北京大学计算机学院

2025 年春季学期　　《编译原理(实验班)》

第十四讲　类型系统

Type Systems

主要内容

- 类型、类型系统
- 类型系统的可靠性
- 多态类型
- (代数)数据类型
- Rust 的类型系统

主要内容

- **类型、类型系统**
- 类型系统的可靠性
- 多态类型
- (代数)数据类型
- Rust 的类型系统

什么是类型？

- 类型是对程序行为的一种**简洁**、**准确**的**形式化**描述

ML `int`

C++ `int`

整数

ML `int -> bool`

C++ `function<bool(int)>`

把整数参数映射为布尔结果的函数

ML `(int -> bool) -> (int list -> int list)`

C++ `function< function<list<int>(list<int>)> ( function<bool(int)> ) >`

把整数谓词映射为整数列表变换的函数

静态且可靠的类型检查

- 类型需要是**可靠的(sound)**：程序行为必须符合其类型
  - 一个具有 int 类型的表达式的计算结果必须是一个整数值
- 对程序的**类型检查(type checking)**需要拒绝**非良类型的程序(ill-typed programs)**
- 编译器在编译过程中**静态(static)**进行类型检查
- **静态类型对于编程语言来说有什么优点和缺点？**

（静态）类型的优点

- 「Well-typed expressions do not go wrong」
  - Milner, A Theory of Type Polymorphism in Programming, 1978.
  - 类型可以作为**经过形式化检查的程序文档**
  - 类型提供了程序的**安全性保障**
- 「Type structure is a syntactic discipline for enforcing levels of abstraction」
  - Reynolds, Types, Abstraction and Parametric Polymorphism, 1983.
  - 类型提供了**模块化和抽象机制**，从而支持**分离编译**并提高了**健壮性**
- 类型还可以助力编译优化，提升程序**性能**

低级语言也可以类型化

- 系统级编程语言 C、C++、Rust 等都是静态类型语言
- 甚至**汇编语言**也可以类型化
  - Morrisett et al., From System F to Typed Assembly Language, 1999.
- 在**类型保持(type-preserving)**编译器中，所有的中间语言也都是类型化的，并且编译的变换过程**保持良类型性**
  - Chlipala, A certified type-preserving compiler from lambda calculus to assembly language, 2007.
  - 保持类型有助于理解编译过程中的程序变换
  - 保持类型也有助于形式化证明编译的正确性

（静态）类型的缺点

- 类型是程序行为的描述，所以标注类型可能**引入冗余**，给程序员带来负担
  - 很多现代语言都实现了一定程度的**类型推导**(type inference)
- 类型一定程序上**限制了语言的「表达能力」**
  - 这里的「表达能力」主要体现在编写程序的灵活性上
  - **可靠、可判定的类型系统一定会拒绝一些实际上安全的程序**

- **编程语言究竟要不要引入类型呢？**

Reynolds 的锐评

- 「One side claims that **untyped languages** preclude compile-time error checking and are **succinct to the point of unintelligibility**, while the other side claims that **typed languages** preclude a variety of powerful programming techniques and are **verbose to the point of unintelligibility**」
  - Reynolds, Three Approaches to Type Structure, 1985.
- 无类型的语言无法进行编译时检查，**简练到难以理解的程度**
- 有类型的语言排除了一些编程技巧，**冗长到难以理解的程度**

Reynolds 的锐评

- 「From the theorist's point of view, **both sides are right**, and their arguments are the motivation for seeking type systems that are **more flexible and succinct** than those of existing typed languages」

- **无(静态)类型语言的角度:**
  - Python 支持了类型标注，通过外部工具支持静态检查
  - JavaScript 衍生出了 TypeScript，引入了灵活、丰富的类型标注

- **有(静态)类型语言的角度:**
  - C++、Go 等语言支持了类型推导，不需要标注所有类型
  - Rust 通过在类型系统中追踪所有权等信息，提供了更强的安全保障

主要内容

- 类型、类型系统
- **类型系统的可靠性**
- 多态类型
- (代数)数据类型
- Rust 的类型系统

回顾：函数式编程语言 micro-ML

$$Exp ::= \text{const}(n) \mid \text{op}(Exp_1, Exp_2) \mid \text{if}(Exp_1, Exp_2, Exp_3) \mid \text{var}(x) \mid \text{let}(x, Exp_1, Exp_2)$$
$$\mid \text{letfun}(f, x, Exp_1, Exp_2) \mid \text{call}(Exp_1, Exp_2)$$

- 用 $\rho \vdash E \Rightarrow v$ 表示**三元关系** (ρ, E, v)，表示「在环境 ρ 下对表达式 E 进行求值的结果是 v」

横线上方表示「前提」

$$\frac{}{\rho \vdash \text{const}(n) \Rightarrow n} \qquad \frac{\rho \vdash E_1 \Rightarrow v_1 \quad \rho \vdash E_2 \Rightarrow v_2 \quad v = f_{\text{op}}(v_1, v_2)}{\rho \vdash \text{op}(E_1, E_2) \Rightarrow v}$$

横线下方表示「结论」

$$\frac{\rho \vdash E_1 \Rightarrow \text{true} \quad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{if}(E_1, E_2, E_3) \Rightarrow v} \qquad \frac{\rho \vdash E_1 \Rightarrow \text{false} \quad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \text{if}(E_1, E_2, E_3) \Rightarrow v} \qquad \frac{\rho(x) = v}{\rho \vdash \text{var}(x) \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{let}(x, E_1, E_2) \Rightarrow v}$$

回顾：函数式编程语言 micro-ML

$$Exp ::= \text{const}(n) \mid \text{op}(Exp_1, Exp_2) \mid \text{if}(Exp_1, Exp_2, Exp_3) \mid \text{var}(x) \mid \text{let}(x, Exp_1, Exp_2)$$
$$\mid \text{letfun}(f, x, Exp_1, Exp_2) \mid \text{call}(Exp_1, Exp_2)$$

- 用 $\rho \vdash E \Rightarrow v$ 表示**三元关系** (ρ, E, v)，表示「在环境 ρ 下对表达式 E 进行求值的结果是 v」

「闭包」：在记录函数定义时，同时记录当时的环境

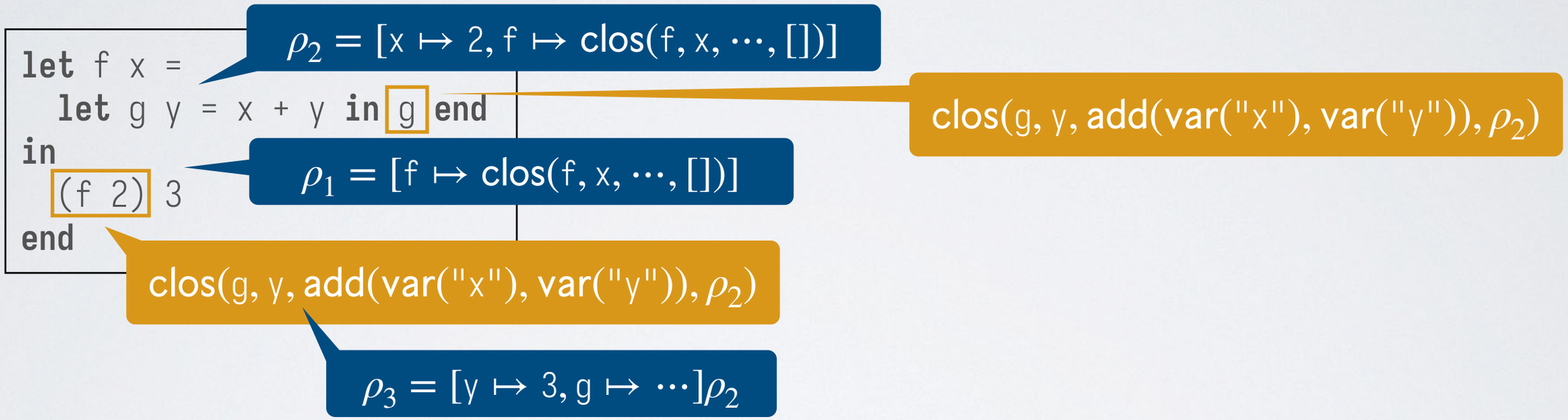
$$\frac{[f \mapsto \text{clos}(f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{letfun}(f, x, E_1, E_2) \Rightarrow v}$$

被调用者的求值结果是一个「闭包」

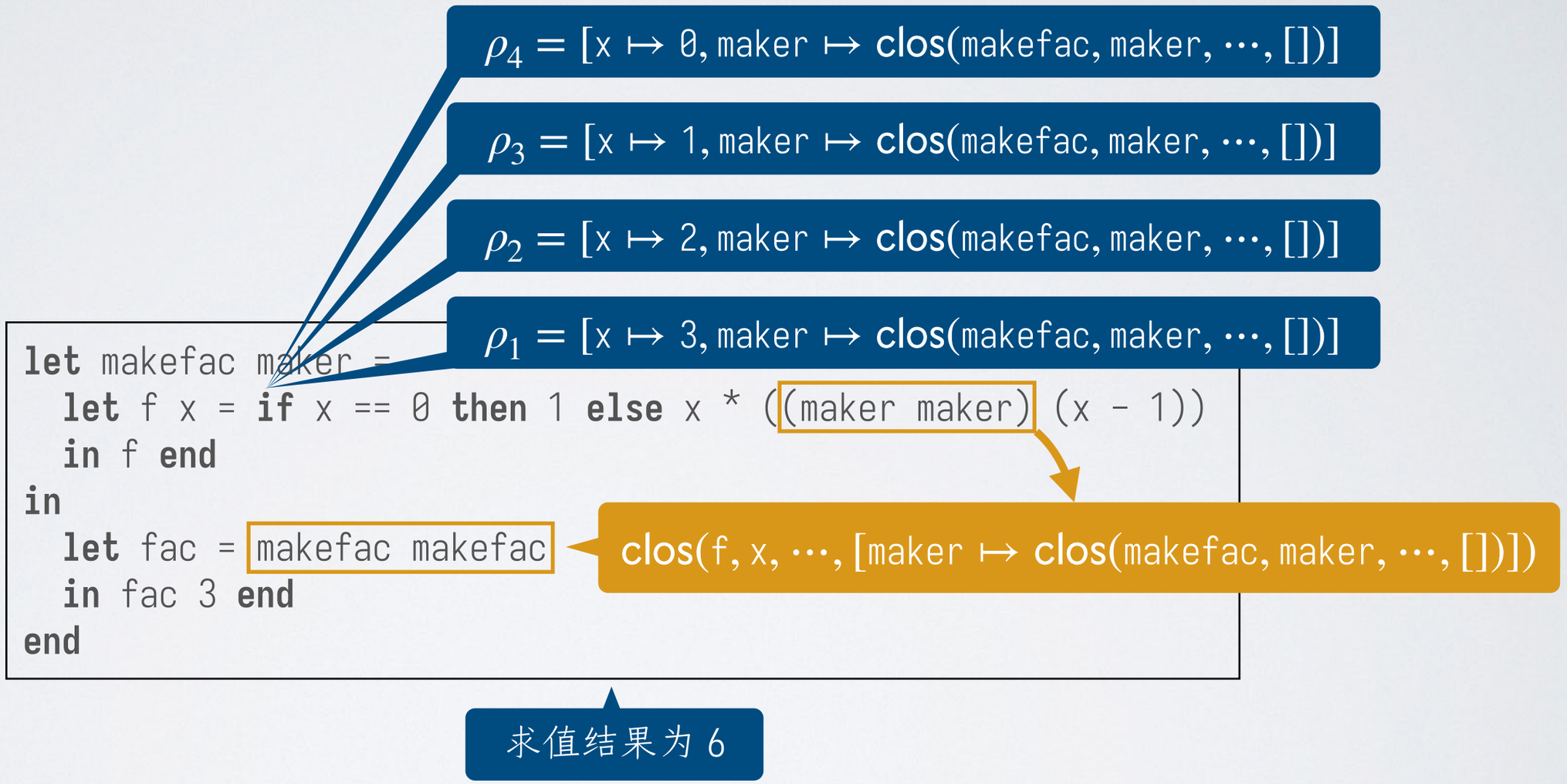
在「闭包」中记录的环境完成函数调用

$$\frac{\rho \vdash E_1 \Rightarrow \text{clos}(f', x', E_1', \rho') \quad \rho \vdash E_2 \Rightarrow v_2 \quad [x' \mapsto v_2, f' \mapsto \text{clos}(f', x', E_1', \rho')]\rho' \vdash E_1' \Rightarrow v}{\rho \vdash \text{call}(E_1, E_2) \Rightarrow v}$$

回顾：函数式编程语言 micro-ML



回顾：函数式编程语言 micro-ML



Typed micro-ML

- 我们先从最简单的类型开始，只考虑**整数**、**布尔**和**函数**类型：

$$Type ::= \text{int} \mid \text{bool} \mid Type_1 \to Type_2$$

- 用 $\Gamma \vdash E : T$ 表示**三元关系** (Γ, E, T)，表示「在类型上下文 Γ 下表达式 E 求值结果的类型是 T」

- **类型上下文** Γ 记录变量到类型的绑定：

$$\Gamma ::= \varnothing \mid \Gamma; x : T$$

  - 表示的是 E 中可以使用的自由变量以及它们的类型

Typed micro-ML

$$Exp ::= \text{const}(n) \mid \text{op}(Exp_1, Exp_2) \mid \text{if}(Exp_1, Exp_2, Exp_3) \mid \text{var}(x) \mid \text{let}(x, Exp_1, Exp_2)$$
$$\mid \text{letfun}(f, x, Exp_1, Exp_2) \mid \text{call}(Exp_1, Exp_2)$$

- 用 $\Gamma \vdash E : T$ 表示**三元关系** (Γ, E, T)，表示「在类型上下文 Γ 下表达式 E 求值结果的类型是 T」

横线上方表示「前提」

$$\frac{}{\Gamma \vdash \text{const}(n) : \text{int}} \qquad \frac{\Gamma \vdash E_1 : T_1 \quad \Gamma \vdash E_2 : T_2 \quad T = \tau_{\text{op}}(T_1, T_2)}{\Gamma \vdash \text{op}(E_1, E_2) : T}$$

横线下方表示「结论」

要求两个分支具有相同的类型

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : T \quad \Gamma \vdash E_3 : T}{\Gamma \vdash \text{if}(E_1, E_2, E_3) : T} \qquad \frac{}{\Gamma \vdash \text{var}(x) : \Gamma(x)}$$

在上下文中记录变量的类型

$$\frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma; x : T_1 \vdash E_2 : T_2}{\Gamma \vdash \text{let}(x, E_1, E_2) : T_2}$$

Typed micro-ML

$$Exp ::= \text{const}(n) \mid \text{op}(Exp_1, Exp_2) \mid \text{if}(Exp_1, Exp_2, Exp_3) \mid \text{var}(x) \mid \text{let}(x, Exp_1, Exp_2)$$
$$\mid \text{letfun}(f, x, Exp_1, Exp_2) \mid \text{call}(Exp_1, Exp_2)$$

- 用 $\Gamma \vdash E : T$ 表示**三元关系** (Γ, E, T)，表示「在类型上下文 Γ 下表达式 E 求值结果的类型是 T」

可以使用 Γ 中的变量；这也是需要闭包的原因

函数的类型和参数的类型

函数体的类型就是返回类型

在 E_2 中可以使用函数 f

$$\frac{\Gamma; f : T_1 \to T_2; x : T_1 \vdash E_1 : T_2 \qquad \Gamma; f : T_1 \to T_2 \vdash E_2 : T}{\Gamma \vdash \text{letfun}(f, x, E_1, E_2) : T}$$

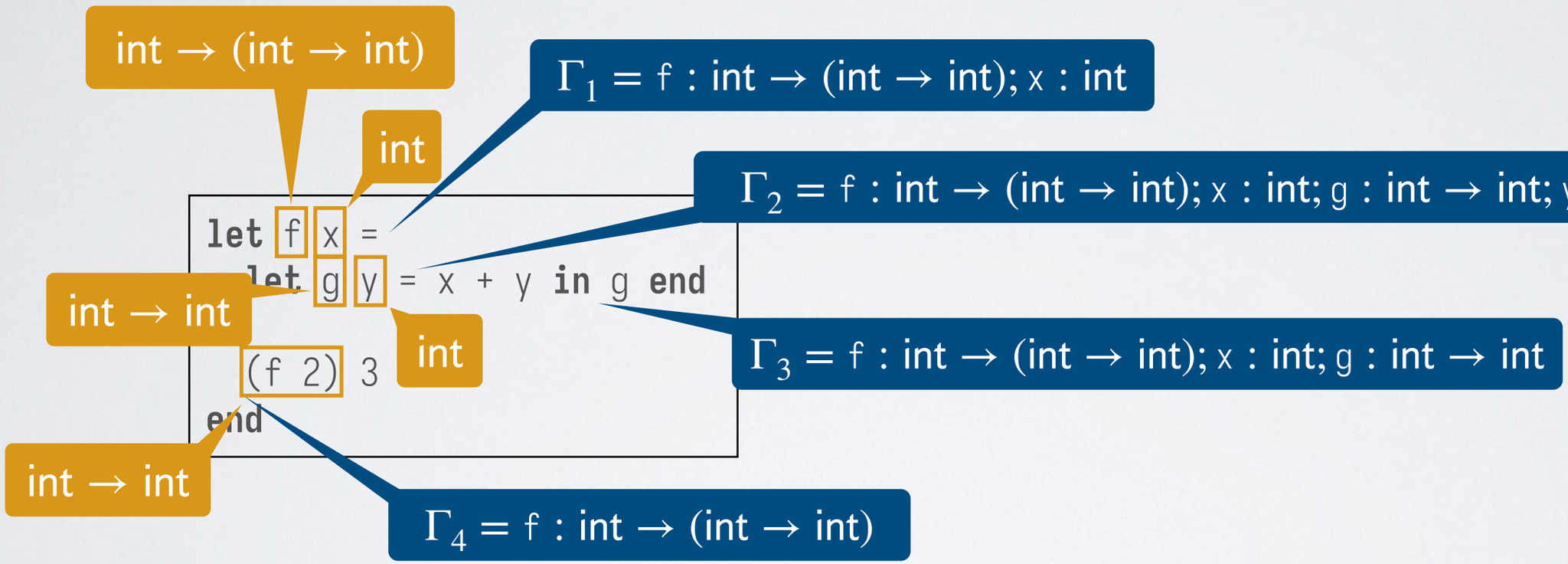
被调用函数的类型

参数的类型

$$\frac{\Gamma \vdash E_1 : T_1 \to T_2 \qquad \Gamma \vdash E_2 : T_1}{\Gamma \vdash \text{call}(E_1, E_2) : T_2}$$

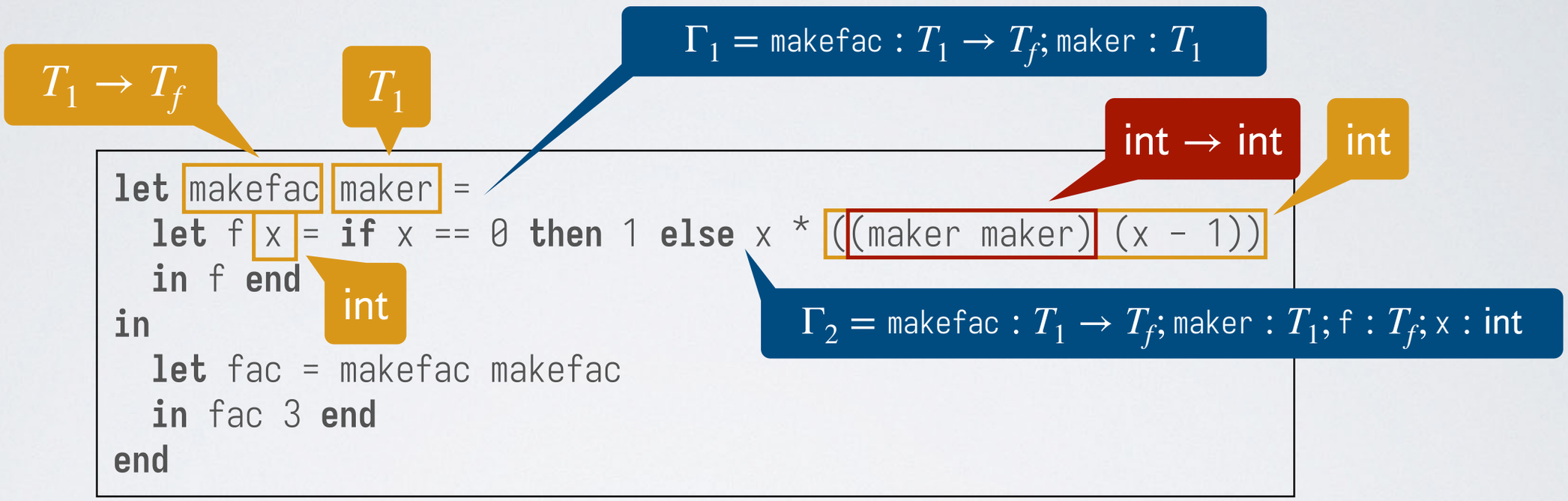
Typed micro-ML

$$\frac{\Gamma; f : T_1 \to T_2; x : T_1 \vdash E_1 : T_2 \qquad \Gamma; f : T_1 \to T_2 \vdash E_2 : T}{\Gamma \vdash \text{letfun}(f, x, E_1, E_2) : T}$$



$$\frac{\Gamma \vdash E_1 : T_1 \to T_2 \qquad \Gamma \vdash E_2 : T_1}{\Gamma \vdash \text{call}(E_1, E_2) : T_2}$$

Typed micro-ML



$$T_1 = T_1 \to (\text{int} \to \text{int})$$

- 简单的类型系统确实会限制语言的「表达能力」
  - 尽管如此，也可以很容易写出能够通过类型检查的阶乘函数

类型系统的可靠性

- 「Well-typed expressions do not go wrong」
  - Milner, A Theory of Type Polymorphism in Programming, 1978.
- **如何严格地(形式化)描述上面的性质?**
- 一个**良类型(well-typed)**的完整程序要么不停机,要么正常求出一个值;换言之,它不应该在求值过程中出错
- **类型可靠性定理:** 若 $\varnothing \vdash E : T$, 则
  - 要么存在一个 v, 使得 $[] \vdash E \Rightarrow v$ 且 v 具有类型 T
  - 要么 $[] \vdash E \Rightarrow^{\omega}$, 即对 E 的求值不停机(也不出错)
- **如何证明这个可靠性定理?**

类型系统的可靠性

- **弱化的版本**：若 $\varnothing \vdash E : T$ 且 $[] \vdash E \Rightarrow v$，则 v 具有类型 T
- **如何形式化定义「v 具有类型 T」？**

$$Val ::= n \mid \text{true} \mid \text{false} \mid \text{clos}(f, x, E, \rho)$$

$$\frac{}{n : \text{int}} \qquad \frac{}{\text{true} : \text{bool}} \qquad \frac{}{\text{false} : \text{bool}} \qquad \frac{\rho : \Gamma \qquad \Gamma, f : T_1 \to T_2, x : T_1 \vdash E : T_2}{\text{clos}(f, x, E, \rho) : T_1 \to T_2}$$

$$\frac{}{[] : \varnothing} \qquad \frac{\rho : \Gamma \qquad v : T}{[x \mapsto v]\rho : \Gamma; x : T}$$

- 若 $\varnothing \vdash E : T$ 且 $[] \vdash E \Rightarrow v$，则 $v : T$
  - **思路：归纳法**

类型系统的可靠性

- 在三元关系 $[] \vdash E \Rightarrow v$ 的**推导过程**上进行归纳

$$\frac{}{[] \vdash \text{const}(n) \Rightarrow n}$$

已知 $\dfrac{}{\varnothing \vdash \text{const}(n) : \text{int}}$ 从而 $n : \text{int}$

$$\frac{[] \vdash E_1 \Rightarrow \text{true} \quad \boxed{[] \vdash E_2 \Rightarrow v}}{[] \vdash \text{if}(E_1, E_2, E_3) \Rightarrow v}$$

已知

$$\frac{\varnothing \vdash E_1 : \text{bool} \quad \boxed{\varnothing \vdash E_2 : T} \quad \varnothing \vdash E_3 : T}{\varnothing \vdash \text{if}(E_1, E_2, E_3) : T}$$

根据归纳假设 $v : T$

无法应用归纳假设

$$\frac{[] \vdash E_1 \Rightarrow v_1 \quad \boxed{[x \mapsto v_1] \vdash E_2 \Rightarrow v}}{[] \vdash \text{let}(x, E_1, E_2) \Rightarrow v}$$

已知

$$\frac{\varnothing \vdash E_1 : T_1 \quad \boxed{x : T_1 \vdash E_2 : T_2}}{\varnothing \vdash \text{let}(x, E_1, E_2) : T_2}$$

类型系统的可靠性

- 若 $\varnothing \vdash E : T$ 且 $[] \vdash E \Rightarrow v$, 则 $v : T$
- **强化一下：**若 $\Gamma \vdash E : T$、$\rho \vdash E \Rightarrow v$ 且 $\rho : \Gamma$, 则 $v : T$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \mathsf{let}(x, E_1, E_2) \Rightarrow v}$$

已知

$$\frac{\Gamma \vdash E_1 : T_1 \quad \Gamma; x : T_1 \vdash E_2 : T_2}{\Gamma \vdash \mathsf{let}(x, E_1, E_2) : T_2}$$

根据归纳假设 $v_1 : T_1$

从而 $[x \mapsto v_1]\rho : \Gamma; x : T_1$

根据归纳假设 $v_2 : T_2$

$$\frac{\rho(x) = v}{\rho \vdash \mathsf{var}(x) \Rightarrow v}$$

已知

$$\frac{}{\Gamma \vdash \mathsf{var}(x) : \Gamma(x)}$$

从而 $\rho(x) : \Gamma(x)$

类型系统的可靠性

- 若 $\Gamma \vdash E : T$、$\rho \vdash E \Rightarrow v$ 且 $\rho : \Gamma$, 则 $v : T$

$$\frac{[f \mapsto \mathsf{clos}(f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \mathsf{letfun}(f, x, E_1, E_2) \Rightarrow v}$$

已知 $$\frac{\Gamma; f : T_1 \to T_2; x : T_1 \vdash E_1 : T_2 \qquad \Gamma; f : T_1 \to T_2 \vdash E_2 : T}{\Gamma \vdash \mathsf{letfun}(f, x, E_1, E_2) : T}$$

从而 $$\frac{\rho : \Gamma \qquad \Gamma, f : T_1 \to T_2, x : T_1 \vdash E_1 : T_2}{\mathsf{clos}(f, x, E_1, \rho) : T_1 \to T_2}$$

根据归纳假设 $v : T$

类型系统的可靠性

- 若 $\Gamma \vdash E : T$、$\rho \vdash E \Rightarrow v$ 且 $\rho : \Gamma$, 则 $v : T$

$$\dfrac{\rho \vdash E_1 \Rightarrow \mathsf{clos}(f', x', E_1', \rho') \quad \rho \vdash E_2 \Rightarrow v_2 \quad [x' \mapsto v_2, f' \mapsto \mathsf{clos}(f', x', E_1', \rho')]\rho' \vdash E_1' \Rightarrow v}{\rho \vdash \mathsf{call}(E_1, E_2) \Rightarrow v}$$

已知

$$\dfrac{\Gamma \vdash E_1 : T_1 \to T_2 \quad \Gamma \vdash E_2 : T_1}{\Gamma \vdash \mathsf{call}(E_1, E_2) : T_2}$$

根据归纳假设 $\mathsf{clos}(f', x', E_1', \rho') : T_1 \to T_2$ 且 $v_2 : T_1$

从而

$$\dfrac{\rho' : \Gamma' \qquad \boxed{\Gamma', f' : T_1 \to T_2, x' : T_1 \vdash E_1' : T_2}}{\mathsf{clos}(f', x', E_1', \rho') : T_1 \to T_2}$$

从而 $[x' \mapsto v_2, f' \mapsto \mathsf{clos}(f', x', E_1', \rho')]\rho' : \Gamma'; f' : T_1 \to T_2; x' : T_1$

根据归纳假设 $v : T_2$

类型系统的可靠性

- **完整的类型可靠性定理:** 若 $\Gamma \vdash E : T$ 且 $\rho : \Gamma$, 则
  - 要么存在一个 v, 使得 $\rho \vdash E \Rightarrow v$ 且 $v : T$
  - 要么 $\rho \vdash E \Rightarrow^{\omega}$, 即对 E 的求值不停机(也不出错)

- **一种证明方法: 扩展三元关系 $\rho \vdash E \Rightarrow v$ 支持「不停机」**

$$\frac{\rho \vdash E_1 \Rightarrow \circ}{\rho \vdash \mathsf{op}(E_1, E_2) \Rightarrow \circ}$$

「求值不停机也不出错」

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad \rho \vdash E_2 \Rightarrow \circ}{\rho \vdash \mathsf{op}(E_1, E_2) \Rightarrow \circ}$$

$$\frac{\rho \vdash E_1 \Rightarrow \circ}{\rho \vdash \mathsf{if}(E_1, E_2, E_3) \Rightarrow \circ}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathsf{true} \quad \rho \vdash E_2 \Rightarrow \circ}{\rho \vdash \mathsf{if}(E_1, E_2, E_3) \Rightarrow \circ}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathsf{false} \quad \rho \vdash E_3 \Rightarrow \circ}{\rho \vdash \mathsf{if}(E_1, E_2, E_3) \Rightarrow \circ}$$

$$\frac{\rho \vdash E_1 \Rightarrow \circ}{\rho \vdash \mathsf{let}(x, E_1, E_2) \Rightarrow \circ}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow \circ}{\rho \vdash \mathsf{let}(x, E_1, E_2) \Rightarrow \circ}$$

类型系统的可靠性

- **完整的类型可靠性定理:** 若 $\Gamma \vdash E : T$ 且 $\rho : \Gamma$, 则
  - 要么存在一个 v, 使得 $\rho \vdash E \Rightarrow v$ 且 $v : T$
  - 要么 $\rho \vdash E \Rightarrow^{\omega}$, 即对 E 的求值不停机(也不出错)

- **一种证明方法: 扩展三元关系 $\rho \vdash E \Rightarrow v$ 支持「不停机」**

$$\frac{[f \mapsto \mathsf{clos}(f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow \circ}{\rho \vdash \mathsf{letfun}(f, x, E_1, E_2) \Rightarrow \circ}$$

「求值不停机也不出错」

$$\frac{\rho \vdash E_1 \Rightarrow \circ}{\rho \vdash \mathsf{call}(E_1, E_2) \Rightarrow \circ} \qquad \frac{\rho \vdash E_1 \Rightarrow \mathsf{clos}(f', x', E_1', \rho') \quad \rho \vdash E_2 \Rightarrow \circ}{\rho \vdash \mathsf{call}(E_1, E_2) \Rightarrow \circ}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathsf{clos}(f', x', E_1', \rho') \quad \rho \vdash E_2 \Rightarrow v_2 \quad [x' \mapsto v_2, f' \mapsto \mathsf{clos}(f', x', E_1', \rho')]\rho' \vdash E_1' \Rightarrow \circ}{\rho \vdash \mathsf{call}(E_1, E_2) \Rightarrow \circ}$$

类型系统的可靠性

- **完整的类型可靠性定理**：若 $\Gamma \vdash E : T$ 且 $\rho : \Gamma$，则
  - 要么存在一个 v，使得 $\rho \vdash E \Rightarrow v$ 且 $v : T$
  - **要么 $\rho \vdash E \Rightarrow \circ$。**
- **一种证明方法：扩展三元关系 $\rho \vdash E \Rightarrow v$ 支持「不停机」**

$$\frac{[f \mapsto \mathsf{clos}(f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow \circ}{\rho \vdash \mathsf{letfun}(f, x, E_1, E_2) \Rightarrow \circ}$$

「求值不停机也不出错」

$$\frac{\rho \vdash E_1 \Rightarrow \circ}{\rho \vdash \mathsf{call}(E_1, E_2) \Rightarrow \circ} \qquad \frac{\rho \vdash E_1 \Rightarrow \mathsf{clos}(f', x', E_1', \rho') \quad \rho \vdash E_2 \Rightarrow \circ}{\rho \vdash \mathsf{call}(E_1, E_2) \Rightarrow \circ}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathsf{clos}(f', x', E_1', \rho') \quad \rho \vdash E_2 \Rightarrow v_2 \quad [x' \mapsto v_2, f' \mapsto \mathsf{clos}(f', x', E_1', \rho')]\rho' \vdash E_1' \Rightarrow \circ}{\rho \vdash \mathsf{call}(E_1, E_2) \Rightarrow \circ}$$

主要内容

- 类型、类型系统
- 类型系统的可靠性
- **多态类型**
- (代数)数据类型
- Rust 的类型系统

什么是多态？

- 多态(polymorphism)指的是一段程序可以同时支持多种不同的类型
  - 举例：面向对象编程语言中的**子类型多态**(subtyping polymorphism)
- ad-hoc polymorphism
  - 多态函数在不同类型上的行为是不同的
  - C++ 的运算符重载、Rust 的 trait、Haskell 的 typeclass 等
- **parametric polymorphism**
  - 多态函数在不同类型上的行为是相同的
  - C++ 的 template、Rust 的泛型、Haskell 的 universal types 等
- 在一个语言中，不同的多态可以混合存在

参数多态

- parametric polymorphism
- 例如排序函数，它需要适用于整数列表、布尔列表等

适用于任意类型 X

$$\forall X.(X \to X \to \text{bool}) \to (\text{list}(X) \to \text{list}(X))$$

C++

```
template <typename X>
void sort(vector<X> &vec, function<bool(const X&, const X&)> cmp);
```

Rust

```
fn sort<X, F>(vec: &mut Vec<X>, cmp: F)
where F: Fn(&X, &X) -> bool;
```

ML

```
val sort : ('x -> 'x -> bool) -> 'x list -> 'x list;
```

参数多态

- 参数多态提供了一种类型层面的**抽象(abstraction)**机制
- 例如排序函数：$\forall X.(X \to X \to \text{bool}) \to (\text{list}(X) \to \text{list}(X))$
  - 通常来说，该函数的实现只能「抽象」地操作 X 类型的值：可以移动、复制、作为参数传进比较函数，但不能观测这些值的内部结构
  - 也就是说，在排序函数的实现中，X 是一个**抽象类型**
- **类型在一定程度上约束了程序的行为**
  - $\forall X.X \to X$ 类型的值只可能是多态恒等函数
    - 当然，如果编程语言允许不停机或者副作用，上述说法就不成立
  - $\forall X.X \to X \to X$ 类型的值只有两种可能，要么是返回第一个参数的多态函数，要么是返回第二个参数的多态函数

Polymorphic micro-ML

- 在已有类型的基础上引入**类型变量 X** 和**多态类型 $\forall X.T$**

$$Type ::= \text{int} \mid \text{bool} \mid Type_1 \to Type_2 \mid X \mid \forall X.Type$$

- 扩展函数定义和函数调用以支持类型参数：

$$Exp ::= \text{const}(n) \mid \text{op}(Exp_1, Exp_2) \mid \text{if}(Exp_1, Exp_2, Exp_3) \mid \text{var}(x) \mid \text{let}(x, Exp_1, Exp_2)$$
$$\mid \text{letfun}(f, \langle \vec{X} \rangle, x, Exp_1, Exp_2) \mid \text{call}(Exp_1, \langle \vec{T} \rangle, Exp_2)$$

类型上下文中允许声明类型变量，可以在后面的类型绑定中使用

$$\frac{\Gamma; f : \forall \vec{X}.T_1 \to T_2; \vec{X}; x : T_1 \vdash E_1 : T_2 \qquad \Gamma; f : \forall \vec{X}.T_1 \to T_2 \vdash E_2 : T}{\Gamma \vdash \text{letfun}(f, \langle \vec{X} \rangle, x, E_1, E_2) : T}$$

$$\frac{\Gamma \vdash E_1 : \forall \vec{X}.T_1 \to T_2 \qquad \Gamma \vdash E_2 : T_1[\vec{T}/\vec{X}]}{\Gamma \vdash \text{call}(E_1, \langle \vec{T} \rangle, E_2) : T_2[\vec{T}/\vec{X}]}$$

将 T_1 和 T_2 中的类型变量 $\vec{X}$ 都替换为实际类型参数 $\vec{T}$

Polymorphic micro-ML

$\forall X.X \to X$

$\Gamma_1 = \mathtt{f} : \forall X.X \to X;\ X;\ \mathtt{x} : X$

$\Gamma_2 = \mathtt{f} : \forall X.X \to X$

```
let f <X> x = x
in
  if f <bool> true
  then f <int> 42
  else f <int> 233
end
```

$(X \to X)[\text{bool}/X] = \text{bool} \to \text{bool}$

$(X \to X)[\text{int}/X] = \text{int} \to \text{int}$

$\Gamma_1 = \mathtt{f} : \forall X.X \to X$

```
let f <X> x = x
in
  f <forall Y. Y->Y> f
end
```

$(X \to X)[(\forall Y.Y \to Y)/X] = (\forall Y.Y \to Y) \to (\forall Y.Y \to Y)$

类型擦除

- 在实际的程序运行过程中，并不需要追踪类型参数
- 编译器可以进行**类型擦除(type erasure)**来减少运行时开销

$$
\begin{aligned}
Exp ::= & \ \text{const}(n) \mid \text{op}(Exp_1, Exp_2) \mid \text{if}(Exp_1, Exp_2, Exp_3) \mid \text{var}(x) \mid \text{let}(x, Exp_1, Exp_2) \\
\mid & \ \text{letfun}(f, \langle \overrightarrow{X} \rangle, x, Exp_1, Exp_2) \mid \text{call}(Exp_1, \langle \overrightarrow{T} \rangle, Exp_2)
\end{aligned}
$$

  - 即擦除掉 letfun 中的 $\langle \overrightarrow{X} \rangle$ 和 call 中的 $\langle \overrightarrow{T} \rangle$

- **这样做可能会有什么问题？**
  - 在我们的 Polymorphic micro-ML 上没啥问题
  - 如果运行时需要根据类型选择不同的行为，则可能带来一些问题
    - 可以通过进一步扩展类型系统，允许将类型「反射」成一个值

主要内容

- 类型、类型系统
- 类型系统的可靠性
- 多态类型
- **(代数)数据类型**
- Rust 的类型系统

数据类型

- 我们日常使用的编程语言有着**更为丰富的类型构造方式**
  - C/C++ 中的结构体(struct)、枚举体(enum)、联合体(union)等
  - Haskell/ML 中的**代数数据类型(algebraic data type)**
- 本部分涵盖以下内容:
  - 乘积类型(product type)、加和类型(sum type)
  - 递归类型(recursive type)
  - 代数数据类型(algebraic data type)
  - 泛化代数数据类型(generalized algebraic data type)

乘积类型

- 二元乘积(binary product)类型表示的是二元组的类型

$$
\begin{aligned}
Exp &::= \cdots \mid \mathsf{pair}(Exp_1, Exp_2) \mid \mathsf{fst}(Exp_1) \mid \mathsf{snd}(Exp_1) \\
Type &::= \cdots \mid Type_1 \times Type_2
\end{aligned}
$$

- 针对二元组的求值规则($\rho \vdash E \Rightarrow v$):

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad \rho \vdash E_2 \Rightarrow v_2}{\rho \vdash \mathsf{pair}(E_1, E_2) \Rightarrow \langle v_1, v_2 \rangle} \qquad \frac{\rho \vdash E_1 \Rightarrow \langle v_1, v_2 \rangle}{\rho \vdash \mathsf{fst}(E_1) \Rightarrow v_1} \qquad \frac{\rho \vdash E_1 \Rightarrow \langle v_1, v_2 \rangle}{\rho \vdash \mathsf{snd}(E_1) \Rightarrow v_2}$$

- 针对二元乘积类型的类型规则($\Gamma \vdash E : T$):

$$\frac{\Gamma \vdash E_1 : T_1 \quad \Gamma \vdash E_2 : T_2}{\Gamma \vdash \mathsf{pair}(E_1, E_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash E_1 : T_1 \times T_2}{\Gamma \vdash \mathsf{fst}(E_1) : T_1} \qquad \frac{\Gamma \vdash E_1 : T_1 \times T_2}{\Gamma \vdash \mathsf{snd}(E_1) : T_2}$$

- 练习:给出 $\mathsf{letpair}(x, y, E_1, E_2)$ 的求值规则和类型规则,其含义是把 E_1 求值得到的二元组分别绑定到 x 和 y 上

乘积类型

- 从二元乘积可以推广得到多元乘积 $T_1 \times T_2 \times \cdots \times T_n$
- 给每个成分添加标号可以得到结构体 $\{\ell_i : T_i\}_{i=1}^{n}$
- **从代数的角度，零元乘积是类型乘积运算的「单位元」**

$$Exp ::= \cdots \mid \text{unit} \qquad Type ::= \cdots \mid 1$$

- 求值规则和类型规则：

$$\frac{}{\rho \vdash \text{unit} \Rightarrow \langle\rangle} \qquad \frac{}{\Gamma \vdash \text{unit} : 1}$$

- 对任意类型 T，以下类型是**同构**的：$1 \times T$、$T \times 1$、T
  - **如何说明这种同构性？**

加和类型

- **二元加和(binary sum)类型表示的是有两个分支的枚举类型**
  - 不同于 C/C++ 的枚举类型，加和类型的每个分支可以额外携带数据
  - 要使用加和类型的值，需要进行分类讨论(case)

$$
\begin{aligned}
Exp &::= \cdots \mid \mathsf{inl}(Exp_1) \mid \mathsf{inr}(Exp_1) \mid \mathsf{case}(Exp_1, Exp_2, Exp_3) \\
Type &::= \cdots \mid Type_1 + Type_2
\end{aligned}
$$

- 求值规则：

$$\frac{\rho \vdash E_1 \Rightarrow v_1}{\rho \vdash \mathsf{inl}(E_1) \Rightarrow \mathsf{inl}(v_1)} \qquad \frac{\rho \vdash E_1 \Rightarrow v_1}{\rho \vdash \mathsf{inr}(E_1) \Rightarrow \mathsf{inr}(v_1)}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathsf{inl}(v_1) \quad \rho \vdash \mathsf{call}(E_2, v_1) \Rightarrow v}{\rho \vdash \mathsf{case}(E_1, E_2, E_3) \Rightarrow v} \qquad \frac{\rho \vdash E_1 \Rightarrow \mathsf{inr}(v_1) \quad \rho \vdash \mathsf{call}(E_3, v_1) \Rightarrow v}{\rho \vdash \mathsf{case}(E_1, E_2, E_3) \Rightarrow v}$$

- **练习：使用加和类型来表达布尔类型**

加和类型

- **二元加和(binary sum)类型表示的是有两个分支的枚举类型**
  - 不同于 C/C++ 的枚举类型，加和类型的每个分支可以额外携带数据
  - 要使用加和类型的值，需要进行分类讨论(case)

$$
\begin{aligned}
Exp &::= \cdots \mid \mathsf{inl}(Exp_1) \mid \mathsf{inr}(Exp_1) \mid \mathsf{case}(Exp_1, Exp_2, Exp_3) \\
Type &::= \cdots \mid Type_1 + Type_2
\end{aligned}
$$

- 类型规则：

$$\frac{\Gamma \vdash E_1 : T_1}{\Gamma \vdash \mathsf{inl}(E_1) : T_1 + T_2} \qquad \frac{\Gamma \vdash E_1 : T_2}{\Gamma \vdash \mathsf{inr}(E_1) : T_1 + T_2}$$

$$\frac{\Gamma \vdash E_1 : T_1 + T_2 \quad \Gamma \vdash E_2 : T_1 \to T \quad \Gamma \vdash E_3 : T_2 \to T}{\Gamma \vdash \mathsf{case}(E_1, E_2, E_3) : T}$$

- **练习：使用加和类型的类型规则推导布尔类型的类型规则**

加和类型

- 从二元加和可以推广得到多元加和 $T_1 + T_2 + \cdots + T_n$
- 给每个分支加标号可以得到枚举体 $< \ell_i : T_i >_{i=1}^{n}$
- **从代数的角度，零元加和是类型加和的「单位元」**

$$Exp ::= \cdots \mid \mathsf{absurd}(Exp_1) \qquad Type ::= \cdots \mid 0$$

- 求值规则和类型规则：

$$\frac{\Gamma \vdash E_1 : 0}{\Gamma \vdash \mathsf{absurd}(E_1) : T}$$

为什么没有求值规则？

- 对任意类型 T，以下类型是**同构**的：$0 + T$、$T + 0$、T
  - **练习：说明这种同构性**

递归类型

- 有了乘积和加和类型后，我们可以讨论递归类型了
- 对于自然数类型 nat，有**同构关系** $\text{nat} \simeq 1 + \text{nat}$
  - 一个自然数要么是零(左分支)，要么是自然数的后继(右分支)
- 对于列表类型 $\text{list}(X)$，有**同构关系** $\text{list}(X) \simeq 1 + X \times \text{list}(X)$
  - 一个列表要么是空列表(左分支)，要么是头部元素和尾部列表的二元组(右分支)
- **如何理解这种递归的同构「方程」？**
  - 不同的编程语言可以有不同的理解方式
  - 大体上可以分为三种

递归类型：理解之一

- equi-recursive type

- 把类型同构关系 $\simeq$ 理解成**类型等价关系**
  - 对于自然数，在类型检查时，有 $\text{nat} = 1 + \text{nat}$
  - 对于列表，在类型检查时，有 $\text{list}(X) = 1 + X \times \text{list}(X)$
  - 表达式 inl(unit) 既可以具有类型 nat，也可以具有类型 $\text{list}(X)$

- 类型检查算法更复杂，但是理论复杂性基本不变
  - 即类型规则、类型系统可靠性证明等基本不用变

递归类型：理解之一

$T_1 \to T_f$

T_1

$\Gamma_1 = \texttt{makefac} : T_1 \to T_f; \texttt{maker} : T_1$

int → int

int

```
let makefac maker =
  let f x = if x == 0 then 1 else x * ((maker maker) (x - 1))
  in f end
in
  let fac = makefac makefac
  in fac 3 end
end
```

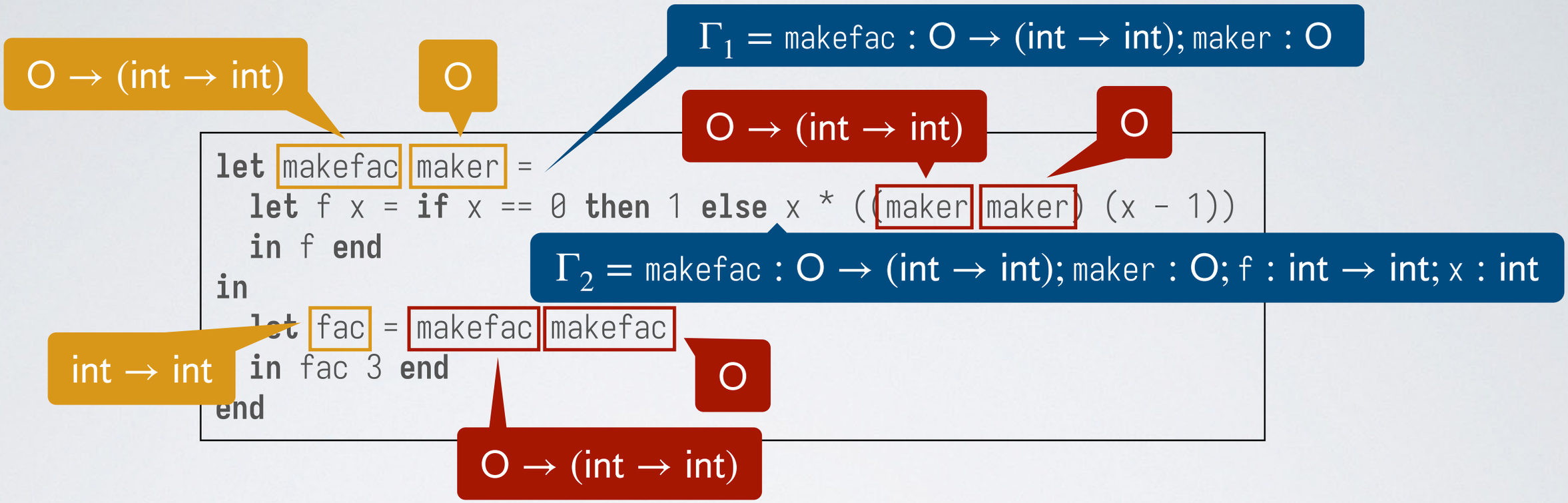
int

$\Gamma_2 = \texttt{makefac} : T_1 \to T_f; \texttt{maker} : T_1; \texttt{f} : T_f; \texttt{x} : \text{int}$

$$T_1 = T_1 \to (\text{int} \to \text{int})$$

- 引入递归类型 O 满足 $O \simeq O \to (\text{int} \to \text{int})$

递归类型：理解之一



- 引入递归类型 O 满足 $O \simeq O \to (int \to int)$
  - 对第一个 maker，「展开」类型 O 一次得到函数类型
  - 对第二个 makefac，「折叠」函数类型一次得到类型 O

递归类型：理解之二

- structural iso-recursive type

- 把类型同构关系 $\simeq$ 理解成**类型结构的同构关系**
  - 引入新的表达式作为同构关系的「证据」
  - fold 即「折叠」，unfold 即「展开」

$$\boxed{\begin{array}{l} Exp ::= \cdots \mid \mathsf{fold}_{Type}(Exp_1) \mid \mathsf{unfold}_{Type}(Exp_1) \\ Type ::= \cdots \mid \mu X\,.\,Type \end{array}}$$

- 求值规则和类型规则：

$$\frac{\rho \vdash E_1 \Rightarrow v_1}{\rho \vdash \mathsf{fold}_T(E_1) \Rightarrow \mathsf{fold}_T(v_1)}$$

$$\frac{\rho \vdash E_1 \Rightarrow \mathsf{fold}_T(v_1)}{\rho \vdash \mathsf{unfold}_T(E_1) \Rightarrow v_1}$$

$$\frac{\Gamma \vdash E_1 : \boxed{T[(\mu X\,.\,T)/X]}}{\Gamma \vdash \mathsf{fold}_{\mu X.T}(E_1) : \mu X\,.\,T}$$

将 T 中的 X 都替换为递归类型本身 $\mu X\,.\,T$ 来表示「展开」

$$\frac{\Gamma \vdash E_1 : \mu X\,.\,T}{\Gamma \vdash \mathsf{unfold}_{\mu X.T}(E_1) : \boxed{T[(\mu X\,.\,T)/X]}}$$

递归类型：理解之二

- 对于自然数，定义 nat 为 $\mu X.1 + X$
  - inl(unit) 的类型可以是 $1 + (\mu X.1 + X)$，即 $1 +$ nat
  - $\text{fold}_{\mu X.1+X}$(inl(unit)) 的类型为 $\mu X.1 + X$，即 nat
- 对于列表，定义 list(X) 为 $\mu Y.1 + X \times Y$
  - inl(unit) 的类型可以是 $1 + (\mu Y.1 + X \times Y)$，即 $1 +$ list(X)
  - $\text{fold}_{\mu Y.1+X\times Y}$(inl(unit)) 的类型为 $\mu Y.1 + X \times Y$，即 list(X)
- 类型检查算法基本不变，但是理论复杂性有所增加
  - 类型系统可靠性需要针对递归类型补充证明

递归类型：理解之三

- nominal iso-recursive type

- 把类型同构关系 $\simeq$ 理解成**类型「名字」的同构关系**
  - 为每个递归类型引入一个单独的名字
  - 把「折叠」和「展开」隐藏在每个递归类型相关的表达式中

$$
\begin{array}{l}
Exp ::= \cdots \mid \mathsf{inl}(Exp_1) \mid \mathsf{inr}(Exp_1) \mid \mathsf{case}(Exp_1, Exp_2, Exp_3) \\
Type ::= \cdots \mid \mathsf{nat}
\end{array}
$$

- 类型规则：

$$
\frac{\Gamma \vdash E_1 : 1}{\Gamma \vdash \mathsf{inl}(E_1) : \mathsf{nat}} \qquad \frac{\Gamma \vdash E_1 : \mathsf{nat}}{\Gamma \vdash \mathsf{inr}(E_1) : \mathsf{nat}} \qquad \frac{\Gamma \vdash E_1 : \mathsf{nat} \quad \Gamma \vdash E_2 : 1 \to T \quad \Gamma \vdash E_3 : \mathsf{nat} \to T}{\Gamma \vdash \mathsf{case}(E_1, E_2, E_3) : T}
$$

- 某种程度上，nat 是一个**抽象类型**

递归类型：理解之三

- nominal iso-recursive type

- 把类型同构关系 $\simeq$ 理解成**类型「名字」的同构关系**
  - 为每个递归类型引入一个单独的名字
  - 把「折叠」和「展开」隐藏在每个递归类型相关的表达式中

$$
\begin{aligned}
Exp &::= \cdots \mid \mathsf{inl}(Exp_1) \mid \mathsf{inr}(Exp_1) \mid \mathsf{case}(Exp_1, Exp_2, Exp_3) \\
Type &::= \cdots \mid \mathsf{list}(Type)
\end{aligned}
$$

- 类型规则：

$$\frac{\Gamma \vdash E_1 : 1}{\Gamma \vdash \mathsf{inl}(E_1) : \mathsf{list}(T)} \qquad \frac{\Gamma \vdash E_1 : T \times \mathsf{list}(T)}{\Gamma \vdash \mathsf{inr}(E_1) : \mathsf{list}(T)}$$

$$\frac{\Gamma \vdash E_1 : \mathsf{list}(T) \quad \Gamma \vdash E_2 : 1 \to T' \quad \Gamma \vdash E_3 : T \times \mathsf{T} \to T'}{\Gamma \vdash \mathsf{case}(E_1, E_2, E_3) : T'}$$

基本上都是乘积类型和加和类型相关规则的特例

代数数据类型

- algebraic data type (ADT)
- 按照 nominal iso-recursive type 的思路，允许程序员使用**乘积、加和、递归类型的组合**来自定义数据类型

Rust

```rust
enum Nat {
  Zero,
  Succ(Nat),
}

enum List<X> {
  Nil,
  Cons(X, Box<List<X>>),
}
```

ML

```
type nat =
  | Zero
  | Succ of nat
;;

type 'x list =
  | Nil
  | Cons of 'x * 'x list
;;
```

代数数据类型

- 每个代数数据类型引入若干个**构造器(constructor)**:

```
let one : nat = Succ Zero;;

let singleton : nat list = Cons (one, Nil);;
```

- 使用**模型匹配(pattern match)**来进行检视和分类讨论:

```
let predecessor
  (n : nat)
: nat =
  match n with
  | Zero -> Zero
  | Succ m -> m
;;
```

```
let rec append
  (l1 : 'x list) (l2 : 'x list)
: 'x list =
  match l1 with
  | Nil -> l2
  | Cons (h, t) -> Cons (h, append t l2)
;;
```

代数数据类型

- 「We suspect that a great many errors are caused by the complications introduced when encoding data in terms of the commonly-supplied low-level types; the provision of **a simple and powerful facility for defining types** should greatly simplify the programmer's task」
  - Burstall, MacQueen, Sannella, HOPE: An experimental applicative language, 1980.
- 代数数据类型的关键特征：
  - 具名类型
  - 具名数据构造器
  - 模式匹配

代数数据类型

- 一些语言中允许另一种代数数据类型的声明方式:

```
type nat =
  | Zero
  | Succ of nat
;;

type 'x list =
  | Nil
  | Cons of 'x * 'x list
;;
```

```
type nat =
  | Zero :          nat
  | Succ : nat -> nat
;;

type _ list =
  | Nil  :                    'x list
  | Cons : 'x * 'x list -> 'x list
;;
```

- 注意到最右侧的类型都是一致的(包括类型参数)
  - **有可能允许不同的分支中最右侧得到的类型具有不同的参数吗?**

案例：表达式求值

$$Exp ::= \text{int}(n) \mid \text{pair}(Exp_1, Exp_2) \mid \text{fst}(Exp_1) \mid \text{snd}(Exp_1)$$

```
type exp =
  | EInt of int
  | EPair of exp * exp
  | EFst of exp
  | ESnd of exp
;;

type value =
  | VInt of int
  | VPair of value * value
;;
```

```
let as_pair (v : value) : value * value =
  match v with
  | VPair (v1, v2) -> (v1, v2)
  | _ -> assert false
;;

let rec eval (e : exp) : value =
  match e with
  | EInt x -> VInt x
  | EPair (e1, e2) -> VPair (eval e1, eval e2)
  | EFst e -> fst (as_pair (eval e))
  | ESnd e -> snd (as_pair (eval e))
;;
```

由于表达式可能不是「良类型」的，求值时需要运行时检查

案例：表达式求值

- 可以让表达式的数据类型 exp 携带其「类型」信息

```
type _ exp =
  | EInt  :                    int ->          int exp
  | EPair : 't1 exp * 't2 exp -> ('t1 * 't2) exp
  | EFst  :   ('t1 * 't2) exp ->          't1 exp
  | ESnd  :   ('t1 * 't2) exp ->          't2 exp
;;

type _ value =
  | VInt  :                      int ->          int value
  | VPair : 't1 value * 't2 value -> ('t1 * 't2) value
;;
```

t exp 表示该表达式的求值结果的类型是 t value

- **generalized abstract data type (GADT)**
  - 允许 ADT 的不同分支有不同的类型参数

案例：表达式求值

$\forall X_1, X_2 \,.\, \text{value}(X_1 \times X_2) \to \text{value}(X_1) \times \text{value}(X_2)$

```
let as_pair : type t1 t2 . (t1 * t2) value -> t1 value * t2 value =
  fun v ->
    match v with
    | VPair (v1, v2) -> (v1, v2)
;;

let rec eval : type t . t exp -> t value =
  fun e ->
    match e with
    | EInt x -> VInt x
    | EPair (e1, e2) -> VPair (eval e1, eval e2)
    | EFst e -> fst (as_pair (eval e))
    | ESnd e -> snd (as_pair (eval e))
;;
```

由于表达式总是「良类型」的，求值时不需要运行时检查

$\forall X \,.\, \text{exp}(X) \to \text{value}(X)$

案例：表达式求值

- 甚至可以不需要引入 value 数据类型

```
let rec eval : type t . t exp -> t =
  fun e ->
    match e with
    | EInt x -> x
    | EPair (e1, e2) -> (eval e1, eval e2)
    | EFst e -> fst (eval e)
    | ESnd e -> snd (eval e)
;;
```

- **挑战：如何扩展表达式数据类型支持变量和 let 表达式？**
  - 提示：exp 类型需要两个参数，即 ('g, 't) exp，其中 'g 表示上下文
  - 提示：用自然数表示变量，表示从内到外第几个 let 引入的变量

案例：良类型的 printf

```
# open Printf;;
# printf "%d * %s = %d\n" 2 "12" 24;;
2 * 12 = 24
- : unit = ()
```

- printf 后面需要传入参数的个数和类型依赖于**格式字符串**

- **问题：如何使得整个 printf 的行为是良类型的？**

```
# let desc : _ format6 = "%d * %s = %d\n";;
val desc : (int -> string ->int -> 'a, 'b, 'c, 'd, 'd, 'a) format6 =
  Format
   (Int (Int_d, No_padding, No_precision,
     String_literal (" * ",
      String (No_padding,
       String_literal (" = ",
        Int (Int_d, No_padding, No_precision,
         Char_literal ('\n', End_of_format))))))),
   "%d * %s = %d\n")
```

格式字符串不只是个字符串，它会被解析成一个复杂的数据结构

案例：良类型的 printf

- **这个数据结构本质上是一个「列表」：**

其它的都相当于 cons，但表示的列表元素类型可以不相同：

- Int、String 对应 "%d"、"%s"
- String_literal、Char_literal 对应格式中的字符串、字符字面量

```
Int (Int_d, No_padding, No_precision,
String_literal (" * ",
String (No_padding,
String_literal (" = ",
Int (Int_d, No_padding, No_precision,
Char_literal ('\n',
End_of_format)))))
```

End_of_format 相当于 nil

- **让我们用 GADT 来实现一个简化版：**

```
type desc =
  | Nil
  | Lit of string * desc
  | Int of desc
  | Str of desc
```

→

```
type _ desc =
  | Nil :                          ?? desc
  | Lit : string * ?? desc -> ?? desc
  | Int :          ?? desc -> ?? desc
  | Str :          ?? desc -> ?? desc
```

案例：良类型的 printf

```
type _ desc =
  | Nil :                              ?? desc
  | Lit : string * ?? desc ->          ?? desc
  | Int :          ?? desc ->          ?? desc
  | Str :          ?? desc ->          ?? desc
```

- 问题：**printf 的类型应该是什么？**
  - `printf : 'a desc -> 'a`
  - `printf "%d * %s = %d" : int -> string -> int -> unit`
  - `"%d * %s = %d" = Int (Lit (" * ", Str (Lit (" = ", Int Nil)))) : (int -> string -> int -> unit) desc`
- Nil 表示的是没有额外需要输出的东西了

案例：良类型的 printf

```
type _ desc =
  | Nil :                          unit desc
  | Lit : string * ?? desc ->   ?? desc
  | Int :             ?? desc ->   ?? desc
  | Str :             ?? desc ->   ?? desc
```

- Nil 表示的是没有额外需要打印的东西了
  - 所以它的类型是 unit desc
- Lit (s, d1) 表示的是先打印字符串 s，再按照 d1 进行打印

案例：良类型的 printf

```
type _ desc =
  | Nil :                          unit desc
  | Lit : string * 'a desc ->       'a desc
  | Int :             ?? desc ->    ?? desc
  | Str :             ?? desc ->    ?? desc
```

- Nil 表示的是没有额外需要打印的东西了
  - 所以它的类型是 unit desc
- Lit (s, d1) 表示的是先打印字符串 s，再按照 d1 进行打印
  - 所以它的类型应该与 d1 的类型相同
- Int d1 表示输入一个整数并打印，然后按照 d1 进行打印

案例：良类型的 printf

```
type _ desc =
  | Nil :                                  unit desc
  | Lit : string * 'a desc ->                 'a desc
  | Int :             'a desc -> (int -> 'a) desc
  | Str :             ?? desc ->                 ?? desc
```

- Nil 表示的是没有额外需要打印的东西了
  - 所以它的类型是 unit desc
- Lit (s, d1) 表示的是先打印字符串 s，再按照 d1 进行打印
  - 所以它的类型应该与 d1 的类型相同
- Int d1 表示输入一个整数并打印，然后按照 d1 进行打印
  - 如果 d1 的类型是 'a desc，那么 Int d1 是 (int -> 'a) desc

案例：良类型的 printf

```
type _ desc =
  | Nil :                                  unit desc
  | Lit : string * 'a desc ->                 'a desc
  | Int :             'a desc ->    (int -> 'a) desc
  | Str :             'a desc -> (string -> 'a) desc
```

- Nil 表示的是没有额外需要打印的东西了
  - 所以它的类型是 unit desc
- Lit (s, d1) 表示的是先打印字符串 s，再按照 d1 进行打印
  - 所以它的类型应该与 d1 的类型相同
- Int d1 表示输入一个整数并打印，然后按照 d1 进行打印
  - 如果 d1 的类型是 'a desc，那么 Int d1 是 (int -> 'a) desc

案例：良类型的 printf

```
type _ desc =
  | Nil :                                  unit desc
  | Lit : string * 'a desc ->                 'a desc
  | Int :           'a desc ->    (int -> 'a) desc
  | Str :           'a desc -> (string -> 'a) desc
```

```
let rec printf : type a . a desc -> a =
  fun d ->
    match d with
    | Nil ->
      ???
    | Lit (s, d1) ->
      ???
    | Int d1 ->
      ???
    | Str d1 ->
      ???
```

此处类型应为 unit

案例：良类型的 printf

```
type _ desc =
  | Nil :                                  unit desc
  | Lit : string * 'a desc ->                 'a desc
  | Int :             'a desc ->    (int -> 'a) desc
  | Str :             'a desc -> (string -> 'a) desc
```

```
let rec printf : type a . a desc -> a =
  fun d ->
    match d with
    | Nil ->
      ()
    | Lit (s, d1) ->
      ???
    | Int d1 ->
      ???
    | Str d1 ->
      ???
```

d1 的类型为 a desc

此处类型应为 a

案例：良类型的 printf

```
type _ desc =
  | Nil :                                  unit desc
  | Lit : string * 'a desc ->                 'a desc
  | Int :           'a desc ->    (int -> 'a) desc
  | Str :           'a desc -> (string -> 'a) desc
```

```
let rec printf : type a . a desc -> a =
  fun d ->
    match d with
    | Nil ->
      ()
    | Lit (s, d1) ->
      print_string s; printf d1
    | Int d1 ->
      ???
    | Str d1 ->
      ???
```

d1 的类型为 a desc

此处类型应为 int -> a

案例：良类型的 printf

```
type _ desc =
  | Nil :                                  unit desc
  | Lit : string * 'a desc ->                 'a desc
  | Int :             'a desc ->    (int -> 'a) desc
  | Str :             'a desc -> (string -> 'a) desc
```

```
let rec printf : type a . a desc -> a =
  fun d ->
    match d with
    | Nil ->
      ()
    | Lit (s, d1) ->
      print_string s; printf d1
    | Int d1 ->
      fun n -> print_int n; printf d1
    | Str d1 ->
      ???
```

案例：良类型的 printf

```
type _ desc =
  | Nil :                                  unit desc
  | Lit : string * 'a desc ->                 'a desc
  | Int :           'a desc ->    (int -> 'a) desc
  | Str :           'a desc -> (string -> 'a) desc
```

```
let rec printf : type a . a desc -> a =
  fun d ->
    match d with
    | Nil ->
      ()
    | Lit (s, d1) ->
      print_string s; printf d1
    | Int d1 ->
      fun n -> print_int n; printf d1
    | Str d1 ->
      fun s -> print_string s; printf d1
```

案例：良类型的 printf

```
type _ desc =
  | Nil :                                  unit desc
  | Lit : string * 'a desc ->                 'a desc
  | Int :             'a desc ->    (int -> 'a) desc
  | Str :             'a desc -> (string -> 'a) desc
```

```
# let desc = Int (Lit (" * ", Str (Lit (" = ", Int (Lit ("\n", Nil))))));
val desc : (int -> string ->int -> unit) desc =
  Int (Lit (" * ", Str (Lit (" = ", Int (Lit ("\n", Nil))))))

# printf desc 2 "12" 24;;
2 * 12 = 24
- : unit = ()
```

- 关于 GADT 的形式化可参考：Xi, Chen, Chen, Guarded Recursive Datatype Constructors, 2003.

主要内容

- 类型、类型系统
- 类型系统的可靠性
- 多态类型
- (代数)数据类型
- **Rust 的类型系统**

Rust 101

```rust
// i32 为 32 位带符号整型
fn add2(x: i32, y: i32) -> i32 {
    x + y // 隐式函数返回值
}
```

```rust
let x = 1; // 不可变变量

let mut y = 1; // 可变变量
y = y + 1; // 可变变量赋值
```

```rust
// C 风格的枚举类型
enum Direction {
    Left, Right,
    Up, Down,
}
let up = Direction::Up;
```

```rust
// 枚举类型可携带额外信息域
enum OptionalI32 {
    AnI32(i32),
    Nothing,
}
let two = OptionalI32::AnI32(2);
let nothing = OptionalI32::Nothing;
```

```rust
// 模式匹配
match two {
    OptionalI32::AnI32(n) => println!("i32: {}", n),
    OptionalI32::Nothing => println!("nothing"),
}
```

Rust 101

```rust
// 拥有数据所有权的指针
// 当所有者作用域结束时，自动释放内存
let mut mine: Box<i32> = Box::new(3);
*mine = 5;
```

```rust
// 转移数据所有权
let mut now_its_mine = mine;
*now_its_mine += 2;
// println!("{}", mine); // 这个会报错
```

```rust
// 使用 Box 定义递归类型
enum List {
    Nil,
    // Cons(i32, List) 会报错
    // 因为无法静态计算类型大小
    Cons(i32, Box<List>),
}

use List::{Cons, Nil};
```

```rust
// l1 拥有这个列表的所有权
let l1 = Cons(1,
              Box::new(Cons(2,
                            Box::new(Nil))));
// 列表的所有权从 l1 转移到 l2
let l2 = l1;
// let l3 = l1; // 这个会报错
```

在 Rust 中进行函数式编程

```rust
// 计算列表 l1 和 l2 连接后所得的列表
fn append(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, tl) => Cons(hd, Box::new(append(*tl, l2))),
    }
}
```

对 tl: Box<List> 解引用后，tl 指向的内存会被自动释放

为了构建新的 List，需要申请一块内存来存放数据

```rust
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice(l: List) -> List {
    match l {
        Nil => Nil,
        Cons(hd, tl) => Cons(hd, Box::new(Cons(hd, Box::new(twice(*tl))))),
    }
}
```

在 Rust 中进行函数式编程

```rust
// 计算列表 l1 和 l2 连接后所得的列表
fn append2(l1: List, l2: List) -> List {
    match l1 {
        Nil => l2,
        Cons(hd, mut Δ) => {
            let tl: List = *Δ;
            *Δ = append2(tl, l2);
            Cons(hd, Δ)
        }
    }
}
```

这是一个 Box<List>

将 Δ 指向内存中的列表
转移到 tl 变量所拥有

复用 Δ 指向的内存来存储
(并拥有)递归调用返回的列表

通过复用 Box<List> 完成了函数式编程，避免了额外的内存开销

在 Rust 中进行函数式编程

```rust
// 计算将 a 插入有序列表 l 中后所得的列表
fn insert(a: i32, l: List) -> List {
    match l {
        Nil => Cons(a, Box::new(Nil)),
        Cons(hd, tl) => {
            if a <= hd {
                Cons(a, Box::new(Cons(hd, tl)))
            } else {
                Cons(hd, Box::new(insert(a, *tl)))
            }
        }
    }
}
```

因为返回列表长度加一，所以一定需要一个额外的 Box::new

在 Rust 中进行函数式编程

```rust
// 计算将 a 插入有序列表 l 中后所得的列表
fn insert2(mut Δ1: Box<List>, a: i32, l: List) -> List {
    match l {
        Nil => {
            *Δ1 = Nil;
            Cons(a, Δ1)
        }
        Cons(hd, mut Δ2) => {
            let tl = *Δ2;
            if a <= hd {
                *Δ2 = tl;
                *Δ1 = Cons(hd, Δ2);
                Cons(a, Δ1)
            } else {
                *Δ2 = insert2(Δ1, a, tl);
                Cons(hd, Δ2)
            }
        }
    }
}
```

显式传入一块可用的内存，在插入 a 的位置使用

当需要递归调用时，把这块可用内存也作为参数传过去

通过显式传递函数计算中所需的额外内存，避免了在函数内部进行内存分配

在 Rust 中进行函数式编程

```rust
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>,
        hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

```rust
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

```rust
// 计算将 a 插入有序列表 l 中后所得的列表
fn insert3(Δ1: Box<List>, a: i32, l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => cons(Δ1, a, nil()),
        ListDestr::Cons(Δ2, hd, tl) => {
            if a <= hd {
                cons(Δ1, a, cons(Δ2, hd, tl))
            } else {
                cons(Δ2, hd, insert3(Δ1, a, tl))
            }
        }
    }
}
```

函数式但是「原地」更新数据结构！

Non-Size-Increasing Computation

- 由于计算中所需的额外内存要通过参数传入，所以计算整体**不增加内存使用**
- 能表达的可计算函数等价于 EXPTIME，即指数时间可计算

Δ 只能被使用一次，故该程序不能通过类型检查

```
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice2(l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => Nil,
        ListDestr::Cons(Δ, hd, tl) => cons(Δ, hd, cons(Δ, hd, twice2(tl))),
    }
}
```

M. Hofmann. 2000. A type system for bounded space and functional in-place update. In ESOP'00.

Non-Size-Increasing Computation

- 由于计算中所需的额外内存要通过参数传入，所以计算整体**不增加内存使用**
- 能表达的可计算函数等价于 EXPTIME，即指数时间可计算

```
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice2(mut extra: Vec<Box<List>>, l: List) -> List {
    match list_destr(l) {
        ListDestr::Nil => Nil,
        ListDestr::Cons(Δ1, hd, tl) => {
            let Δ2 = extra.pop().unwrap();
            cons(Δ1, hd, cons(Δ2, hd, twice2(extra, tl)))
        }
    }
}
```

twice2 函数需要的额外内存块等于 l 的长度，不是一个常数，故通过一个 Vec 传进来

M. Hofmann. 2000. A type system for bounded space and functional in-place update. In ESOP'00.

从资源的角度看内存

```
fn nil() -> List {
    Nil
}
fn cons(mut Δ: Box<List>,
        hd: i32, tl: List) -> List {
    *Δ = tl;
    Cons(hd, Δ)
}
```

在构建 Cons 列表时，需要一个**内存块**

在构建 Cons 列表时，需要一单位**资源**

```
enum ListDestr {
    Nil,
    Cons(Box<List>, i32, List),
}
fn list_destr(l: List) -> ListDestr {
    match l {
        Nil => ListDestr::Nil,
        Cons(hd, mut Δ) => {
            let tl = *Δ;
            *Δ = Nil;
            ListDestr::Cons(Δ, hd, tl)
        }
    }
}
```

在解构 Cons 列表时，得到一个可用**内存块**

在解构 Cons 列表时，得到一单位可用**资源**

以一个抽象的视角来看

```
// Trait 就像是 Interface 或 Typeclass
trait ListTrait: Sized {
  type Resource;

  fn nil() -> Self;
  fn cons(Δ: Self::Resource, hd: i32, tl: Self) -> Self;

  fn destr(self) -> Option<(Self::Resource, i32, Self)>;
}
```

某种**资源**

将**资源**存储在数据结构中

从数据结构中获取**资源**

- 数据结构中存储的资源可**支付对应的资源消耗**，比如循环和递归调用的次数
- 若将资源视作数据结构**类型的静态信息**，则得到了一种**静态资源分析**方法

以一个抽象的视角来看

```
// 在类型中追踪资源信息
trait ListTraitStatic<Resource>: Sized {
    fn nil() -> Self;
    fn cons(Δ: Resource, hd: i32, tl: Self) -> Self;

    fn destr(self) -> Option<(Resource, i32, Self)>;
}
```

泛型的类型参数

```
// 资源的计数单位
struct Delta;
```

在类型层面，通过 (Delta, Delta, …) 来表示多个单位的资源

```
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<(Delta, Delta)>,
    T2: ListTraitStatic<Delta>,
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) => T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

输入列表中每个元素带 2 单位的资源

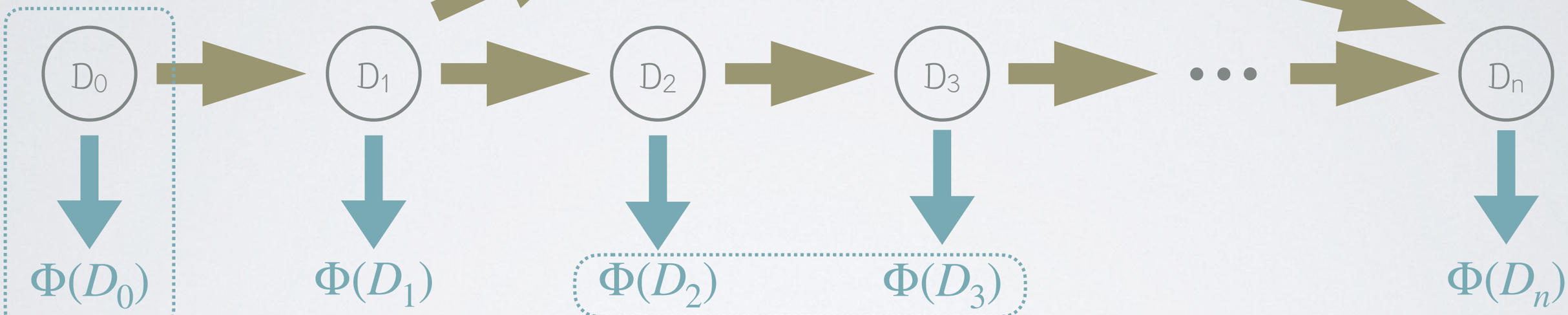
输出列表中每个元素带 1 单位的资源

数据结构存储的资源即其势能

均摊分析的势能方法

D_i 表示程序运行的状态

有向边表示状态之间的转移及资源消耗



程序初始化时的势能是总资源消耗的上界!

$$\Phi(D_2) \geq Cost(D_2, D_3) + \Phi(D_3)$$

势能函数将每个程序状态映射为一个非负数

带有势能标注的类型

```
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<(Delta, Delta)>,
    T2: ListTraitStatic<Delta>,
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) =>
            T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

带有势能标注的类型

```
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>,  → 表示有 2 个 Delta，即 2 单位的势能
    T2: ListTraitStatic<1>,  → 表示有 1 个 Delta，即 1 单位的势能
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) =>
            T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

带有势能标注的类型

```
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<2>,  → 表示有 2 个 Delta，即 2 单位的势能
    T2: ListTraitStatic<1>,  → 表示有 1 个 Delta，即 1 单位的势能
{
    match l.destr() {
        None => T2::nil(),
        Some(((Δ1, Δ2), hd, tl)) =>
            T2::cons(Δ1, hd, T2::cons(Δ2, hd, twice3(tl))),
    }
}
```

得到 2 单位势能

存储 1 单位势能

存储 1 单位势能

带有势能标注的类型

```rust
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<3>,
    T2: ListTraitStatic<1>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 3 个 Delta，即 3 单位的势能

可以用于分析时间复杂度

得到 3 单位势能

消耗 1 单位势能

存储 1 单位势能

存储 1 单位势能

通过线性规划进行自动资源分析

```rust
// 计算将 l 中每个元素拷贝两次所得的列表
fn twice3<T1, T2>(l: T1) -> T2
where
    T1: ListTraitStatic<p>,
    T2: ListTraitStatic<q>,
{
    match l.destr() {
        None => T2::nil(),
        Some((hd, tl)) => {
            tick<1>();
            T2::cons(hd, T2::cons(hd, twice3(tl)))
        }
    }
}
```

表示有 **p** 个 Delta，即 **p** 单位的势能

消耗 **1** 单位势能

得到 **p** 单位势能

存储 **q** 单位势能

存储 **q** 单位势能

$$p \geq 0$$
$$q \geq 0$$
$$p \geq 1 + q + q$$

- $p = 3, q = 1$: `ListTraitStatic<3> -> ListTraitStatic<1>`
- $p = 5, q = 2$: `ListTraitStatic<5> -> ListTraitStatic<2>`

Rust 的类型系统

- Rust 的类型系统挺复杂，本节仅考虑**线性类型**相关的内容
  - 即不考虑生命周期、借用机制等特性

- （参考《编程语言的设计原理》课程的课件）

参考文献

- Benjamin C. Pierce. Types and Programming Languages. The MIT Press.
- Robert Harper. Practical Foundations for Programming Languages. Cambridge University Press.
- James Noble and Tobias Wrigstad. RUST: Regions, Uniqueness, Ownership & Types.
- Qihao Lian and Di Wang. 2025. Automatic Linear Resource Bound Analysis for Rust via Prophecy Potentials. In OOPSLA'25.