本应能推出类型为 Null 的。只不过前面的类型规则主动地忽略掉了 M，才无法证明类型安全性。

画一条时间轴以助理解：




最开始的时候，$t_0$ 手中不握有 $M_0$ 中的地址，因而如以前所述，$M_0$ 中的内容与 $t_0$ 的类型无瓜葛，可忽略之。

待程序运行了一阵，申请了一些存储空间，来到 $\langle M_i, t_i \rangle$ 时，事情就不同了。$t_i$ 手中已握有 $M_i$ 中的（部分）地址，因此，$M_i$ 中的内容会影响到 $t_i$ 的类型。

经过这一轮分析，我们明确：前面定义的类型推导规则仅在一开始适用，待程序运行起来后就不够全面了。由于类型安全性讨论的是单步运行前后的关系，所以为了证明安全性，

必须扩展类型规则，把M也囊括进来。

怎么囊括法呢？其实上一个例子已能给我们诸多启发：既然$l$的类型与M相关，那么不妨建立一个「类型仓库」Σ，专门存放M中各存储单元的类型，比如

$\Sigma(l) = Int$。

M:

| location | value |
| --- | --- |
| $l_1$ | $v_1$ |
| ⋮ | ⋮ |
| $l_n$ | $v_n$ |

Σ:

| location | type |
| --- | --- |
| $l_1$ | type of $v_1$ |
| ⋮ | ⋮ |
| $l_n$ | type of $v_n$ |

然后在类型推导时，可直接取用Σ中的信息。很自然地，类型推导关系须由原先的三元关系 $\Gamma \vdash t:T$ 晋升为四元关系 $\Gamma;\Sigma \vdash t:T$.

def 类型推导关系 (扩展版)

判断形式: $\Gamma;\Sigma \vdash t:T$

判断规则: ★ $$\frac{l \in dom\,\Sigma \quad \Sigma(l)=T}{\Gamma;\Sigma \vdash l : T\,Ptr}\ [\text{T-DIRECT}]$$

$$\frac{\Gamma;\Sigma \vdash t:T}{\Gamma;\Sigma \vdash (alloc\ t) : T\,Ptr}\ [\text{T-ALLOC}] \qquad \frac{\Gamma;\Sigma \vdash t : T\,Ptr}{\Gamma;\Sigma \vdash\ ?t : T}\ [\text{T-RETRIEVE}]$$

$$\frac{\Gamma;\Sigma \vdash t_1 : T\,Ptr \quad \Gamma;\Sigma \vdash t_2 : T}{\Gamma;\Sigma \vdash (?t_1 := t_2) : Null}\ [\text{T-ASSIGN}]$$

$$\frac{\Gamma;\Sigma \vdash t_1 : T \blacktriangleright T' \quad \Gamma;\Sigma \vdash t_2 : T}{\Gamma;\Sigma \vdash (t_1\ t_2) : T'}\ [\text{T-APPLY}]$$

$$\frac{\Gamma \cup \{x:T\};\Sigma \vdash t : T'}{\Gamma;\Sigma \vdash (\lambda x{:}T.t) : T \blacktriangleright T'}\ [\text{T-FUNCTION}]$$

$$\frac{\Gamma;\Sigma \vdash t_1 : T \quad \Gamma \cup \{x:T\};\Sigma \vdash t_2 : T'}{\Gamma;\Sigma \vdash (let\ x=t_1\ in\ t_2) : T'}\ [\text{T-LET}]$$

$$\frac{\Gamma;\Sigma \vdash t_1 : Null \quad \Gamma;\Sigma \vdash t_2 : T}{\Gamma;\Sigma \vdash (t_1 ; t_2) : T}\ [\text{T-SEQ}]$$

说实话，除了第一条规则，Σ就是条鸡肋，生硬地安插进原有规则，却形同无物。因此我们只讲第一条规则：

手中拿到地址$l$，我们首先确认$l$地址在Σ中的确有记录，接下去再取出$l$中值的类型 $\Sigma(l)=T$，最

后，断言$l$的类型是 $T\ Ptr$，因为$l$指向了一个存有$T$型值的存储单元。

$l$ is a pointer to type-$T$ value.

$\boxed{v \text{ has type } T}$

类型推导关系弄明白了，你接下去要问：怎样保证$\Sigma$与$M$的一致性呢？或者说：怎样约束$\Sigma$的确正确地反映了$M$中各单元的类型呢？下面的定义即为约束：

def $\Sigma$与$M$的一致性.

设$M$是存储器，而$\Sigma$是类型仓库。如果 $\operatorname{dom} M = \operatorname{dom} \Sigma$，而且 $\forall l \in \operatorname{dom} M$ 均有 $\Gamma;\Sigma \vdash M(l):\Sigma(l)$，则称$\Sigma$与$M$在$\Gamma$意义下一致，记作 $\Sigma \overset{\Gamma}{\sim} M$。（$\Gamma$是假设集.）

$M$

| location | value |
|---|---|
| $l_1$ | $v_1 = M(l_1)$ |
| $l_2$ | $v_2 = M(l_2)$ |
| $\vdots$ | $\vdots$ |
| $l_n$ | $v_n = M(l_n)$ |

$\overset{\Gamma}{\sim} \Sigma$

| location | type |
|---|---|
| $l_1$ | $T_1 = \Sigma(l_1) = \text{type of } v_1$ |
| $l_2$ | $T_2 = \Sigma(l_2) = \text{type of } v_2$ |
| $\vdots$ | $\vdots$ |
| $l_n$ | $T_n = \Sigma(l_n) = \text{type of } v_n$ |

e.g.1 $\forall \Gamma$,

$M$

| | |
|---|---|
| $l_1$ | 134 |
| $l_2$ | True |
| $l_3$ | $\lambda x:Int.\,x+1$ |

$\overset{\Gamma}{\sim} \Sigma$

| | |
|---|---|
| $l_1$ | Int |
| $l_2$ | Bool |
| $l_3$ | $Int \blacktriangleright Int$ |

e.g.2

$M$

| | |
|---|---|
| $l_1$ | 134 |

$\not\sim \Sigma$

| | |
|---|---|
| $l_2$ | Int |

e.g.3 $\Gamma = \{y:Int,\ f:Int \blacktriangleright Bool\}$

$M$

| | |
|---|---|
| $l_1$ | $\lambda x:Int.\,y$ |
| $l_2$ | $f$ |

$\overset{\Gamma}{\sim} \Sigma$

| | |
|---|---|
| $l_1$ | $Int \blacktriangleright Int$ |
| $l_2$ | $Int \blacktriangleright Bool$ |

e.g.4 $\Gamma = \emptyset$

$M$

| | |
|---|---|
| $l_1$ | $\lambda x:Int.\,(?l_2)0$ |
| $l_2$ | $\lambda x:Int.\,(?l_1)0$ |

$\overset{\Gamma}{\sim} \Sigma$

| | |
|---|---|
| $l_1$ | $Int \blacktriangleright Int$ |
| $l_2$ | $Int \blacktriangleright Int$ |

$\overset{\Gamma}{\sim} \Sigma'$

| | |
|---|---|
| $l_1$ | $Int \blacktriangleright Bool$ |
| $l_2$ | $Int \blacktriangleright Bool$ |

...

末尾一例最有趣。此例的M中存在着「循环引用」，因此，单看M无法弄清其中值的类型，惟有以Σ为前提才可推出类型。Σ不同，则类型不同；有多个Σ与M一致。

可以说，在此种情形下，Σ与M中值的类型是相互耦合的。

---

problem 请你想个办法，写程序生成 e.g.4 中M的构型。写好以后，单步运行之直至结束以验证之。虽然我们暂未提及Σ应当如何一步步地变化，你认为它是怎样改变的？

---

**Theorem 1 (Preservation)**

设$t$是一个程序，$\Gamma$是假设集。若

(1) $\Sigma \simeq M$ （类型仓库与存储器一致）

(2) $\Gamma;\Sigma \vdash t:T$ （程序具有类型T）

(3) $\langle M, t\rangle \to \langle M', t'\rangle$ （程序可单步运行）

则$\exists \Sigma'$：

(1) $\Sigma' \simeq M'$ （新的类型仓库与运行后的存储器一致）

(2) $\Gamma;\Sigma' \vdash t':T$ （程序保有类型T）

remark.

看起来繁琐，说起来简单。$t$一开始具有类型$T$。运行一步后，$M$变为$M'$，$t$变为$t'$。我们把Σ「依样子」更新成Σ'以兼顾M'，则可推出$t'$保有类型$T$。

定理本可以加强为：∃唯一的$\Sigma' \geq \Sigma$

(1) $\Sigma' \simeq M'$ (2) $\Gamma;\Sigma' \vdash t':T$，

用以刻画Σ连贯、确定性的变化，从而把 e.g.4 中的不确定性排除掉。可是，这样证明起来就冗长了些。请你在阅读下面证明以后，思考如何证明加强版。

proof. 关于→作归纳。

[E-ALLOC, E-RETRIEVE, E-ASSIGN-L, E-ASSIGN-R, E-APPLY-L, E-APPLY-R, E-LET-L, E-SEQ-L, E-SEQ-R]

这些情形套路一致，不值一提.

[E-ALLOC-VAL]

已知 (1) $\Sigma \overset{\Gamma}{\sim} M$

(2) $\Gamma;\Sigma \vdash (\text{alloc } v): T$

(3) $\langle M, \text{alloc } v\rangle \to \langle M[l\mapsto v], l\rangle$ $(l \notin dom M)$

结合(2)和所有类型规则，立即可知：

(4) T必须为某种指针型，即 $T = T_0\ Ptr$

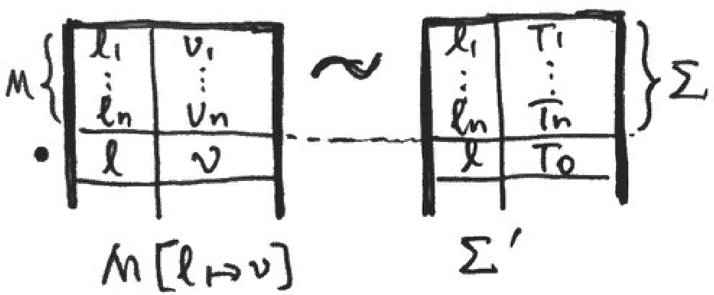
(5) $\Gamma;\Sigma \vdash v : T_0$

我们构造 $\Sigma' = \Sigma \cup \{l : T_0\}$

那么自然，

$\Gamma;\Sigma' \vdash M[l\mapsto v](l) : \Sigma'(l)$

又因为$\Sigma'$别的部分与 $M[l\mapsto v]$ 保持一致，所以总起来说 $\Sigma' \overset{\Gamma}{\sim} M[l\mapsto v]$




而根据[T-DIRECT]，显然有

$\Gamma;\Sigma' \vdash l : T_0\ Ptr$，即 $\Gamma;\Sigma' \vdash l : T$.

[E-RETRIEVE-VAL]

已知 (1) $\Sigma \overset{\Gamma}{\sim} M$

(2) $\Gamma;\Sigma \vdash (?l) : T$

(3) $\langle M, ?l\rangle \to \langle M, v\rangle$ $\begin{pmatrix} l \in dom M \\ v = M(l) \end{pmatrix}$

结合(2)和所有类型规则，可知

(4) $\Gamma;\Sigma \vdash l : T\ Ptr$

$\Rightarrow$ (5) $\Sigma(l) = T$

又由于(1)，所以

(6) $\Gamma;\Sigma \vdash v : \Sigma(l)$

即 $\Gamma;\Sigma \vdash v : T$

是故，取 $\Sigma' := \Sigma$ 即可。

[E-ASSIGN]

已知 (1) $\Sigma \overset{\Gamma}{\sim} M$

(2) $\Gamma;\Sigma \vdash (?l := v) : T$

(3) $\langle M, ?(:= v)\rangle \to \langle M[l\mapsto v], null\rangle$

由类型规则知，T只能为Null.

又因为 $\Gamma;\Sigma \vdash null : Null$

故 $\Gamma;\Sigma \vdash null : T$

另外，类型规则要求(4) $\Gamma;\Sigma \vdash l : T_0\ Ptr$

且 (5) $\Gamma;\Sigma \vdash v:T_0$

由(4)知，(6) $\Sigma(l)=T_0$

由(5)知，(7) $\Gamma;\Sigma \vdash M[l\mapsto v](l):T_0$

由(6)(7)知 (8) $\Gamma;\Sigma \vdash M[l\mapsto v](l):\Sigma(l)$

从而 $\Sigma \overset{\Gamma}{\sim} M[l\mapsto v]$.

[E-APPLY，E-LET]

这两条与从前相似，均基于替换引理 (Substitution Lemma)，此处略。■

**Theorem 2 (Progress)**

设程序 $t$ 通过类型检查（即 $\emptyset;\Sigma \vdash t:T$）又设 $\Sigma \overset{\Gamma}{\sim} M$，那么 $t$ 要么是值，要么能继续单步运行（即 $\exists t', M'$ 使 $\langle M,t\rangle \to \langle M',t'\rangle$）

proof. 关于 $\vdash$ 作归纳即可。■

两个定理相互观照，立即得到

**Theorem 3**

设程序 $t$ 通过了类型检查，具备类型 $T$，那么它在化为值以前不会卡住，而且全程均具有类型 $T$。

（「全程具有类型 $T$」的确切说法：

$\langle M,t\rangle \to \langle M_1,t_1\rangle \to \cdots \langle M_n,t_n\rangle$，

$\forall i=1,\ldots,n$ 有，$\exists \Sigma_i \overset{\Gamma}{\sim} M_i$ 使

$\emptyset;\Sigma_i \vdash t_i:T$ ）

作为本章的画龙点睛之笔，我们用一幅图点明 $\Sigma$ 与 $\Gamma$ 的功用和区别。