扩展类型系统

成熟的编程语言当然不只真值这样的基础类型，它至少还应该具备整数、元组/列表之类常用的类型。此外，我们也得想办法把「递归」兼并进来，否则语言的表达能力就太弱了。本章探讨如是的扩展。

约定

自本章起，不再用蓝色字特意标识「外部变元」。变元究竟从属于内部还是外部，请由上下文推定。

另外，随着扩充的进行，语言的字符集 Σ 也会不断扩展，我们将不再一一写明。同样，语言 Λ、单步执行关系、类型推导关系均会不断扩展。我们只做加法，不做减法。

总结上章经验，为语言添加新类型是有套路的：

(1) 确定新类型的实例有怎样的书写格式。（比如，对于 Bool 型，只要写下 True 或 False）

(2) 确定新类型本身的名字（如 Bool）

(3) 规定新类型要怎样使用/怎样运算（如在 if-then-else 中的 if 从句中使用）

(4) 确定单步执行规则

(5) 确定类型推导规则

语法	→	语义	→	类型检查
(1)(2)(3)		(4)		(5)

我们将遵循类似的道路来扩充其它类型。

1° 整数

def $T ::= \ldots$ （已有的类型）

$\quad\mid \mathit{Int}$ （整数类型）

我们想让整数（e.g. 1, -19, 8927）获得原生支持，自然得把数字符号 0-9 和负号「~」添进 Σ 中，并让语法支持之。基本的运算 +, -, *, / 当然也是免不了的（正如少了 if-then-else 以后，True 与 False 便无处可用）。为了省却优先级的烦恼，我们要求算数表达式用前缀式书写。

def $t ::= \ldots$ （已有的项）

$\quad\mid i$ （整数）

$\quad\mid +tt$ （加）

$\quad\mid -tt$ （减）

$\quad\mid *tt$ （乘）

$\quad\mid /tt$ （除）

$\quad\mid t=t$ （比较）

其中 $i ::= i'$ （正整数）

$\quad\mid \sim i'$ （负整数）

$i' ::= i'0 \mid i'1 \mid \ldots \mid i'9 \mid \varepsilon$

def $v ::= \ldots$ （已有的值）

$\quad\mid i$ （整数）

下面，以加法为例定义单步运行关系。

def $\rightarrow$.（原有的规则省略）

$$\frac{t_1 \rightarrow t_1'}{+t_1t_2 \rightarrow +t_1't_2} \qquad \frac{t_2 \rightarrow t_2'}{+it_2 \rightarrow +it_2'}$$

$$\frac{ADD(i_1, i_2, i_3)}{+i_1i_2 \rightarrow i_3}$$

remark. 其中第三条规则还可以分正数、负数的情况讨论，这里就一篓子写到一起了。

继续以加法为例定义类型推导关系。作为习题，请你补充上比较操作的类型推导关系。

def $\vdash$ (原有规则省略)

$$\frac{}{\Gamma \vdash i : Int} \qquad \frac{\Gamma \vdash t_1 : Int \quad \Gamma \vdash t_2 : Int}{\Gamma \vdash (+\ t_1\ t_2) : Int}$$

至此，整数就作为一个独立的派别在系统中存在了。使用整数、计算整数都受到原生支持，且有类型系统的保护。

仿此，你可以往语言中加入各种实用的基本类型，比如小数、有理数、复数、字符串等等。与基本类型相对的，是所谓「复合类型」，比如我们前面认识的 ▶，以及接下来要谈的元组、记录等。复合类型完全是在基本类型上搭架子；若一个基本类型也没有，复合类型就成了无本之木。

复合类型的单步运行规则和类型推导规则往往比较复杂，值得我们单独考量。

2° 元组和记录

有时我们喜欢把一组相关的数据放在一起，构成一个整体。元组和记录提供了很好用的接口。先来看看目标效果：

- $\{1, 1, 2, 3, 5, 8, 13\}$ 声明了一个七元组（有序数组），它包含了七个整数。
- $\{\{True, False\}, \{0, 1, 2\}, 5, 7\}$ 声明了一个四元组，里面元素类型各异。
- $\{\lambda x{:}Int.\ x,\ False,\ 2\}.2$ 取出元组中的第2个元素，即 False.

记录是元组的加强版，给每个元素带上了文字标签：

- $\{name=\text{"Swallow"},\ age=21\}$ 声明了一条记录，它包含两个条目。
- $\{left=6,\ right=7,\ val=21\}.right$ 抽取出记录中对应元素，即 7.

以下是元组的实现方式：

def
$$t ::= \ldots \mid \{\underline{elements}\} \mid t.i \qquad \underline{elements} ::= t \mid t, \underline{elements}$$

$$v ::= \ldots \mid \{\underline{tupleVal}\} \qquad \underline{tupleVal} ::= v \mid v, \underline{tupleVal}$$

def $\rightarrow$.（原有规则略）

$$\frac{t_j \rightarrow t_j'}{\{v_1, \ldots, v_{j-1}, t_j, \ldots, t_n\} \rightarrow \{v_1, \ldots, v_{j-1}, t_j', \ldots, t_n\}}$$

$$(\forall n \in \mathbb{N},\ \forall j: 1 \le j \le n)$$

$$\frac{t \rightarrow t'}{t.i \rightarrow t'.i} \qquad \frac{}{\{v_1, \ldots, v_n\}.i \rightarrow v_i} (\forall n, \forall i: 1 \le i \le n)$$

def $\vdash$（原有规则略）

$$\boxed{T ::= \ldots \mid \underline{Tlist} \qquad \underline{Tlist} ::= T \mid T \times \underline{Tlist}}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \ldots \quad \Gamma \vdash t_n : T_n}{\Gamma \vdash \{t_1, \ldots, t_n\} : T_1 \times T_2 \times \cdots \times T_n} \quad (\forall n \in \mathbb{N})$$

$$\frac{\Gamma \vdash \{t_1, \ldots, t_n\} : T_1 \times \cdots \times T_n}{\Gamma \vdash \{t_1, \ldots, t_n\}.i : T_i} \quad (\forall n \in \mathbb{N},\ \forall i: 1 \le i \le n)$$

记录的实现也是类似的，除了需要引入「标签」以外，并无更多区别。因此我们将其留作习题。

remark. 如果我们先定义记录，再去考虑元组的话，会发现元组实为记录的「特例」—— 令标签 $=1, 2, \cdots$ 即退化为元组。或者说，我们可以借助记录去定义元组的行为。具体说来是这样的：

给定一个含元组的程序 t，我们可用某种算法 σ 将其转换为不含元组而只含记录的程序 $\sigma(t)$，并且保证

$$\begin{cases} t \rightarrow t' \iff \sigma(t) \rightarrow \sigma(t') \\ \Gamma \vdash t : T \iff \Gamma \vdash \sigma(t) : \sigma(T) \end{cases}$$

这里σ充当了同构映射，串连起了含元组与不含元组的系统。（σ的构造留作习题）σ也可被理解为「编译器」，把具有丰富特性的语言翻译成特性较少的语言，并保持二者在语义上的等同。

既然这个σ是存在的，所以，「元组」的概念不是必需的，而是为了方便而为之。这样子的构造俗称「语法糖」。把语法糖「编译」成更底层的构造，既有优点亦有缺点。优点是「不用想事儿」，层次性强，且许多结论不必重复证明；缺点是不够直接，效率可能要打折扣。

3° let...in... 命名语句

经常编程的人都有重用代码的习惯。若同一个功能在多处调用，则可以为该功能起个名字，以便调用时书写。比如，

我们希望计算 $\sin^3(x^3)$。用λ演算便要这么写：

```
λx:Float.
**sin(**xxx) sin(**xxx) sin(**xxx)
```

无疑不如

```
let Cube = λx.**xxx in
    λy:Float. Cube(sin(Cube y))
```

简洁和清晰。let...in...语句提供了初级的命名手段。

def $t ::= \ldots$

$\quad\mid$ let $x=t$ in t

def $\to$.（原有规则略）

$$\frac{t_1 \to t_1'}{\text{let } x=t_1 \text{ in } t_2 \to \text{let } x=t_1' \text{ in } t_2}$$

$$\frac{}{\text{let } x=v \text{ in } t \to t[x/v]}$$

def $\vdash$.（原有规则略）

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \cup \{x : T_1\} \vdash t_2 : T_2}{\Gamma \vdash (\mathrm{let}\ x = t_1\ \mathrm{in}\ t_2) : T_2}$$

remark. let…in…语句也是种语法糖，可编译成（λ…)(…)的样式。

4° 柔性

编程时，类型的条条框框常常过于死板，比如，函数只能接收规定类型的参数，又只能返回规定类型的值，不够灵活。考虑下面的两个例子：

e.g.1 欲实现 max(x, y) 函数，使之既能处理整数，又能处理浮点数。在诸如Pascal这样的「古典」语言中，唯一的出路是定义一个整数版的 max1，一个浮点数版的 max2。这极其笨拙。

在C++和Java中，我们则可以利用重载机制来达成目标。C++甚至允许我们使用模板来泛化类型。

e.g.2. 欲实现一个栈。要求 pop() 函数返回栈顶元素；若栈已空，则返回一个特殊的错误信息。在C中，我们通常用 -1 来代指错误，可是，这仍然容易造成混淆和故障。有没有办法把错误信息与数据在类型上区分开来？—— 想实现之，就要求函数的返回类型不只一种。

柔性类型应运而生。简而言之，它把若干类型 $T_1, T_2, \cdots, T_n$「加」在一起，对象可任取其一；但无论具体取了哪个，对象的类型都统一唤作 T。

用法示例：

- let (arg = 98 : Int as Int+Float) in …

命名了一个叫「arg」的量，类型为Int+Float，即「从Int与Float中任取一个」。这里由于

98是整数，所以取了Int。但它的类型名仍然是Int+Float.

- let (arg2= $98.7^{:\mathrm{Float}}$ as Int+Float) in ...

  命名了一个叫「arg2」的量，类型为Int+Float。由于98.7是浮点数，故取了Float。无论如何，对外仍声称类型为Int+Float。

- λx: Int+Float.

```
switch x
  case y:Int then (- y 1) as Int+Float
| case y:Float then (- y 1.0) as Int+Float
```

定义了一个适用于Int与Float的「减1」函数。Switch-Case语句将根据x的内在类型作分支。比如，将arg传入，则进入分支一；将arg2传入，则进入分支二。

函数返回值仍为Int+Float型。

def
$$
\begin{aligned}
t ::= \; & \dots \\
\mid \; & t{:}T \text{ as } T \\
\mid \; & \text{switch } t \ \underline{\text{CaseList}}
\end{aligned}
$$

$$
\begin{aligned}
\underline{\text{CaseList}} ::= \; & \text{case } y{:}T \text{ then } t \\
\mid \; & \text{case } y{:}T \text{ then } t \mid \text{CaseList}
\end{aligned}
$$

def
$$
\begin{aligned}
T ::= \; & \dots \\
\mid \; & \underline{Tlist2}
\end{aligned}
\qquad
\begin{aligned}
\underline{Tlist2} ::= \; & T \\
\mid \; & T + \underline{Tlist2}
\end{aligned}
$$

def
$$
\begin{aligned}
v ::= \; & \dots \\
\mid \; & v{:}T \text{ as } T
\end{aligned}
$$

def →. (原有规则略)

$$
\frac{t \to t'}{(t{:}T_1 \text{ as } T_2) \to (t'{:}T_1 \text{ as } T_2)}
$$

$$
\frac{t \to t'}{\begin{array}{l}\text{switch } t \\ \quad \text{case } y_1{:}T_1 \text{ then } t_1 \\ \quad \vdots \\ \quad \mid \text{case } y_n{:}T_n \text{ then } t_n\end{array} \to \begin{array}{l}\text{switch } t' \\ \quad \text{case } y_1{:}T_1 \text{ then } t_1 \\ \quad \vdots \\ \quad \mid \text{case } y_n{:}T_n \text{ then } t_n\end{array}}
$$

$$
\frac{}{\begin{array}{l}\text{switch } v{:}T_i \text{ as } T \\ \quad \text{case } y_1{:}T_1 \text{ then } t_1 \\ \quad \vdots \\ \quad \mid \text{case } y_n{:}T_n \text{ then } t_n\end{array} \to t_i[y_i/v]}
$$

def $\vdash$ (原有规则略)

$$\frac{\Gamma \vdash t : T_i}{\Gamma \vdash (t : T_i \text{ as } T_1 + \cdots + T_n) : T_1 + T_2 + \cdots + T_n} \quad (\forall n \in \mathbb{N},\ \forall i : 1 \le i \le n)$$

$$\frac{\Gamma \cup \{y_1 : T_1\} \vdash t_1 : T \quad \cdots \quad \Gamma \cup \{y_n : T_n\} \vdash t_n : T \qquad \Gamma \vdash t : T_1 + T_2 + \cdots + T_n}{\Gamma \vdash \begin{pmatrix} \text{switch } t \\ \quad \text{case } y_1 : T_1 \text{ then } t_1 \\ \quad \vdots \\ \quad | \text{case } y_n : T_n \text{ then } t_n \end{pmatrix} : T}$$

remark. 第二条规则与 if-then-else 很相似，它要求：无论条件取成什么，结果类型们都清一色为 T。这是为了类型安全所作的牺牲。

另外，其实「$t : T_1$ as T_2」的写法中，「$: T_1$」可以被省略——因为类型系统有能力推导之。可是，那样的话，→ 与 ⊢ 的定义便耦合在一起，比较复杂。

接下来，我们以一个简短的例子看看柔性类型的应用。

e.g. 设计一张能存储某个 $f : \mathbb{Z} \to \mathbb{Z}$ 的表。初始时，映射为空。用户可不断往映射 f 中添加项目，比如添加映射 $23 \mapsto 74$；也可访问所需项目。

假定 Error 是已定义好的基本类型。error 是其实例。

EmptyTable := $\lambda n : \text{Int}.\ \text{error as } (\text{Int} + \text{Error})$

(初始时，映射为空，因此无论传入什么 n，皆返回 Error)

ExtendTable := $\lambda t : \text{Int} \to (\text{Int} + \text{Error}).$
$\quad \lambda m : \text{Int}.\ \lambda v : \text{Int}.$
$\quad\quad \lambda n : \text{Int}.$
$\quad\quad\quad \text{if } n = m \text{ then } v \text{ as } (\text{Int} + \text{Error})$
$\quad\quad\quad \text{else } t\ n$

5° 列表

C/C++ 中的数组和 Python 中的列表，都是非常灵活的工具；几乎没有什么实用程序能脱离列表的概念。列表与元组和记录的最大不同在于：列表支持「动态」的增删，因而可以方便地实现插入、合并、拆分等高级操作。

使用示例：

- $\mathbf{nil}^{\,as\,T}$ 定义了一个空列表，类型为 $T\ list$.
- $\mathbf{let}\ l{:}Int\ list = 5::8::11::\mathbf{nil}\ as\ Int\ \ in$

  …… ↳ 严格写来，应为 $(5::(8::(11::\mathbf{nil}\ as\ Int)))$

  定义了一个名为 l 的列表，内部元素依次为 $5,8,11$，类型为 $Int\ list$.
- $\lambda l{:}Int\ list.$

  $\quad switch\ l$

  $\quad\quad case\ nil\ then\ 0$

  $\quad\quad |\ case\ x::y\ then\ x$

  定义了一个函数返回列表首元素（若列表为空则返回 0）。

$$\mathbf{def}\quad t ::= \ldots$$

$\quad |\ \mathbf{nil}\ as\ T$ ⇝ 用以声明空列表

$\quad |\ (t::t)$ ⇝ 用以构造更长的列表

$\quad |\ switch\ t\ case\ nil\ then\ t$
$\quad\quad\quad |\ case\ x::x\ then\ t$ ⇝ 用以区分空与非空列表

$$\mathbf{def}\quad T ::= \ldots \mid T\ list$$

$$\mathbf{def}\quad v ::= \ldots \mid \mathbf{nil}\ as\ T \mid (v::v)$$

$$\mathbf{def}\ \rightarrow.$$

$$\frac{t_i \rightarrow t_i'}{v_1::\cdots::v_{i-1}::t_i::\cdots::t_n \rightarrow v_1::\cdots::v_{i-1}::t_i'::\cdots::t_n}$$

$$\frac{t \rightarrow t'}{\begin{array}{l} switch\ t\ case\ nil\ then\ t_1 \\ \quad |\ case\ x::y\ then\ t_2 \end{array} \rightarrow \begin{array}{l} switch\ t'\ case\ nil\ then\ t_1 \\ \quad |\ case\ x::y\ then\ t_2 \end{array}}$$

$$\frac{}{\begin{array}{l} switch\ (nil\ as\ T)\ case\ nil\ then\ t_1 \\ \quad |\ case\ x::y\ then\ t_2 \end{array} \rightarrow t_1}$$

$$\frac{}{\begin{array}{l} switch\ (v_1::v_2)\ case\ nil\ then\ t_1 \\ \quad |\ case\ x::y\ then\ t_2 \end{array} \rightarrow t_2[x/v_1][y/v_2]}$$

def $\vdash$.

$$\frac{}{\Gamma \vdash (\text{nil as } T) : T\ list} \qquad \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T\ list}{\Gamma \vdash (t_1 :: t_2) : T\ list}$$

$$\frac{\Gamma \vdash t : T\ list \quad \Gamma \vdash t_1 : T^* \quad \Gamma \cup \{x : T, y : T\ list\} \vdash t_2 : T^*}{\Gamma \vdash \left(\begin{array}{l}\text{switch } t \text{ case nil then } t_1 \\ \quad\quad\ \mid \text{case } x :: y \text{ then } t_2\end{array}\right) : T^*}$$

6° 递归

上一章末尾才说过：递归函数在当时的系统下是没有类型的，因为不存在 T_1 与 T_2 使得 $T_1 \to T_2$ 为其类型。现在，我们却试图把递归引入系统中来使之具备类型，这怎么可能呢？

事实上，我们是在做一场交易：强行把「递归」引入系统之中，但牺牲掉系统的一部分安全性。具体地说即：上一章的 Theorem 8 (Progress) 和 Theorem 7 (Preservation) 均成立，可是程序之终止性得不到保障，因而 Theorem 9 (Termination) 不再成立。用一条性质的破坏去换取图灵完全的表达能力，不仅值得，而且必要。

为了更好地理解下面的定义，我们先来回顾一下，在纯粹的 λ 演算中如何实现递归。当时，我们希望编写阶乘函数 Fact，由于无法在函数内部引用自己，所以我们得借助别人，将 Fact 作为参数传递到自己之中，帮助记忆。即：

$$Helper := \lambda f.\lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n \times (f\ f\ (n-1))$$

$$Fact := Helper\ Helper\ n$$

Helper 是功能的实现者，而 Fact 则将其包裹住，助 Helper 回忆起自己是谁。

现在，我们欲在语言层面支持递归。循序渐进，先从简易版说起。在简易版中，

用户这么定义阶乘函数：

let

$$g = \lambda f{:}\mathrm{Int}\to\mathrm{Int}.\ \lambda n{:}\mathrm{Int}.\ \text{if } n=0 \text{ then } 1 \text{ else } n \times f(n-1)$$

in

$$\textbf{fix}\ g$$

注意这里的 g 与 Helper 几乎一样，除了在「递归」时写的是 $n \times f(n-1)$ 而不是 $n \times (f\ f\ (n-1))$。这么简化是为了顺手——用户只需把 f 当作欲定义的函数之别名来使用即可。

程序中的 fix 关键词是系统实现递归功能的核心。概括地说，它会在运行的时候，把 g 中的 f「替代成「g 自己」从而令 g 真真正正地成为递归函数。

def $t ::= \dots \mid \textbf{fix}\ t$

def $\to$.

① $$\frac{t \to t'}{\text{fix}\ t \to \text{fix}\ t'}$$

② $$\frac{}{\text{fix}\ (\lambda f{:}T.\,t) \to t[f/\text{fix}(\lambda f{:}T.\,t)]}$$

remark. 第二条规则值得琢磨。为何不一劳永逸地把 f 替换成 $(\lambda f{:}T.\,t)$ 呢？这样不就实现了自引用了吗？非也！请注意 $(\lambda f{:}T.\,t)$ 自己要接收函数作为参数，而用户使用 f 时却不会提供这一参数，若用 $(\lambda f{:}T.\,t)$ 替 f，则参数少一个，结果不正确。（请参见左边阶乘函数的例子，若把 g 中的 f 用 $(\lambda f{:}\mathrm{Int}\to\mathrm{Int}.\ \lambda n{:}\mathrm{Int}.\ \dots)$ 代替，会有何后果？）

换句话说，fix 关键词实是「按需展开」且「永远保持展开能力」，以便 g 在任何层次均能回忆起自己是谁。

e.g. 阶乘例子运行示例

$(\mathrm{fix}\ g)\ 5$

$\to (\lambda n{:}\mathrm{Int}.\ \text{if } n=0 \text{ then } 1 \text{ else } n\times((\mathrm{fix}\ g)(n-1)))\ 5$

此处的 fix 将为未来调用提供支撑

$\to \text{if } 5=0 \text{ then } 1 \text{ else } 5\times((\mathrm{fix}\ g)\ 4)$

$\to 5\times((\mathrm{fix}\ g)\ 4)$

$\to 5\times[(\lambda n{:}\mathrm{Int}.\ \text{if } n=0 \text{ then } 1 \text{ else } n\times((\mathrm{fix}\ g)(n-1)))\ 4]$

$\to 5\times[\text{if } 4=0 \text{ then } 1 \text{ else } 4\times((\mathrm{fix}\ g)\ 3)]$

$\to 5\times[4\times((\mathrm{fix}\ g)\ 3)]$

$\to \cdots$

下面来分析一下类型推导关系。我们留意到，$g:(\mathrm{Int}\to\mathrm{Int})\to(\mathrm{Int}\to\mathrm{Int})$ 是有类型的，而且长得十分规则。不妨再试试别的递归函数——比如如下判断 n 是否被3整除的函数：

```
let h = λf:Int→Bool. λn:Int.
        if n=0 then True
        else if n=1 then False
        else if n=2 then False
        else if n<0 then f(n+3)
        else f(n-3)
in fix h
```

不难分析出 $h:(\mathrm{Int}\to\mathrm{Bool})\to(\mathrm{Int}\to\mathrm{Bool})$。看来，这其中确有某种规律。

站在用户角度，易解释该规律：用户为了定义递归函数 r^*，首先得定义辅助函数 r（比如前文的 g 与 h），然后调用 $\mathrm{fix}\ r$。而辅助函数 r 的格式总是：接收参数 f 并将其等同于 r^* 来调用。因而，f 的类型总应与 r^* 的类型一致。接收完 f 以后，接下来就是遵循 r^* 之逻辑书写函数了，因而余下部分之类型也应与 r^* 的类型一致。

综合而言，在任何正常、正确的使用场合，r的类型都必须为 $T \blacktriangleright T$ 的形式，且$r^*:T$。既然如此，类型系统就应当给 fix r 赋以类型 T ——毕竟，单步运行规则已经迫使 fix r 之行为与 r^* 无异。

所以，我们有如下的类型推断规则：

def $\vdash$.
$$\frac{\Gamma \vdash t : T \blacktriangleright T}{\Gamma \vdash (\mathrm{fix}\ t) : T}$$

当然，以上论证是从直观出发的，并不严格。我们在下面的两个补丁中论述fix的引入不影响上一章的Theorem 7, 8。

Patch for Theorem 7

引入fix以后Theorem 7 (Preservation) 仍然成立。

proof. 只需添加对上面单步运行规则之证明。

CASE① 已知 $t \to t'$，$\mathrm{fix}\ t \to \mathrm{fix}\ t'$，以及 $\Gamma \vdash \mathrm{fix}\ t : T$。那么 $\Gamma \vdash t : T \blacktriangleright T$。由I.H.知 $\Gamma \vdash t' : T \blacktriangleright T$，故 $\Gamma \vdash \mathrm{fix}\ t' : T$

CASE② 已知 $\mathrm{fix}(\lambda f{:}T.t) \to t[f/\mathrm{fix}(\lambda f{:}T.t)]$ 及 $\Gamma \vdash \mathrm{fix}(\lambda f{:}T.t) : T_0$。故 $\Gamma \vdash (\lambda f{:}T.t) : T_0 \blacktriangleright T_0$，从而 $T = T_0$。易知 $t[f/\mathrm{fix}(\lambda f{:}T.t)]$ 亦有类型 T_0 ■

Patch for Theorem 8

引入fix以后Theorem 8 (Progress) 仍然成立。

proof. 若 $\Gamma \vdash (\mathrm{fix}\ t) : T$ 则必有 $\Gamma \vdash t : T \blacktriangleright T$，从而 $t = \lambda f{:}T.t'$。于是 $\mathrm{fix}\ t \to t'[f/\mathrm{fix}(\lambda f{:}T.t')]$。■

remark. 仔细想来，上一章的引理（尤其是Lemma 5, 6）也需打补丁。请你完成之。

在简易版的基础上，我们来搭建一款关于递归的语法糖。

用法：

recurse

$\quad fact : Int \to Int = \lambda n : Int.$

$\qquad$ if $n=0$ then 1 else $n \times fact(n-1)$

in

$\quad fact\ 7$

直接定义了递归函数fact。这可被翻译成

let

$\quad g = \lambda f : Int \to Int.\ \lambda n : Int.$

$\qquad$ if $n=0$ then 1 else $n \times f(n-1)$

in let $fact = fix\ g$ in

$\quad fact\ 7$

一般地，

recurse $h : T = t$ in t'

被翻译成

let $g = \lambda f : T.\ t[h/f]$

$\quad$ let $h = fix\ g$ in t'

而recurse的类型推导规则为

$$\frac{\Gamma \cup \{h:T\} \vdash t:T \qquad \Gamma \cup \{h:T\} \vdash t':T'}{\Gamma \vdash (\text{recurse } h:T=t \text{ in } t'):T'}$$

附录：提升运行效率

截至目前，我们的程序是遵照「先左后右，然后代入」的原则执行的，单步运行规则由判断形式 $t \to t'$ 定义。整条执行流程呈现 $t \to t' \to t'' \to \cdots$ 的链式结构，或者说，相当于程序本身的不断变形。

但这样子的「运行」效率不高。当我们遇到 $(\lambda x.t_1)t_2$ 这种形式时，需要执行替换操作，得到 $t_1[x/t_2]$；然而，替换操作说来简单，做来麻烦——得要寻出 t_1 之中所有自由的 x 作替换。这需花费至少线性的时间（用以遍历 t_1 的语法树），更别提空间上的「爆炸」了。想想一个大型程序，每一步执行的时间都正比于其尺寸，这等低效是难以容忍的。

于是，人们提出了更高效的做法：把所需做的替换积攒到一张表里，不到迫不得已，不做真实的替换。类似的「惰性」思想在诸如线段树一类的数据结构中时有出现。

先用一个小例子说明大致思路。（本附录中仅考虑纯粹的λ演算；扩展不难）

$$(\lambda f.\lambda x.f(x+1))(\lambda x.x)\,5$$

下一步该用 $(\lambda x.x)$ 去取代 f 了。可是，我们不真的去取代，而是先把这操作记录到表格里：

#	mapping
1	$f \mapsto \lambda x.x$

程序现在剩下 $(\lambda x.f(x+1))5$，该把 5 代入左侧。但我们仍不动手，而是记笔帐：

#	mapping
1	$f \mapsto \lambda x.x$
2	$x \mapsto 5$

现在，只剩下 $f(x+1)3$。此时，由于f是符号，不是值，所以得去表中取值，得到$(\lambda x.x)$。同理，x也是符号，得去表中取值，得到5。双方准备齐全后，把6代入$(\lambda x.x)$：

#	mapping	
1	$f \mapsto \lambda x.x$	
2	$x \mapsto 6$	(renewed)

得到x。最后，查询表格，得结果为6。

或许你已留意到，该方法有个好处：无须进行变元易名。这是因为，「不到迫不得已决不执行替换」的策略，迫使干坏事的λ在替换之时早已不复存在，于是，不可能有变元被λ捉住而改变语义。

exercise 请你尝试用上述方法运行 $(\lambda x.\lambda y.x+y)\ y\ 5$，看看其间奥秘。

不过，事情不只「记帐」这么简单。试考虑下面这个较复杂的例子：

$$(\lambda x.(\lambda f.(\lambda x.f\ x)4)(\lambda y.x+y))\ 3$$

[相当于let $x=3$ in
let $f=\lambda y.x+y$ in
let $x=4$ in $f\ x$]

答案应为$4+3=7$。如果用纯粹的记帐法，则有

#	mapping
1	$x \mapsto 3$

$\Rightarrow$

#	mapping
1	$x \mapsto 3$
2	$f \mapsto \lambda y.x+y$

$\Rightarrow$

#	mapping
1	$x \mapsto 4$
2	$f \mapsto \lambda y.x+y$

最后计算$f\ x$，

$\Rightarrow$

#	mapping
1	$x \mapsto 4$
2	$f \mapsto \lambda y.x+y$
3	$y \mapsto 4$

$\Rightarrow x+y=8$.

得到错误的结果。究其原因，是由于let $x=4$ in … 的作用域没处理好，「溢出到f里面去了。怎样能保证作用域不

溢出呢？解决方案是：在给函数 f 记帐时拍摄一张「快照」，把当时的映射表保留下来：

#	mapping
1	$x \mapsto 3$

$\Rightarrow$

#	mapping
1	$x \mapsto 3$
2	$f \mapsto \{\lambda y.x+y, \{x \mapsto 3\}\}$

这样了，无论后面 x 遭受怎样的修改，均不会影响 f —— f 已经和原来的情形绑定在一起了。

$\Rightarrow$

#	mapping
1	$x \mapsto 4$
2	$f \mapsto \{\lambda y.x+y, \{x \mapsto 3\}\}$

现在，准备计算 $f\ x$。

f 方面，我们从映射表中取得函数体 $\lambda y.x+y$，以及从前拍摄的快照 $\{x \mapsto 3\}$.

x 方面，我们从映射表中取得值 4。

将二者合并，先把映射表恢复到原始情形

#	mapping
1	$x \mapsto 3$

，再把 4 代入 $\lambda y.x+y$，得映射表

#	mapping
1	$x \mapsto 3$
2	$y \mapsto 4$

和结果 $x+y$，最终得 7。

把「快照」的观念运用进来，问题就被完满解决了。以下算法正实现了该目标：

```
Execute(M, t)
  // M是当前的映射表，t是程序
  switch(t)
    case x: return M(x)
    case 常数: return 该常数
    case λx.t0: return {λx.t0, M}
    case t1 t2:
      v1 := Execute(M, t1)
      v2 := Execute(M, t2)
      if v1 ≠ {λx.t0, M0} then      M0 → 函数之运行环境，不同于M
        error
      else
        {λx.t0, M0} := v1
        M0.SetMapping(x ↦ v2)
        Execute(M0, t0)             M0 → 切换为函数的运行环境
```

该方法可用判断形式来严格定义.

def $t \text{ with } M \to t' \text{ with } M'$.

$$\frac{}{x \text{ with } M \to M(x) \text{ with } M}$$

$$\frac{}{\lambda x.t \text{ with } M \to \{\lambda x.t, M\} \text{ with } M}$$

$$\frac{t_1 \text{ with } M \to t_1' \text{ with } M'}{t_1 t_2 \text{ with } M \to t_1' t_2 \text{ with } M}$$

$$\frac{t_2 \text{ with } M \to t_2' \text{ with } M'}{\{\lambda x.t_0, M_0\} t_2 \text{ with } M \to \{\lambda x.t_0, M_0\} t_2' \text{ with } M}$$

$$\frac{}{\{\lambda x.t_0, M_0\} v_2 \text{ with } M \to t_0 \text{ with } M_0[x \mapsto v_2]}$$

remark. 这里提供的是单步运行的判断形式，Execute则是从头到尾运行的算法。二者稍有差别。

下面给出 $(\lambda x.(\lambda f.(\lambda x.fx)4)(\lambda y.x+y))3$ 的完整单步运行流程。

$(\lambda x.(\lambda f.(\lambda x.fx)4)(\lambda y.x+y))3 \text{ with } \emptyset$

$\to \{\lambda x.(\lambda f.(\lambda x.fx)4)(\lambda y.x+y), \emptyset\} 3 \text{ with } \emptyset$

$\to (\lambda f.(\lambda x.fx)4)(\lambda y.x+y) \text{ with } \{x \mapsto 3\}$

$\to \{\lambda f.(\lambda x.fx)4, \{x \mapsto 3\}\}(\lambda y.x+y) \text{ with } \{x \mapsto 3\}$

$\to \{\lambda f.(\lambda x.fx)4, \{x \mapsto 3\}\}\{\lambda y.x+y, \{x \mapsto 3\}\} \text{ with } \{x \mapsto 3\}$

$\to (\lambda x.fx)4 \text{ with } \{x \mapsto 3, f \mapsto C_f\}$

$(C_f := \{\lambda y.x+y, \{x \mapsto 3\}\})$

$\to \{\lambda x.fx, \{x \mapsto 3, f \mapsto C_f\}\} 4 \text{ with } \{x \mapsto 3, f \mapsto C_f\}$

$\to fx \text{ with } \{x \mapsto 4, f \mapsto C_f\}$

$\to C_f\, x \text{ with } \{x \mapsto 4, f \mapsto C_f\}$

$\to C_f\, 4 \text{ with } \{x \mapsto 4, f \mapsto C_f\}$

$= \{\lambda y.x+y, \{x \mapsto 3\}\} 4 \text{ with } \{x \mapsto 4, f \mapsto C_f\}$

$\to x+y \text{ with } \{x \mapsto 3, y \mapsto 4\}$

$\to 3+y \text{ with } \{x \mapsto 3, y \mapsto 4\}$

$\to 3+4 \text{ with } \{x \mapsto 3, y \mapsto 4\} \to 7 \text{ with } \{x \mapsto 3, y \mapsto 4\}$