# 编程语言中的 logical relation

这篇文章会介绍使用 logical relation 研究类型的语义的方法，以及它的各种变种和应用。阅读本文需要以下的前置知识：

- 能够读懂基本的类型规则和 operational semantic
- 能够读懂基本的集合论记号，知道集合论中的 relation 等基本定义

Logical relation 在编程语言，尤其是类型系统的研究中几乎可以说是无处不在。另一方面，各种不同语言、不同应用中对 logical relation 的应用又有很多相似性。这当然不是巧合。当我们定义一门语言时，我们往往会先定义语法，然后在语法上定义 operational semantic，或是其他形式的语义。同时，我们也会定义类型与类型规则，来给不同的程序赋予不同的类型。那么，操作语义与类型和类型规则应该如何联系起来呢？Logical relation 就是完成这一联系的一种通用方法。

这篇文章旨在把 logical relation 的不同应用摆在一起，呈现它们的相似性。对于有基础的读者来说，很多内容和细节可能有些简单，这时读者可以放心跳过：本文的不同章节之间没有很强的依赖关系。

# Contents

# 1 STLC 的停机证明

首先，让我们通过一个简单的应用来展示 logical relation 这一方法的大致模样。考虑经典的 Simply Typed Lambda Calculus（STLC），加上一个内建的 `Answer` 类型。它有如下的语法和类型规则：

$$\begin{aligned}
\texttt{Var} \ni\ & x, y, z ::= \ x \mid y \mid \ \ldots \\
\texttt{Term} \ni\ & t, u ::= \ x \mid \texttt{yes} \mid \texttt{no} \mid \lambda x.t \mid t\ u \\
\texttt{Type} \ni\ & A, B ::= \ \texttt{Ans} \mid A \to B \\
\texttt{Context} \ni\ & \Gamma, \Delta ::= \ \varnothing \mid \Gamma, x : A
\end{aligned}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash \texttt{yes} : \texttt{Ans}} \qquad \frac{}{\Gamma \vdash \texttt{no} : \texttt{Ans}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

这里的 `Ans` 类型很像布尔类型 `Bool`。唯一的区别是它无法用 `if` 来消去。它的作用是拿来观测一段程序的求值：是否停机、结果是 `yes` 还是 `no`？动态语义上，采用 call-by-value 的 $\beta$-reduction：

$$\begin{aligned}
\texttt{Value} \ni\ & v, w ::= \ \texttt{yes} \mid \texttt{no} \mid \lambda x.t \\
\texttt{EvalContext} \ni\ & E ::= \ \square \mid E\ t \mid v\ E
\end{aligned}$$

$$E[(\lambda x.t)\ u] \rightsquigarrow E[t[x \mapsto u]]$$

其中 substitution $t[x \mapsto u]$ 是标准的 capture-avoiding substitution。定义出 STLC 的语法和类型系统和语义之后，我们希望证明它具有的一些好的性质。通过直接归纳，我们可以证明“有类型的程序不会出错”。具体来说，我们可以证明 *progress* 定理，即不会出现形如 `yes` $t$ 的、无法求值的坏的程序：

**Theorem** (progress)**.** 对于任意的 $\Gamma \vdash t : A$，要么 $t \in \texttt{Value}$，要么 $t$ 还能继续求值，即存在 $t'$ 使得 $t \rightsquigarrow t'$

我们可以证明 *subject reduction* 定理，即求值尊重类型：

**Theorem** (subject reduction)**.** 如果 $\Gamma \vdash t : A$ 且 $t \rightsquigarrow t'$，那么 $\Gamma \vdash t' : A$

上述定理都可以通过对表达式/求值关系/类型规则的简单归纳证出（使用一些同样简单归纳即可证出的辅助引理）。但进一步地，我们还希望证明 STLC 是**停机**的：

**Theorem** (termination). 对于任意的 $\varnothing \vdash t : A$，存在 $v$ 使得 $t \rightsquigarrow^* v$

这里 $\rightsquigarrow^*$ 表示零次或多次的 $\rightsquigarrow$。

## 1.1 一次失败的尝试：直接归纳

为了证明 STLC 的停机性，一个很自然的思路是继续对表达式/类型规则进行归纳。然而，这一尝试会失败。考虑 $t\ u$ 停机的证明。假设我们加强定理的表述，使得所有函数都会求值到 $\lambda$，那么我们可以得到：

- $t \rightsquigarrow^* \lambda x.t'$
- $u \rightsquigarrow^* v$

接下来，我们需要证明存在 $w$ 使得 $t'[x \mapsto v] \rightsquigarrow^* w$。但无论对类型规则归纳还是对表达式归纳，都无法证明这一点。对表达式的大小归纳也不可行，因为 $v$ 可能会被复制多份，$t'[x \mapsto v]$ 可能比 $t\ u$ 自身更大。

那么，问题出在哪呢？答案是：我们证明的定理**太弱**了。由于我们在使用归纳法证明定理，定理本身会作为归纳假设用于证明中。如果定理太弱，归纳假设就会太弱，从而导致证明无法完成。这里，“停机”作为归纳假设太弱了。在证明 $t\ u$ 的停机性时，我们需要 $t$ 能求值到一个值 $v$ 更多的性质。$v$ 不仅要是一个值，它还必须是一个“好”的值。例如，下面的值就是“不好”的：

$$v_{\text{bad}} = \lambda y.(\lambda x.x\ x)\ (\lambda x.x\ x)$$

如果 $t\ u$ 中的 $t$ 求值到了 $v_{\text{bad}}$，那么 $t\ u$ 就会不停机。这表明，加强定理、要求每个表达式求值到“好”的值是必要的。

## 1.2 使用 logical relation 证明停机性

那么，什么样的值是“好”的呢？从上面的讨论中我们知道，函数类型 $A \to B$ 的值需要具有一些额外的性质。我们希望它们在被提供好的参数时，也能停机并得到好的结果。这里就是 logical relation 出场的地方：我们可以用一个 logical relation（这里是 unary logical relation/predicate）$\mathcal{V}_A(v)$ 或者 $v \in \mathcal{V}_A$ 来表示 $v$ 是一个类型为 $A$ 的“好”的值。接下来，我们只需要证明以下定理即可（称为 fundamental theorem of logical relation）：

**Theorem.** 对于任意的 $\varnothing \vdash t : A$，存在 $v$ 使得 $t \rightsquigarrow^* v$ 且 $v \in \mathcal{V}_A$

根据定义，显然只要证出了 fundamental theorem of logical relation，也就证出了停机性。接下来是关键的 logical relation $\mathcal{V}_A$。根据上面的讨论，它可以如此定义：

- $\mathcal{V}_{\texttt{Ans}} = \{\texttt{yes}, \texttt{no}\}$。Ans 类型的表达式只需要满足停机性即可
- $\mathcal{V}_{A\to B} = \{v_f \mid \forall v_a \in \mathcal{V}_A, \exists w \in \mathcal{V}_B, v_f\ v_a \rightsquigarrow^* w\}$。一个“好”的 $A \to B$ 的值 $v_f$，在被提供一个“好”（$v_a \in \mathcal{V}_A$）的参数 $v_a$ 时，能够停机并得到一个“好”（$w \in \mathcal{V}_B$）的结果 $w$

为了更好的演示这个 logical relation 的意义，考虑类型 $A = \texttt{Ans} \to \texttt{Ans} \to \texttt{Ans}$。通过展开定义，可以看到 $R_A(v_f)$ 意味着：

> 对于任意的 $v_1, v_2 \in \{\texttt{yes}, \texttt{no}\}$,$v_f\ v_1\ v_2 \rightsquigarrow^* \texttt{yes}$ 或 $v_f\ v_1\ v_2 \rightsquigarrow^* \texttt{no}$

## 1.3 证明 fundamental theorem of logical relation

构造出 logical relation $\mathcal{V}_A(v)$ 后，就只需要证明 fundamental theorem of logical relation 了。然而，证明 $\varnothing \vdash \lambda x.t : A \to B$ 满足 $\mathcal{V}_{A\to B}$ 时，我们需要利用 $t$ 的性质，而 $t$ 是一个 open term！所以，我们首先需要把 fundamental theorem of logical relation 从 closed term 拓展到 open term。一个 open term $\Gamma \vdash t : A$ 需要满足的性质和一个函数 $\Gamma \to t$ 类似：只要将 $\Gamma$ 中的每个变量替换为“好”的 value，$t$ 就应该能停机并计算出一个“好”的值。为此，首先我们需要表达出“将 $\Gamma$ 中的每个变量替换为一个好的 value”的操作：

- 一个 substitution $\sigma$ 是一个 $\texttt{Var} \rightharpoonup \texttt{Value}$ 的 partial function。它是在 $\beta$-reduction 中用到的 capture-avoiding substitution 的多变量版本
- 我们用 $\sigma, x \mapsto v$ 表示往 $\sigma$ 中加入一个新的 $x \mapsto v$ 的映射得到的 substitution
- 我们把 logical relation $\mathcal{V}_A$ 拓展到 context 和 substitution 上：$\sigma \in \mathcal{S}_\Gamma$ 意味着对于任意的 $(x : A) \in \Gamma$，$x[\sigma] \in \mathcal{V}_A$

利用这一定义，就可以写出支持 open term 的 fundamental theorem of logical relation 了：

**Theorem 1.1.** 对于任意的 $\Gamma \vdash t : A$ 和 $\sigma \in \mathcal{S}_\Gamma$，存在 $v \in \mathcal{V}_A$ 使得 $t[\sigma] \rightsquigarrow^* v$

*Proof.* 对 $\Gamma \vdash t : A$ 进行归纳：

- 如果 $t = x$，那么 $(x : A) \in \Gamma$。$x[\sigma]$ 必定是一个值，所以停机性无需证明。根据 $\mathcal{S}_\Gamma$ 的定义，$x[\sigma] \in \mathcal{V}_A$
- 如果 $t = \texttt{yes}$ or $t = \texttt{no}$，此时 $t$ 自身就是值，且根据定义 $t \in \mathcal{V}_{\texttt{Ans}}$ 显然成立
- 如果 $t = \lambda x.t'$，$A = A' \to B$，那么有 $\Gamma, x : A' \vdash t' : B$。由于 $v_f = t[\sigma] = (\lambda x.t')[\sigma]$ 自身就是一个值，停机部分无需证明。只需要证明 $v_f$ 是一个“好”的值即可。

  对于任意的 $v_a \in \mathcal{V}_{A'}$，有 $(\sigma, x \mapsto v_a) \in \mathcal{S}_{\Gamma, x:A'}$。所以，根据归纳假设，存在 $v \in \mathcal{V}_B$ 使得 $t'[\sigma, x \mapsto v_a] \rightsquigarrow^* v$。然而，根据 $\beta$-reduction，$v_f\ v_a = (\lambda x.t')[\sigma]\ v_a \rightsquigarrow t'[\sigma, x \mapsto v_a]$ （和 substitution 的交互很容易通过归纳法另外证明）。所以 $v_f\ v_a \rightsquigarrow^* v$。根据定义，$v_f \in \mathcal{V}_{A' \to B}$ 成立
- 如果 $t = t_1\ t_2$，那么 $\Gamma \vdash t_1 : A' \to A$，$\Gamma \vdash t_2 : A'$。根据归纳假设，$t_1 \rightsquigarrow^* v_1$ 且 $v_1 \in \mathcal{V}_{A' \to A}$，$t_2 \rightsquigarrow^* v_2$ 且 $\mathcal{V}_{A'}(v_2)$。根据 $\mathcal{V}_{A' \to A}$ 的定义，存在 $v \in \mathcal{V}_A$ 使得 $v_1\ v_2 \rightsquigarrow^* v$。由于 $t \rightsquigarrow^* v_1\ v_2 \rightsquigarrow^* v$，待证的命题成立

□

## 1.4 semantic type：直接用语义定义类型

在本节中，我们的证明的结构是：

1. 先定义类型系统
2. 再根据类型定义一个 logical relation，表达我们希望各类型的表达式/值的性质
3. 最后证明 fundamental theorem of logical relation，说明类型正确的表达式都有好的性质

这是一种**语法先行**的做法：我们先定义了类型系统的语法（类型本身的语法和类型规则），再证明定义出的类型系统具有好的语义性质。事实上，有一种倒过来的定义方式：直接利用语义定义类型。在这种定义方式，所

有类型正确的表达式**自动**享有好的语义性质，无需证明！

那么，如何直接用语义定义类型呢？首先，我们把类型看作集合：一个类型 $A$ 是一个集合 $\mathcal{A} \subseteq \texttt{Value}$，里面装着满足 $A$ 的性质的值。接下来，我们可以定义各种类型构造。例如，我们可以定义**语义上的**函数类型：

$$\mathcal{A} \to \mathcal{B} = \{v_f \mid \forall v_a \in \mathcal{A}, \exists w \in \mathcal{B}, v_f\ v_a \rightsquigarrow^* w\}$$

可以看到，这其实就是函数类型的 logical relation。只不过这里我们直接把它作为了函数类型的定义。这么一来，所有 $v \in (\mathcal{A} \to \mathcal{B})$ 都自动是一个“好”的函数，无需另外证明。我们可以类似地将 context $\Gamma$ 定义为一个 substitution 的集合，记作 $\mathit{\Gamma}$。这么一来，open term 的类型 $\mathit{\Gamma} \vDash \mathcal{A}$ 就可以定义成：

$$(\mathit{\Gamma} \vDash \mathcal{A}) = \{t \mid \forall \sigma \in \mathit{\Gamma}, \exists v \in \mathcal{A}, t[\sigma] \rightsquigarrow^* v\}$$

天下没有免费的午餐。我们花了不少精力证明的 fundamental theorem of logical relation 不可能仅仅只是换种定义方式就免费了。那么，证明的工作量转移到了哪里呢？到目前为止，我们通过语义直接定义出了各种类型构造。但是，我们还没有**类型规则**，难以检查一个表达式是否有正确的类型！所以，我们需要重新加入类型规则。但如何证明类型规则和基于语义的类型定义是一致的呢？这就是需要证明的地方。**类型规则从公理变成了定理**。从语言设计的一部分，变成了可以证明的性质。例如，函数的类型规则：

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

会对应这样的一条定理：

$$\forall t \in (\mathit{\Gamma}, x : \mathcal{A} \vDash \mathcal{B}), (\lambda x.t) \in (\mathit{\Gamma} \vDash \mathcal{A} \to \mathcal{B})$$

我们需要证明每条类型规则都是一条合法的定理。但证明完毕之后，我们就重新得到了 fundamental theorem of logical relation：由于每条类型规则都是正确的，我们利用这些类型规则推导出的结果也是正确的。所以每个根据类型规则有类型 $\mathcal{A}$ 的表达式 $t$，都确实满足 $t \in \mathcal{A}$、有好的性质。

# 2 从 predicate 到 relation：证明 observational equivalence

上面我们看到了，如何用 unary logical relation 来联系类型和语义，证明 STLC 的停机性。但是，如果只是使用 unary logical relation，那么“relation”这个名字似乎就没什么必要了：直接称为“predicate”或者“集合”更加合适。事实上，除了 unary logical relation，binary logical relation 也有非常重要的应用。它是用来研究带副作用的程序的行为的非常好用的工具。本节就将用一个简单的例子来展示这一点。

## 2.1 副作用与 observational equivalence

两个程序在什么时候是等价的？对于简单的语言，例如 STLC，使用语义中的等式就足够了。但对于更复杂的语言，例如带副作用的语言，语义中的等式就不够了。例如，考虑下面这两段程序：

- `1`
- `let r = ref 2 in r := 1; !r`

从语义本身来看，它们并不等价：虽然最终的结果都是 `1`，但第二段程序会对可变的 reference 进行修改，产生副作用。但是，直觉告诉我们，这两段程序是等价的。因为第二段程序修改的 reference 是它自己创建的，也无法逃逸出去。理论上，我们没法观测出这两段程序的行为的不同之处（假设内存大小无限）。

那么，应该如何表示“两段程序的行为的差别无法观测”呢？事实上，这种基于语言内部的观测的程序等价叫作 observational equivalence。为了定义 observational equivalence，我们首先要在语言内部选定一个用来表示“观测的结果”的类型。这个类型要有至少两个不一样的值，而且最好能很容易地判断它的值是否相等。前一节中的 `Ans` 类型就是一个最简单的例子。除此之外，布尔类型 `Bool` 和自然数类型 `nat` 也是很常见的选择。

选定了观测的结果之后，我们需要定义什么是一次“观测”。“观测”一段程序，就是用它计算出一个观测结果。所以，我们可以把观测定义为一个 observation context $C[]$，它是一个带洞的表达式：

$$\texttt{ObsContext} \ni C[] ::= \ \square \mid \lambda x.C \mid C\ t \mid t\ C \mid \texttt{set}\ C$$

我们还需要定义 $C[]$ 的类型。我们用 $\Gamma \vdash C[\Delta \vdash A] : B$ 表示：

$$\forall(\Delta \vdash t : A), \Gamma \vdash C[t] : B$$

其中，$\Delta$ 和 $A$ 是被观测的程序依赖的 free variable 和类型。$C[t]$ 表示把表达式 $t$ 插入到 $C[]$ 的洞里，得到一个完整的表达式。$\Gamma$ 和 $B$ 是在 $C[]$ 中插入类型正确的表达式后，得到的完整表达式的 context 和类型。现在，我们可以定义出 observational equivalence $t \simeq u$ 了。假设 `Ans` 是表示观测结果的类型，且 $\Gamma \vdash t, u : A$，那么：

$$\begin{aligned} t \simeq u = &\forall(\varnothing \vdash C[\Gamma \vdash A] : \texttt{Ans}), \\ &C[t] \rightsquigarrow^* v \Leftrightarrow C[u] \rightsquigarrow^* v \end{aligned}$$

这里，evaluation context $C$ 是任意的，说明 $t \simeq u$ 要求**任何**观测都无法区分 $t$ 和 $u$。此外，这个定义是把停机性纳入了考量的。如果 $C[t]$ 和 $C[u]$ 中的任何一方停机，那么另一方也必须停机，而且必须给出相同的结果。反之，如果任何一方不停机，另一方也必须不停机。

值得注意的是，这里对 $C[t]$ 和 $C[u]$ 除了不停机以外的其他副作用没有做出等价性的要求。这是因为，副作用上的不同可以直接通过一个更大的 evaluation context 来观测。例如，考虑 $t = 1$ 和 $u =$ `r := 2; 1`，以 `nat` 为观测结果。那么，可以通过 context：

$$C[] = \texttt{let r = ref 1 in []; !r}$$

来观测出两者的不同：$C[t] \rightsquigarrow^* 1$，而 $C[u] \rightsquigarrow^* 2$。

## 2.2 observational equivalence 与 logical relation

虽然 observational equivalence 的定义非常简单也符合直觉，当待证明的等价性无法通过 operational semantic 简单地证明时（例如前面提到的例子 $1 \simeq$ `let r = ref 2 in r := 1; !r`），直接通过定义证明 observational equivalence 需要对**任意** observation context $C[]$ 进行讨论，而这是非常困难的。所以，为了证明 observational equivalence，我们需要更加强大的工具。而这就是 logical relation 出场的地方。

直接证明 observational equivalence 的难点在于定义中“任意 $C[]$”的部分。所以，我们希望能用一个等价的、但更容易证明的关系来代替 observational equivalence。也就是说，我们要用 `Term` 上的一个 binary *logical*

*relation* $R(t,u)$ 来代替 observational equivalence。在实践中，这个 logical relation 往往会对各个类型分别定义，记作 $R_A(t,u)$。

那么, logical relation $R_A(t,u)$ 应当满足什么样的性质呢? 首先, 它当然应该和 observational equivalence 等价。但除此之外, 它作为一个等价关系还需要易于证明。所以，$R_A(t,u)$ 的定义应当是**局部**的：它只看类型 $A$ 和表达式 $t$、$u$, 而不像 observational equivalence 那样涉及**全部** observation context $C[]$。

但是，一个**局部**的 logical relation，怎么能和**全局**的 observational equivalence 等价呢? 为此，我们需要证明以下两点：

1. 如果 $R_{\texttt{Ans}}(t,u)$，那么 $t \rightsquigarrow^* v \Leftrightarrow u \rightsquigarrow^* v$
2. 如果 $R_A(t,u)$，$\varnothing \vdash C[\varnothing \vdash A] : B$，那么 $R_B(C[t], C[u])$

如果一个 logical relation 满足以上两点，那么它就蕴含 observational equivalence：如果 $R_A(t,u)$ 成立, 那么对于类型合适的 $C[]$, 根据上面的命题 2,$R_{\texttt{Ans}}(C[t], C[u])$。根据上面的命题 1,这又意味着 $C[t] \rightsquigarrow^* v \Leftrightarrow C[u] \rightsquigarrow^* v$。由于 $C[]$ 是任意的，$t \simeq u$ 成立。

## 2.3 一门简单的带副作用的语言

为了演示如何使用 logical relation 证明 observational equivalence，这里使用一门简单的带副作用的语言作为例子。它在 STLC 的基础上加入了自然数，以及一个全局的、可变的自然数计数器，相当于一个全局的 reference cell。它的语法如下：

$$
\begin{aligned}
\mathbb{N} \ni\ & m, n \\
\texttt{Term} \ni\ & t, u ::= x \mid n \mid \lambda x.t \mid t\ u \mid \texttt{get} \mid \texttt{set t} \\
\texttt{Type} \ni\ & A, B ::= \texttt{nat} \mid A \to B \\
\texttt{Context} \ni\ & \Gamma, \Delta ::= \varnothing \mid \Gamma, x : A
\end{aligned}
$$

它的类型规则包含了 STLC 的全部规则（见第一节）。额外的规则是：

$$
\frac{}{\Gamma \vdash n : \texttt{nat}} \qquad \frac{}{\Gamma \vdash \texttt{get} : \texttt{nat}} \qquad \frac{\Gamma \vdash t : \texttt{nat}}{\Gamma \vdash \texttt{set}\ t : \texttt{nat}}
$$

它的 operational semantic 是标准的带状态的 operational semantic。这里由于只有一个全局的计数器，所以状态就是一个自然数。求值顺序依然

是 call-by-value：

$$\begin{aligned} \texttt{Value} \ni \quad v, w &::= \quad n \mid \lambda x.t \\ \texttt{EvalContext} \ni \quad E &::= \quad \square \mid E\ t \mid v\ E \mid \texttt{set}\ E \end{aligned}$$

$$\begin{aligned} (n, E[(\lambda x.t)\ u]) &\rightsquigarrow (n, E[t[x \mapsto u]]) \\ (n, E[\texttt{get}]) &\rightsquigarrow (n, E[n]) \\ (n, E[\texttt{set}\ m]) &\rightsquigarrow (m, E[m]) \end{aligned}$$

我们依然用 $\rightsquigarrow^*$ 来表示 $\rightsquigarrow$ 重复零次或多次。由于 call-by-value 的求值顺序，我们可以将顺序执行 `let x = t in u` 表示成 $(\lambda x.u)\ t$。下面的例子中将会用到这一语法糖。这门语言同样具有 progress 和 subject reduction 的性质，这里不做赘述。用和上一节一样的方法，可以证明这里的语言是停机的。下面将会利用这一事实简化定义和证明。本节中，接下来 observational equivalence 中的“观测结果”类型都默认为 `nat`。这门简单语言中的 observational equivalence 的完整定义如下。对于任意的 $\Gamma \vdash t, u : A$：

$$\begin{aligned} t \simeq u = &\forall (\varnothing \vdash C[\Gamma \vdash A] : \texttt{nat}), n, \\ &(n, C[t]) \rightsquigarrow (m_1, v) \text{ iff } (n, C[u]) \rightsquigarrow (m_2, v) \end{aligned}$$

注意这里不需要要求副作用相等：最终状态 $m_1$ 和 $m_2$ 可以不相等。得益于“任意 $C[]$”的灵活性，我们可以用 observation context `let x = □ in get` 来测试两段程序的副作用是否相等。

## 2.4 定义 observational equivalence 的 logical relation

现在，我们可以开始讨论本节的主题了：如何找到合适的 logical relation 来代替 observational equivalence。这里没有通用的构造方法，但直觉的定义往往就是正确答案。例如，对于函数类型上的 logical relation，我们希望它能蕴含“对任何等价的值，给出等价的结果”。我们首先尝试定义 `Value` 上的 logical relation $\mathcal{V}_A(v, w)$：

- 对于 `nat`，$\mathcal{V}_{\texttt{nat}} = \{(n, n) \mid n \in \mathbb{N}\}$：只有相等的自然数值才等价
- 对于 $A \to B$，我们以“对等价的输入得到等价的输出”为原则：

$$\mathcal{V}_{A \to B} = \{(v_f, w_f) \mid \forall (v_a, w_a) \in \mathcal{V}_A, (v_f\ v_a, w_f\ w_a) \in R_A\}$$

注意在 $\mathcal{V}_{A\to B}$ 的定义的最后，$v_f\ v_a$ 和 $w_f\ w_a$ 不是值，而是表达式！但我们不能直接要求 $v_f\ v_a$ 和 $w_f\ w_a$ 产生等价的值，因为它们可能会产生副作用，而这些副作用也必须等价。所以，我们必须同时定义表达式上的 logical relation $R_A$。如果 $\varnothing \vdash t, u : A$：

$$R_A = \{(t,u) \mid \forall n, (n,t) \rightsquigarrow^* (m_1, v_1) \text{ and } (n,u) \rightsquigarrow^* (m_2, v_2) \\ \Rightarrow m_1 = m_2 \text{ and } (v_1, v_2) \in \mathcal{V}_A\}$$

这里利用了语言停机的性质。如果语言可能不停机，那么还需要要求 $t$ 和 $u$ 对任意初始状态都具有相同的停机状态。直觉上，$R_A(t,u)$ 要求对于任意的初始状态 $n$，$t$ 和 $u$ 产生相同的副作用且计算出等价的结果。

## 2.5 fundamental theorem of *binary* logical relation

类似于证明 STLC 停机时的 predicate/unary logical relation，binary logical relation 也有自己的 fundamental logical relation：

**Theorem.** $\forall(\varnothing \vdash t : A), (t,t) \in R_A$

乍看之下，这条定理似乎并不是很有价值：在 observational equivalence 里，$t \simeq t$ 是显然成立的。但是，在 logical relation 里，$(t,t) \in R_A$ 有着更丰富的含义。例如，为了证明 $(v,v) \in \mathcal{V}_{A\to B}$，我们需要证明对于任意的 $(v_a, w_a) \in \mathcal{V}_A$，有 $R_B(v\ v_a, w\ w_a)$：这里的 $v_a$ 和 $w_a$ 不再是相等的，而仅仅是等价的了！所以，fundamental theorem of (binary) logical relation 告诉我们的其实是：每个表达式都以某种方式（由它的类型决定）尊重 logical relation 自身。

为了证明这里的 fundamental theorem of logical relation，类似于 STLC 的停机证明，我们需要把定义拓展到 open term。首先，我们定义 substitution 上的 logical relation：

$$R_\Gamma = \{(\sigma, \delta) \mid \forall (x : A) \in \Gamma\ (x[\sigma], x[\delta]) \in \mathcal{V}_A\}$$

接下来，可以定义 open term 上的 logical relation。如果 $\Gamma \vdash t, u : A$，那么：

$$R_A^\Gamma = \{(t,u) \mid \forall (\sigma, \delta) \in R_\Gamma, (t[\sigma], u[\delta]) \in R_A\}$$

现在，我们可以证明完整的 fundamental theorem of logical relation 了：

**Theorem 2.1.** $\forall(\Gamma \vdash t : A), (t,t) \in R_A^\Gamma$

*Proof.* 对 $t$ 进行归纳。假设有满足 $(\sigma, \delta) \in R_\Gamma$：

- $t = x$，那么 $(x : A) \in \Gamma$。根据 $R_\Gamma$ 的定义显然
- $t = \lambda x.t'$，那么 $A = A' \to B$，$\Gamma, x : A' \vdash t' : B$。由于 $t$ 已经是一个值了，只需要证明它满足 `Value` 上的 logical relation 即可。对于任意满足 $\mathcal{V}_{A'}(v_a, w_a)$ 的 $v_a$，$w_a$，有 $((\sigma, x \mapsto v_a), (\delta, x \mapsto w_a)) \in R_{\Gamma, x:A'}$。根据归纳假设，$(t'[\sigma, x \mapsto v_a], t'[\sigma, x \mapsto w_a]) \in R_B$。根据 $\beta$-reduction，这意味着 $(t[\sigma]\ v_a, t[\sigma]\ w_a) \in R_B$。所以根据定义，$(t[\sigma], t[\delta]) \in \mathcal{V}_{A' \to B}$
- $t = t_f\ t_a$，那么 $\Gamma \vdash t_f : A' \to A$，$\Gamma \vdash t_a : A'$。对于任意的初始状态 $n$，根据对 $t_f$ 的归纳假设，$(n, t_f[\sigma]) \rightsquigarrow^* (m, v_f)$，$(n, t_f[\delta]) \rightsquigarrow^* (m, w_f)$，且 $(v_f, w_f) \in \mathcal{V}_{A' \to A}$。根据对 $t_a$ 的归纳假设，$(m, t_a[\sigma]) \rightsquigarrow^* (m', v_a)$，$(m, t_a[\delta]) \rightsquigarrow^* (m', w_a)$，且 $(v_a, w_a) \in \mathcal{V}_{A'}$。根据 $\mathcal{V}_{A' \to A}$ 的定义，有 $R_A(v_f\ v_a, w_f\ w_a)$。所以 $R_A((t_f\ t_a)[\sigma], (t_f\ t_a)[\delta])$ 成立
- $t = \texttt{get}$，$A = \texttt{nat}$。对于任意的初始状态 $n$，$(n, t[\sigma]) \rightsquigarrow (n, n)$，$n$，$(n, t[\delta]) \rightsquigarrow (n, n)$。根据定义，$(n, n) \in \mathcal{V}_{\texttt{nat}}$。所以根据定义 $(\texttt{get}, \texttt{get}) \in R_{\texttt{nat}}$
- $t = \texttt{set}\ t'$，$A = \texttt{nat}$。对于任意的初始状态 $n$，根据归纳假设，$(n, t'[\sigma]) \rightsquigarrow (m, v)$，$(n, t'[\delta]) \rightsquigarrow (m, w)$，且 $(v, w) \in R_{\texttt{nat}}$。根据定义这意味着 $v = w$。而 $(m, \texttt{set}\ v) \rightsquigarrow (v, v)$，$(m, \texttt{set}\ w) \rightsquigarrow (w, w)$。所以根据定义 $(\texttt{set}\ t', \texttt{set}\ t') \in R_{\texttt{nat}}$

□

## 2.6 证明 logical relation 与 observational equivalence 等价

虽然我们证明了 fundamental theorem of logical relation，但我们的最终目的是证明 $(t, u) \in R_A^\Gamma$ 和 $t \simeq u$ 等价。为此，fundamental theorem of logical relation 是不可或缺的。首先，我们证明 $(t, u) \in R_A^\Gamma$ 蕴含 $t \simeq u$。参照前面讨论过的思路，我们证明如下两个引理：

**Lemma 2.2.** 如果 $(t, u) \in R_{\texttt{nat}}$，那么对于任意的 $n$，$(n, t) \rightsquigarrow^* (m_1, v) \Leftrightarrow (n, u) \rightsquigarrow^* (m_2, v)$

*Proof.* 根据 $R_{\texttt{nat}}$ 的定义，$(n, t) \rightsquigarrow^* (m, v_1)$，$(n, u) \rightsquigarrow^* (m, v_2)$，且 $(v_1, v_2) \in \mathcal{V}_{\texttt{nat}}$。根据 $\mathcal{V}_{\texttt{nat}}$ 的定义，$v_1 = v_2$。所以目标引理成立。 □

**Lemma 2.3.** $\forall(\Delta \vdash t, u : A), (\Gamma \vdash C[\Delta \vdash A] : B), R_A^{\Delta}(t, u) \Rightarrow R_B^{\Gamma}(C[t], C[u])$

*Proof.* 如果要严谨地证明, 需要像 fundamental theorem 的证明一样对 $C$ 进行归纳。其中, 对 $C = C'\ t$ 和 $C = t\ C'$ 的情况, 需要对 $t$ 使用 fundamental theorem of logical relation 来完成证明。这里由于空间原因, 只给出一个不严谨的简单证明。

我们可以把每个 $C[t]$ 写成 $(\lambda x.C[x\ \vec{t}])\ (\lambda\Delta.t)$。其中 $\vec{t}$ 是 $t/u$ 依赖的、$\Delta$ 中的 free variable 在 $C[]$ 中的定义（由于 $C[]$ 中只有一个“洞”, 不需要担心 $\vec{t}$ 中的表达式被反复求值的副作用问题）。令 $v_f = \lambda x.C[x\ \vec{t}]$, 根据 fundamental theorem of logical relation, $(v_f, v_f) \in \mathcal{V}^{\Gamma}_{(\Delta\to A)\to B}$。根据定义, $(\lambda\Delta.t, \lambda\Delta.u) \in \mathcal{V}_{\Delta\to A}$。所以 $(v_f\ (\lambda\Delta.t), v_f\ (\lambda\Delta.u)) \in R_B^{\Gamma}$ □

**Corollary 2.4** (soundness)**.** 对于任意的 $\Gamma \vdash t, u : A$, 如果 $(t, u) \in R_A^{\Gamma}$, 那么 $t \simeq u$

为了证明 logical relation 和 observational equivalence 等价, 我们还希望证明 *completeness*:

**Theorem** (completeness)**.** $\forall(\Gamma \vdash t, u : A), t \simeq u \Rightarrow (t, u) \in R_A^{\Gamma}$

*Proof.* 由于空间原因这里不做证明。提示:

- 先单独证明 $\Gamma = \varnothing$ 时的 closed term 版本。证明的方法是对类型 $A$ 进行归纳。对于 $R_A/\mathcal{V}_A$ 的不同要求, 构造不同的 observation context $C[]$ 来证明
- 再证明一条引理: 如果 $t \simeq u$, $(\sigma, \delta) \in R_{\Gamma}$, 那么 $t[\sigma] \simeq u[\delta]$。证明方法同样是对 $t[\sigma] \simeq u[\delta]$ 定义中的 $C[]$ 进行归纳
- 利用 closed term 的 completeness 和上面的引理, 即可将 completenss 拓展到 open term

□

# 3 parametricity 其一：abstract type

在前面两节，我们看到了如何使用 logical relation 来证明类型系统和 operational semantic 的各种性质。然而，logical relation 的作用远不止于此。它还可以用来证明一些和 operational semantic 完全无关的类型系统性质。其中，*parametricity* 就是一个非常著名而重要的例子。接下来的两节中将会介绍 parametricity（和它的两个主要应用）。

## 3.1 abstract type

软件工程的一个重要原则是**隐藏实现**。在一个模块对外的接口中，把类型与函数的实际实现隐藏起来。在 ML 家族的语言中，往往用 module system 来实现这一点。例如，一个栈的数据结构，可以用如下的 module signature 来表示：

```
module type Stack = sig
    type 'a t

    val empty : 'a t
    val push  : 'a -> 'a t -> 'a t
    val pop   : 'a t -> 'a t
    val top   : 'a t -> 'a option
end
```

每个实现了 `Stack` 这个签名的 module 都可以被当作 `Stack` 使用。而且，使用 `Stack` 的代码无法获得具体的实现内部 `'a t` 的定义，也无法区分不同的实现。事实上，ML 家族的 module system 的这种应用，是 abstract type 的一个特例。一个 abstract type 包含：

- 一个类型 $\alpha$
- $\alpha$ 上的一些操作/函数

使用 abstract type 的程序可以自由地使用 abstract type 提供的操作，但无法看到 $\alpha$ 的内部实现。所以，使用 abstract type 的程序**只**能使用 abstract type 提供的操作。

Abstract type 具有比普通的类型更好的性质。如果一个 abstract type 的所有操作都尊重 $\alpha$ 上的某个性质 $P$，那么即是 $\alpha$ 的具体实现不一定满足

$P$，使用 abstract type 的程序构造出的 $\alpha$ 也一定满足 $P$。例如，我们可以用整数和 abstract type 来实现自然数：

```
module type Nat = sig
    type t
    val zero : t
    val succ : t -> t
end

module IntNat : Nat = struct
    type t = int

    let zero   = 0
    let succ n = n + 1
end
```

`IntNat` 的具体实现中，`t = int` 本身可能有负值。但使用 `Nat` 签名中的操作构造出来的数一定是非负的。使用 `Nat` 的程序无法看到 `t` 的具体实现、只能使用 `Nat` 提供的操作。因此它们构造出的数也一定是非负的。

但是，要如何保证 abstract type 满足这样的性质呢？如何保证程序无法用一些奇怪的方式“偷窥”到某个 abstract type 的具体实现？这就需要我们证明一条 *representation theorem*。这条定理告诉我们，使用 abstract type 的程序无法区分不同的具体实现。而 representation theorem 的表述如证明，就需要 logical relation 的出场。

## 3.2 支持 abstract type 的 STLC

为了演示如何使用 logical relation 表述和证明 representation theorem，本节将使用如下的演示语言。它在 STLC 的基础上加入了 abstract type。简单起见，我们不支持自定义的 abstract type 声明，把所有 abstract type 和

它们支持的操作用一个签名固定下来。STLC + abstract type 的语法如下：

$$
\begin{aligned}
\texttt{Var} \ni\ & x, y, z \\
\texttt{Op} \ni\ & f, g, h \\
\texttt{Term} \ni\ & t, u ::= x \mid f \mid \texttt{yes} \mid \texttt{no} \mid \lambda x.t \mid t\ u \\
\texttt{TypeVar} \ni\ & \alpha, \beta \\
\texttt{Type} \ni\ & A, B ::= \alpha \mid \texttt{Ans} \mid A \to B \\
\texttt{Context} \ni\ & \Gamma, \Delta ::= \varnothing \mid \Gamma, x : A
\end{aligned}
$$

我们往 STLC 中加入了一些类型变量（$\alpha$、$\beta$），这些类型变量表示不同的 abstract type。除此之外，我们还加入了 abstract type 对应的操作（$f$、$g$）。为了指明一段程序用到了哪些 abstract type，我们定义**签名** $\Sigma$。一个签名 $\Sigma$ 包含：

- 一个类型变量的集合，表示签名中定义了哪些 abstract type。Abstract type $\alpha$ 在签名 $\Sigma$ 中有定义记作 $\alpha \in \Sigma$
- 一个 $\texttt{Op} \to \texttt{Type}$ 的 partian function，记录了签名 $\Sigma$ 中有定义的操作对应的类型。操作 $f$ 在签名 $\Sigma$ 中有类型 $A$ 记作 $\Sigma(f) = A$

有了签名之后，我们就可以定义 STLC + abstract type 的类型规则了。首先，我们要求类型中的类型变量都必须在签名中有定义：

$$\frac{\alpha \in \Sigma}{\Sigma \vdash \alpha \text{ type}} \qquad \frac{}{\Sigma \vdash \texttt{Ans} \text{ type}} \qquad \frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash (A \to B) \text{ type}}$$

$$\frac{}{\Sigma \vdash \varnothing \text{ ctx}} \qquad \frac{\Sigma \vdash \Gamma \text{ ctx} \qquad \Sigma \vdash A \text{ type}}{\Sigma \vdash (\Gamma, x : A) \text{ ctx}}$$

表达式的类型规则如下：

$$\frac{\Sigma \vdash \Gamma \text{ ctx} \qquad (x : A) \in \Gamma}{\Sigma; \Gamma \vdash x : A} \qquad \frac{\Sigma \vdash \Gamma \text{ ctx} \qquad \Sigma(f) = A}{\Sigma; \Gamma \vdash f : A}$$

$$\frac{\Sigma \vdash \Gamma \text{ ctx} \qquad c \in \{\texttt{yes}, \texttt{no}\}}{\Sigma; \Gamma \vdash c : \texttt{Ans}}$$

$$\frac{\Sigma; \Gamma, x : A \vdash t : B}{\Sigma; \Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Sigma; \Gamma \vdash t : A \to B \qquad \Sigma; \Gamma \vdash u : A}{\Sigma; \Gamma \vdash t\ u : B}$$

由于签名 $\Sigma$ 在类型规则中是不会变化的，下面在书写类型推断时可能会把 $\Sigma$ 略去，将 $\Sigma;\Gamma \vdash t : A$ 写作 $\Gamma \vdash t : A$。STLC + abstract type 的操作语义和 STLC 没有区别，这里不再写出。有时，我们会希望讨论不包含任何 abstract type 的类型和表达式。我们用 $\texttt{Type}_0$ 表示不含任何类型变量的类型，用 $\texttt{Value}_0/\texttt{Term}_0$ 表示不包含任何操作 $f$ 的值/表达式。

## 3.3 abstract type 的实现

STLC + abstract type 中的 $\Sigma$ 提供了 abstract type 的签名，但语言中还没有实现 abstract type 的方式。首先，我们需要定义一个签名 $\Sigma$ 的**实现**。签名 $\Sigma$ 的一个实现 $\eta$（记作 $\eta : \Sigma$）是一个 partial function。对于每个 $\alpha \in \Sigma$，有一个集合 $\eta(\alpha) \subseteq \texttt{Value}_0$。利用 $\eta$，我们可以给所有 $\Sigma \vdash A$ type 的合法类型赋予语义：

$$\begin{aligned}
[\![\alpha]\!]_\eta &= \eta(\alpha) \\
[\![\texttt{Ans}]\!]_\eta &= \{\texttt{yes}, \texttt{no}\} \\
[\![A \to B]\!]_\eta &= \{v_f \mid \forall v_a \in [\![A]\!]_\eta, \exists w \in [\![B]\!]_\eta, v_f\ v_a \rightsquigarrow^* w\}
\end{aligned}$$

可以看到，这里的 $[\![A]\!]_\eta$ 也是一个（unary）logical relation，它在 $A \to B$ 上的定义就是证明 STLC 停机时的定义。类型的语义可以拓展到 context $\Gamma$：$[\![\Gamma]\!]_\eta$ 是一个 substitution 的集合：

$$[\![\Gamma]\!]_\eta = \{\sigma \mid \forall (x : A) \in \Gamma, x[\sigma] \in [\![A]\!]_\eta\}$$

接下来，我们将 $\eta : \Sigma$ 的定义拓展到操作上：

对于每个 $\Sigma(f) = A$，有 $\eta(f) \in [\![A]\!]_\eta$

利用拓展后的 $\eta$，可以给所有表达式赋予语义：

$$\begin{aligned}
[\![x]\!]_\eta &= x \\
[\![f]\!]_\eta &= \eta(f) \\
[\![\lambda x.t]\!]_\eta &= \lambda x.[\![t]\!]_\eta \\
[\![t\ u]\!]_\eta &= [\![t]\!]_\eta\ [\![u]\!]_\eta
\end{aligned}$$

最后,我们可以证明如下的 soundness 定理。它其实也是一条 fundamental theorem of logical relation：

**Theorem** (soundness)**.**

$$\forall (\Gamma \vdash t : A), (\eta : \Sigma), (\sigma \in [\![\Gamma]\!]_\eta), \exists v \in [\![A]\!]_\eta, [\![t]\!]_\eta[\sigma] \rightsquigarrow^* v$$

*Proof.* 和 Theorem 1.1 类似 □

## 3.4 用 logical relation 表达 representation theorem

我们定义了签名的实现。下面，该考虑如何表述并证明 representation theorem 了。我们希望证明，程序无法分辨一个 abstract type 的两个不同 representation。

首先，我们需要定义“无法分辨”。而上一节的 observational equivalence 就是一个很好的“无法分辨”的定义！参照 observational equivalence 的思路，用 `Ans` 作为“观测结果”。那么，representation theorem 就应该说明，任何类型为 `Ans` 的表达式，在 abstract type 的不同解释下都能得出相同的结果。所以，representation theorem 可以如此定义：

**Theorem** (representation)**.**

$$\forall(\eta_1, \eta_2 : \Sigma), (\varnothing \vdash t : \texttt{Ans}), [\![t]\!]_{\eta_1} \rightsquigarrow^* v \textit{ iff } [\![t]\!]_{\eta_2} \rightsquigarrow^* v$$

（其实这条定理过强了，所以并不成立。但后面会讨论如何修改它）

如果尝试直接用归纳法证明这条定理，会和证明 STLC 停机时一样、由于定理本身不够强而失败。所以，我们需要构造一个 logical relation 来加强这条定理、把它从 closed term 拓展到 open term 来完成证明。这里我们在讨论的是两个表达式之间的关系，所以我们要构造的是一个 binary logical relation $R_A$。证明的最后一步，就是证明 fundamental theorem of logical relation：

**Theorem.**

$$\forall(\eta_1, \eta_2 : \Sigma), (\Gamma \vdash t : A), ([\![t]\!]_{\eta_1}, [\![t]\!]_{\eta_2}) \in R_A^\Gamma$$

这条定理也常常被称为 abstraction theorem 或 parametricity。因为它告诉我们语言中的 abstract type 真的是“抽象的”，语言对 abstract type 的使用是“参数化”的，无法对 abstract type 的不同实现做出不同处理。

## 3.5 parametricity 的 logical relation

接下来，我们只需要构造出 logical relation $R_A$ 即可。但是，类型 $A$ 中可能包含 abstract type $\alpha$，而且 $\eta_1$ 和 $\eta_2$ 可能对 $\alpha$ 有不同的实现！那么，我们应该如何定义 $R_\alpha$ 呢?

直觉上，我们希望签名中的操作在不同实现中是“一样”的。也就是说，$R_\alpha$ 会连接签名中相同的操作（的语义)。但满足这一要求的 logical relation

可能有很多。为了说明 abstract type 的实现真的无关紧要，我们希望对于**任意**这样的 logical relation，都能证明 representation theorem。所以，我们首先需要定义一个“好”的 logical relation $R$ 长什么样。

事实上，有一种更简洁的方式可以定义“好”的 logical relation。在证明 STLC 停机时，logical relation $\mathcal{V}_A$ 其实有两重含义：

- $\mathcal{V}_A$ 可以看作一个逐类型定义的、`Value` 上的 predicate
- $\mathcal{V}_A$ 也可以看作一个 $\texttt{Type} \to \mathcal{P}(\texttt{Value})$ 的函数，其中 $\mathcal{P}$ 为取幂集的操作。这个函数可以看成给每个类型赋予一个集合/predicate 作为语义

对于一个值上的 binary logical relation $\mathcal{V}_A$，同样可以有两种解读：

- $\mathcal{V}_A$ 可以看作一个逐类型定义的、`Value` 上的 binary logical relation
- $\mathcal{V}_A$ 也可以看作一个 $\texttt{Type} \to \mathcal{P}(\texttt{Value} \times \texttt{Value})$ 的函数，这个函数可以看成给每个类型赋予一个 binary logical relation 作为语义

所以证明 representation theorem 需要的 binary logical relation $R_A$，可以看成一个给类型赋予语义的函数——就像定义 abstract type 的实现时用到的 $[\![A]\!]_\eta$ 一样。而为了给 abstract type 赋予语义，定义 $\mathcal{V}_A$ 时我们同样需要一个 $\Sigma$ 的实现。只不过，这个实现会把类型解释为 binary logical relation 而非值的集合！

这么一来，$\mathcal{V}_A$ 的定义就和 $[\![A]\!]_\eta$ 几乎一模一样了。首先，我们定义 $\Sigma$ 的一个（在 $\eta_1$、$\eta_2$ 之间的）（binary logical relation）实现 $\rho$：

> 如果 $\rho \in R_\Sigma(\eta_1, \eta_2)$，那么对于每个 $\alpha \in \Sigma$，都有 $\rho(\alpha) \subseteq \eta_1(\alpha) \times \eta_2(\alpha)$

给定一个 abstract type 的（binary logical relation）实现 $\rho$，我们可以定义所有类型的（binary logical relation）语义 $\mathcal{V}_A^\rho$（值）和 $R_A^\rho$（表达式）：

$$
\begin{aligned}
\mathcal{V}_A^\rho &\subseteq \texttt{Value} \times \texttt{Value} \\
\mathcal{V}_\alpha^\rho &= \rho(\alpha) \\
\mathcal{V}_{\texttt{Ans}}^\rho &= \{(\texttt{yes}, \texttt{yes}), (\texttt{no}, \texttt{no})\} \\
\mathcal{V}_{A \to B}^\rho &= \{(v_f, w_f) \mid \forall (v_a, w_a) \in \mathcal{V}_A^\rho, (v_f\ v_a, w_f\ w_a) \in R_B^\rho\}
\end{aligned}
$$

$$
\begin{aligned}
R_A^\rho &\subseteq \texttt{Term} \times \texttt{Term} \\
R_A^\rho &= \{(t, u) \mid \exists (v, w) \in \mathcal{V}_A^\rho, t \rightsquigarrow^* v \text{ and } u \rightsquigarrow^* w\}
\end{aligned}
$$

接下来，我们把 $R$ 的定义拓展到操作上。这对应于“好”的 logical relation 中，“签名中的操作会被联系在一起”的部分：

$$\text{对于每个 } \Sigma(f) = A,\ (\eta_1(f), \eta_2(f)) \in \mathcal{V}_A^\rho$$

例如，如果 $\Sigma$ 中有一个 abstract type $\alpha$ 和一个常量 $c : \alpha$，$\eta_1$ 和 $\eta_2$ 都把 $\alpha$ 解释为 $\texttt{Ans}$，但 $\eta_1(c) = \texttt{yes}$、$\eta_2(c) = \texttt{no}$，那么就应该有 $(\texttt{yes}, \texttt{no}) \in \rho(\alpha)$。总结来说，一个“好”的 logical relation $\rho \in R_\Sigma(\eta_1, \eta_2)$ 满足：

- 对于每个 $\alpha \in \Sigma$，有 $\rho(\alpha) \subseteq \eta_1(\alpha) \times \eta_2(\alpha))$
- 对于每个 $\Sigma(f) = A$，有 $(\eta_1(f), \eta_2(f)) \in \mathcal{V}_A^\rho$

其中，$\mathcal{V}_A^\rho$ 是把 $\rho$ 拓展到全体类型得到的 logical relation。

## 3.6 fundamental theorem of logical relation

现在，证明只差最后一步了：熟悉的 fundamental theorem of logical relation。在此之前，首先我们需要按照惯例，把 logical relation 拓展到 open term。类似于 observational equivalence 中的 logical relation，context 的语义是两个 substitution 之间的 logical relation：

$$R_\Gamma^\rho = \{(\sigma, \delta) \mid \forall (x : A) \in \Gamma, (x[\sigma], x[\delta]) \in \mathcal{V}_A^\rho\}$$

现在，我们可以开始证明 fundamental theorem of logical relation 了：

**Theorem 3.1.** 对于任意的 $\rho \in R_\Sigma(\eta_1, \eta_2)$，$\Gamma \vdash t : A$ 和 $(\sigma, \delta) \in R_\Gamma^\rho$，有 $(\llbracket t \rrbracket_{\eta_1}[\sigma], \llbracket t \rrbracket_{\eta_2}[\delta]) \in R_A^\rho$

*Proof.* 直接对 $t$ 进行归纳即可，和之前的两个 fundamental theorem 几乎没有区别。在 $t = f$ 的情况直接使用 $R \in \texttt{Rel}_\Sigma(\eta_1, \eta_2)$ 的定义即可 □

证明了 fundamental theorem of logical relation 后，我们可以证明如下的弱化版的 representation theorem：

**Theorem** (representation)**.**

$$\begin{aligned}&\forall (\eta_1, \eta_2 : \Sigma), (\varnothing \vdash t : \texttt{Ans}),\\ &\quad \textit{if there exists } \rho \in R_\Sigma(\eta_1, \eta_2),\\ &\quad \textit{then } \llbracket t \rrbracket_{\eta_1} \rightsquigarrow^* v \textit{ iff } \llbracket t \rrbracket_{\eta_2} \rightsquigarrow^* v\end{aligned}$$

*Proof.* 根据 Theorem 3.1 和 $\mathcal{V}_{\texttt{Ans}}^\rho$ 的定义 □

比起我们最初想要的 representation theorem，这里证明的版本多出了一个条件：$\eta_1$ 和 $\eta_2$ 之间要存在至少一个 relation $R \in \mathtt{Rel}_\Sigma(\eta_1, \eta_2)$。。但是，不是所有 $\eta_1$ 和 $\eta_2$ 之间都存在至少一个 relation。例如，考虑一个 abstract type $\alpha$，它有如下的操作：

$$c : \alpha \qquad\qquad f : \alpha \to \mathtt{Ans}$$

那么，不同的实现可以给 $(f\ c) : \mathtt{Ans}$ 赋予不同的值，从而使用这个 abstract type 的程序就能观测出两个实现的不同了！所以，要求两个实现是被 logical relation 连在一起是必要的、也是合理的。

虽然不受限制的 representation theorem 是不成立的，但是我们可以找到很多具体的例子，它们能够应用 fundamental theorem of logical relation。如果一个签名 $\Sigma$ 里没有任何操作，那么此时任何两个实现 $\eta_1$、$\eta_2$ 之间都可以联系起来：取 $\rho(\alpha) = \eta_1(\alpha) \times \eta_2(\alpha)$ 即可。这意味着，由于没有任何操作，$\alpha$ 中的任何值都是无法分辨的。更一般地，如果所有操作都形如 $A_1 \to \ldots \to A_n \to \alpha$，也就是我们利用签名中的操作无法观测一个 abstract type 的话，那么同样可以证明任何两个实现都是 related 的，从而证明任何两个实现都无法区分。

# 4 parametricity 其二：theorems for free

Abstract type 的 parametricity 帮助我们严谨地证明了 abstract type 真的是“抽象”的、不同的具体实现真的是无法分辨的。然而，abstract type 的 abstraction theorem 中依然提到了 logical relation，使得这条定理看上去离实践有一定距离。

事实上，parametricity 有更直接、更“语法”的应用。Abstraction theorem 是关于 abstract type 的。而当我们把 abstract type 替换成类型变量、往语言中加入 polymorphism，parametricity 能帮助我们证明多态程序的一些非常重要的性质。而且，这些性质**只需要看类型不需要看实现**就能证明：这正是“theorems for free”的含义。

## 4.1 一门简单的带 polymorphism 的语言

为了演示 parametricity 与 polymorphism 的结合，本节使用如下的语言：

$$
\begin{aligned}
\texttt{Var} \ni\ & x, y, z \\
\texttt{Op} \ni\ & f, g, h \\
\texttt{Term} \ni\ & t, u ::= x \mid \texttt{yes} \mid \texttt{no} \mid \lambda x.t \mid t\ u \\
\texttt{TypeVar} \ni\ & \alpha, \beta \\
\texttt{Type} \ni\ & A, B ::= \alpha \mid \texttt{Ans} \mid A \to B \mid \forall \alpha.A \\
\texttt{Context} \ni\ & \Gamma, \Delta ::= \varnothing \mid \Gamma, x : A
\end{aligned}
$$

语言的求值规则和 STLC 一样。类型规则如下：

$$\frac{\Sigma \vdash \Gamma\ \text{ctx} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Sigma \vdash \Gamma\ \text{ctx} \qquad c \in \{\texttt{yes}, \texttt{no}\}}{\Gamma \vdash c : \texttt{Ans}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

$$\frac{\Gamma \vdash t : A \qquad \alpha \notin \texttt{FTV}(\Gamma)}{\Gamma \vdash t : \forall \alpha.A} \qquad \frac{\Gamma \vdash t : \forall \alpha.A}{\Gamma \vdash t : A[\alpha \mapsto B]}$$

其中，类型上的 substitution $A[\alpha \mapsto B]$ 是标准的 capture-avoiding substitution。$\texttt{FTV}(X)$ 表示 $X$ 中的所有自由类型变量。这里的语言就是标准的

second-order lambda calculus（加上一个内建类型 `Ans`）。它具有 progress、subject reduction 与停机性。

## 4.2 构造 logical relation 其一：实例化类型变量

Parametricity 应该如何与 polymorphism 结合? 答案和前一节一样: 把类型解释为一个 binary logical relation, 然后 $\forall\alpha.A$ 则会对“任意合法的 binary logical relation”具有一些性质。所以, 我们可以直接开始复刻上一节的构造。

首先，我们需要定义如何在语义上“实例化”类型变量，将其替换为一个具体的类型。为此，我们将类型解释为“closed value 的集合”。令 $\texttt{Value}_0$ 为所有 closed value 的集合，令 $\eta : \texttt{TypeVar} \to \mathcal{P}(\texttt{Value}_0)$ 为一个从类型变量到 closed value 集合的函数（我们不关心的类型变量可以映射为空集。也可以更精确地指明每个 $\eta$ 的 domain，这里为了简单起见不这么做），那么:

$$
\begin{aligned}
[\![A]\!]_\eta &\subseteq \texttt{Value}_0 \\
[\![\alpha]\!]_\eta &= \eta(\alpha) \\
[\![\texttt{Ans}]\!]_\eta &= \{\texttt{yes}, \texttt{no}\} \\
[\![A \to B]\!]_\eta &= \{v_f \mid \forall v_a \in [\![A]\!]_\eta, \exists w \in [\![B]\!]_\eta v_f\ v_a \rightsquigarrow^* w\} \\
[\![\forall\alpha.A]\!]_\eta &= \{v \mid \forall V \subseteq \texttt{Value}_0, v \in [\![A]\!]_{\eta,\alpha\mapsto V}\}
\end{aligned}
$$

和前面几节类似, 我们把 $[\![A]\!]_\eta$ 从值的集合拓展到表达式的集合: $t \in [\![A]\!]_\eta$ 意味着 $\exists v \in [\![A]\!]_\eta, t \rightsquigarrow^* v$。这里，$\forall\alpha.A$ 的解释是比较有意思的部分。它的含义是：如果无论把 $\alpha$ 替换成什么类型（**任意的** $V \subseteq \texttt{Value}_0$），$v$ 都是好的，那么 $v$ 作为类型 $\forall\alpha.A$ 的值是好的。

这里的 $[\![A]\!]_\eta$ 同样是类型的一个 (unary) logical relation 解释。所以，可以证明如下的 fundamental theorem of logical relation/soundness：

**Theorem.** $\forall(\Gamma \vdash t : A), (\sigma \in [\![\Gamma]\!]_\eta), \exists v \in [\![A]\!]_\eta, t[\sigma] \rightsquigarrow^* v$

## 4.3 构造 logical relation 其二：type as binary logical relation

有了实例化类型变量的方法后,就可以开始重头戏:binary logical relation 的构造了。和前一节一样，我们采用“给每个类型赋予一个 binary logical

relation 作为其语义”的解读。对于两个 $\eta_1, \eta_2 \in \texttt{TypeVar} \to \mathcal{P}(\texttt{Value}_0)$，它们之间的 (binary) logical relation 定义为：

$$\rho \in R(\eta_1, \eta_2) \text{ iff } \forall \alpha, \rho(\alpha) \subseteq \eta_1(\alpha) \times \eta_2(\alpha)$$

也就是说，对每个类型变量 $\alpha$，$R$ 都提供了一个 logical relation，把 $\alpha$ 在 $\eta_1$、$\eta_2$ 中联系起来。下面，给定一个 $\rho \in R(\eta_1, \eta_2)$，我们可以解释任意类型、将它们在 $\eta_1$、$\eta_2$ 中的解释联系起来：

$$\begin{aligned}
\mathcal{V}^\rho_A &\subseteq [\![A]\!]_{\eta_1} \times [\![A]\!]_{\eta_2} \\
\mathcal{V}^\rho_\alpha &= \rho(\alpha) \\
\mathcal{V}^\rho_{\texttt{Ans}} &= \{(\texttt{yes}, \texttt{yes}), (\texttt{no}, \texttt{no})\} \\
\mathcal{V}^\rho_{A \to B} &= \{(v_f, w_f) \mid \forall (v_a, w_a) \in \mathcal{V}^\rho_A, (v_f\ v_a, w_f\ w_a) \in R^\rho_B\} \\
\mathcal{V}^\rho_{\forall \alpha.A} &= \{(v, w) \mid \forall V_1, V_2 \subseteq \texttt{Value}_0, \forall R_V \subseteq V_1 \times V_2, (v, w) \in \mathcal{V}^{\rho, \alpha \mapsto R_V}_A\}
\end{aligned}$$

$$\begin{aligned}
R^\rho_A &\subseteq \texttt{Term}_0 \times \texttt{Term}_0 \\
R^\rho_A &= \{(t, u) \mid \exists (v, w) \in \mathcal{V}^\rho_A, t \rightsquigarrow^* v \text{ and } u \rightsquigarrow^* w\}
\end{aligned}$$

同样地，我们把 $[\![A]\!]_R$ 从值的集合拓展到表达式的集合。这里，$(v, w) \in [\![\forall \alpha.A]\!]_R$ 的 binary logical relation 的含义如下：对于任何联系在一起（$R_V$）的类型实例（$V_1$、$V_2$），$v$ 和 $w$ 都是联系在一起的。目前，这个定义还显得比较抽象。我们只能看出，它是 $[\![\forall \alpha.A]\!]_\eta$ 的 binary logical relation 版本。但很快，我们就会看到 $[\![\forall \alpha.A]\!]_R$ 的妙用。在此之前，我们首先需要证明 fundamental theorem of logical relation。

## 4.4 fundamental theorem of logical relation

和之前一样，我们首先把 logical relation 拓展到 open term：

$$R^\rho_\Gamma = \{(\sigma, \delta) \mid \forall (x : A) \in \Gamma, (x[\sigma], x[\delta]) \in \mathcal{V}^\rho_A\}$$

然后，我们证明如下的 fundamental theorem of logical relation：

**Theorem 4.1.**

$$\begin{aligned}
&\forall \eta_1, \eta_2, \rho \in R(\eta_1, \eta_2), (\Gamma \vdash t : A), (\sigma, \delta) \in R^\rho_\Gamma, \\
&\quad (t[\sigma], t[\delta]) \in R^\rho_A
\end{aligned}$$

*Proof.* 直接对 $\Gamma \vdash t : A$ 归纳即可。这里展示和 polymorphism 相关的两种情况：

- $A = \forall\alpha.A'$，$\Gamma \vdash t : A'$ 且 $\alpha \notin \texttt{FTV}(\Gamma)$。给定任意的 $V_1, V_2 \subseteq \texttt{Value}_0$ 和 $R_V \subseteq V_1 \times V_2$，由于 $\alpha \notin \texttt{FTV}(\Gamma)$，容易证明 $(\sigma, \delta) \in R_\Gamma^{\rho,\alpha\mapsto R_V}$ 依然成立。所以，根据归纳假设，有 $(t[\sigma], t[\delta]) \in R_{A'}^{\rho,\alpha\mapsto R_V}$。由于 $R_V$ 是任意的，根据定义，$(t[\sigma], t[\delta]) \in R_{\forall\alpha.A'}^{\rho}$
- $A = A'[\alpha \mapsto B]$，$\Gamma \vdash t : \forall\alpha.A'$。根据归纳假设，有 $(t[\sigma], t[\delta]) \in R_{\forall\alpha.A'}^{\rho}$。根据 $R_{\forall\alpha.A'}^{\rho}/\mathcal{V}_{\forall\alpha.A'}^{\rho}$ 的定义，令 $V_1 = [\![B]\!]_{\eta_1}$、$V_2 = [\![B]\!]_{\eta_2}$、$R_V = [\![B]\!]_R$，可得 $(t[\sigma], t[\delta]) \in R_{A'}^{\rho,\alpha\mapsto[\![B]\!]_R}$。

  通过单独归纳，容易证明 $R_A^\rho$ 和类型变量的 substitution 之间有如下互动：

  $$R_{A[\alpha\mapsto B]}^{\rho} = R_A^{\rho,\alpha\mapsto[\![B]\!]_R}$$

  所以 $(t[\sigma], t[\delta]) \in R_{A'[\alpha\mapsto B]}^{\rho}$。这正是我们想要证明的命题

□

## 4.5 theorems for free!

我们又把标准的构造 logical relation、证明 fundamental theorem of logical relation 的流程跑了一遍。唯一的新意，就是 logical relation 以“对于好的参数，给出好的结果”的方式被拓展到了 polymorphism 上。证明 fundamental theorem of logical relation 后，该考虑最重要的问题了：如何利用 fundamental theorem of logical relation/abstraction theorem 证明有用的性质?

让我们从一个简单的例子开始, 考虑一个值 $\varnothing \vdash v : \forall\alpha.\alpha \to \alpha$。从类型签名看, 它很可能是单位函数 $\texttt{id} = \lambda x.x$。那么, 是否有别的可能性呢? 似乎很难想像。但是, 要如何严谨地证明这一点呢? 这就需要 parametricity 的登场。

给定一个 $\varnothing \vdash v_{\texttt{id}} : \forall\alpha.\alpha \to \alpha$，根据 Theorem 4.1，有 $(v_{\texttt{id}}, v_{\texttt{id}}) \in \mathcal{V}_{\forall\alpha.\alpha}^{\rho}$（由于 $\forall\alpha.\alpha$ 中没有自由类型变量，$R$ 无关紧要）。接下来，我们展开 $\mathcal{V}_{\forall\alpha.\alpha}^{\rho}$ 的定义。对于任意的 $V_1, V_2 \subseteq \texttt{Value}_0$，$R_V \subseteq V_1 \times V_2$，有：

$$\begin{aligned} & \forall(v_1, v_2) \in \mathcal{V}_\alpha^{\rho,\alpha\mapsto R_V}, (v_{\texttt{id}}\ v_1, v_{\texttt{id}}\ v_2) \in R_\alpha^{\rho,\alpha\mapsto R_V} \\ \Leftrightarrow\quad & \forall(v_1, v_2) \in R_V, \exists(w_1, w_2) \in R_V, v_{\texttt{id}}\ v_1 \rightsquigarrow^* w_1 \text{ and } v_{\texttt{id}}\ v_2 \rightsquigarrow^* w_2 \end{aligned}$$

注意这里的 $V_1$、$V_2$ 和 $R_V$ 都是任意的！所以，给定一个类型 $B$ 和一个 $\varnothing \vdash v_0 : B$，我们可以构造如下的 $R_V$：

$$\begin{aligned} R_V &\subseteq [\![B]\!]_\eta \times [\![B]\!]_\eta \\ R_V &= \{(v_0, v_0)\} \\ (v_1, v_2) \in R_V &\Leftrightarrow v_1 = v_0 \text{ and } v_2 = v_0 \end{aligned}$$

代入这个 $R_V$，parametricity 告诉我们：

$$\begin{aligned} & \forall (v_1, v_2) \in R_V, \exists (w_1, w_2) \in R_V, v_{\texttt{id}}\ v_1 \rightsquigarrow^* w_1 \text{ and } v_{\texttt{id}}\ v_2 \rightsquigarrow^* w_2 \\ \Leftrightarrow \quad & v_{\texttt{id}}\ v_0 \rightsquigarrow^* v_0 \end{aligned}$$

也就是说，**任意的** $\varnothing \vdash v_{\texttt{id}} : \forall \alpha.\alpha \to \alpha$，都一定满足 $v_{\texttt{id}} = \texttt{id}$。所以，类型为 $\forall \alpha.\alpha \to \alpha$ 的函数**只**有 `id` 一个！

这里，我们已经可以看到 parametricity 的强大之处了。对于类型为 $\forall \alpha.A$ 的表达式，我们可以代入一个**任意的** binary logical relation。通过构造合适的 logical relation，就可以**只根据类型**得到很多“免费”的性质。例如，还可以证明如下的“免费”定理：

- 对于任意的 $\varnothing \vdash f : \forall \alpha.\alpha \to \alpha \to \alpha$，$\varnothing \vdash v_1, v_2 : A$，有 $f\ v_1\ v_2 \rightsquigarrow^* v_1$ 或 $f\ v_1\ v_2 \rightsquigarrow^* v_2$
- 对于任意的 $\varnothing \vdash f : \forall \alpha.\alpha \to B$，如果 $\alpha \notin \texttt{FTV}(B)$，那么对于任意的 $\varnothing \vdash v_1, v_2 : A$，有 $f\ v_1 = f\ v_2$
- （需要往语言中加入列表类型 $[A]$）对于任意的 $\varnothing \vdash f : \forall \alpha.[\alpha] \to [\alpha]$，$\varnothing \vdash v : [A]$，$f\ v$ 一定是 $v$ 中元素的重新排列组合。这一点可以用如下的定理刻画：

  $$g^* \circ f = f \circ g^*$$

  其中 $g^*$ 把函数 $g : A \to B$ 应用到一个列表的每个元素上，是一个 $[A] \to [B]$ 的函数
- 对于任意的 $\varnothing \vdash m : \forall \alpha\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$，一定有 $m = p \circ g^*$。其中 $g : A \to B$，$p : \forall \alpha.[\alpha] \to [\alpha]$（根据上一条，这说明 $p$ 会且只会把参数中的元素重新排列组合）

## 4.6 parametricity 与 lambda encoding

利用 parametricity，我们不仅能得到很多免费的定理，还可以证明一些用 $\lambda$ 表达式编码常见数据类型的方法的正确性。例如，考虑布尔类型 `Bool`，它的 church encoding 如下：

$$\begin{aligned}
\texttt{Bool} &= \forall\alpha.\alpha \to \alpha \to \alpha \\
\texttt{true} &= \lambda x.\lambda y.x \\
\texttt{false} &= \lambda x.\lambda y.y \\
(\texttt{if } t \texttt{ then } u_1 \texttt{ else } u_2) &= t\ u_1\ u_2
\end{aligned}$$

利用这个 encoding，我们可以把使用 `Bool` 的程序翻译到没有 `Bool` 的 polymorphic lambda calculus。然而，如何证明 $\texttt{Bool} = \forall\alpha.\alpha \to \alpha \to \alpha$ 中只有 `true` 和 `false`，没有其他“不好”的元素呢？这就需要 parametricity 的登场。利用 parametricity，可以证明对于任意的 $\varnothing \vdash b : \forall\alpha.\alpha \to \alpha \to \alpha$，$b\ v_1\ v_2$ 要么是 $v_1$，要么是 $v_2$。所以，$\forall\alpha.\alpha \to \alpha \to \alpha$ 中确实只有 `true` 和 `false`。类似地，其他 lambda encoding 的正确性也可以用 parametricity 加以证明：

$$\begin{aligned}
\texttt{Nat} &= \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha \\
\texttt{List}\ A &= \forall\alpha.\alpha \to (A \to \alpha \to \alpha) \to \alpha \\
A \times B &= \forall\alpha.(A \to B \to \alpha) \to \alpha
\end{aligned}$$

## 4.7 一些会破坏 parametricity 的语言构造

本节中演示的语言享有 parametricity 这一性质。但是，不是所有语言都满足 parametricity。有一些很常见的语言构造，就会破坏 parametricity。

第一个例子是 polymorphic equality，即一个多态的比较函数 $(=) : \forall\alpha.\alpha \to \alpha \to \texttt{Bool}$。对 $\forall\alpha.\alpha \to \alpha \to \alpha$ 这个类型运用 parametricity，可以知道它必须是个常数函数：要么是 $\lambda x.\lambda y.\texttt{true}$、要么是 $\lambda x.\lambda y.\texttt{false}$。然而，polymorphic equality 显然不是一个常数函数。所以，它会破坏 parametricity。

第二个例子是 type case，即在多态函数内部对类型进行分类讨论的能力。例如，考虑如下函数：

$$\begin{aligned}
f = \lambda(x : \alpha).\texttt{case } \alpha \texttt{ of} \\
\texttt{Bool} &\Rightarrow \texttt{true} \\
_ &\Rightarrow \texttt{false}
\end{aligned}$$

$\varnothing \vdash f : \forall\alpha.\alpha \to \texttt{Bool}$。根据 parametricity 它应当是一个常数函数，但实际上它不是。所以，type case 也会破坏 parametricity。

一般来说，允许对不同类型有不同行为的语言构造往往会破坏 parametricity。如果一个语言构造对不同的类型有着“一致”的行为，那么它就不会破坏 parametricity。

# 5 Kripke logical relation 与 step-indexing

在第 2 节中，我们使用 logical relation 来描述了一个带副作用的语言的 observation equivalence。然而，这门语言只有一个自然数计数器。现实中，编程语言往往支持动态创建新的 reference。为了描述动态创建 reference 的语义，需要将普通的 logical relation 拓展为 *Kripke* logical relation。

往 $\lambda$-calculus 中加入 reference，尤其是装有函数的 reference，会导致一些意外的后果。它可以在本来没有递归的语言中直接实现递归。例如，阶乘函数 `fact` 可以用 reference 直接实现：

```
let fact =
    let fact_itself = ref (fun x => 0) in
    fact_itself := (fun x =>
        if x = 0
        then 1
        else x * !fact_itself (x - 1));
    !fact_itself
```

这种用 reference 实现递归的方式被称为 *Landing's Knot*。Landing's Knot 的存在意味着，加入了 reference 的语言不再停机了。此外，在描述它的语义时，也会因此遇到一些困难。而解决这一困难的一种方式是 *step indexing*。它在 logical relation 的定义中加入一个自然数 index，表示剩下的执行步数。

本节将以一门支持 reference 的简单语言为例，介绍 Kripke logical relation 和 step indexing 这两种重要的构造。由于篇幅所限和 reference 自身的复杂性，本节构造的 logical relation 只是对类型系统的语义的一个描述，从中不能导出一些有用的性质（因为本节的语言不一定停机）。

## 5.1 一门支持动态创建 reference 的语言

下面给出本节中的语言的语法。它支持动态创建可变的 reference、对它们进行读取和赋值。而且 reference 中可以存储任意类型，包括函数和其他

reference:

$$
\begin{aligned}
\mathbb{N} \ni\ & m, n \\
\texttt{Location} \ni\ & l \\
\texttt{Term} \ni\ & t, u ::= x \mid l \mid n \mid \texttt{ref}\ t \mid\ !t \mid t := u \mid \lambda x.t \mid t\ u \\
\texttt{Type} \ni\ & A, B ::= \texttt{nat} \mid \texttt{ref}\ A \mid A \to B \\
\texttt{Context} \ni\ & \Gamma, \Delta ::= \varnothing \mid \Gamma, x : A
\end{aligned}
$$

注意到在语法中加入了“内存地址”$l \in \texttt{Location}$，这些地址是在描述语义时内部使用的，不能出现在用户写的程序里。语言的操作语义如下:

$$
\begin{aligned}
\texttt{Value} \ni\ & v, w ::= l \mid n \mid \lambda x.t \\
\texttt{EvalContext} \ni\ & E ::= \square \mid \texttt{ref}\ E \mid\ !E \mid E := t \mid v := E \mid E\ t \mid v\ E \\
& S \in \texttt{Store} = \texttt{Location} \rightharpoonup \texttt{Value}
\end{aligned}
$$

$$
\frac{}{(S, E[(\lambda x.t)\ v]) \rightsquigarrow (S, E[t[x \mapsto v]])} \qquad \frac{l \notin \texttt{dom}(S)}{(S, E[\texttt{ref}\ v]) \rightsquigarrow (S[l \mapsto v], E[l])}
$$

$$
\frac{S(l) = v}{(S, E[!l]) \rightsquigarrow (S, E[v])} \qquad \frac{l \in \texttt{dom}(S)}{(S, E[l := v]) \rightsquigarrow (S[l \mapsto v], E[0])}
$$

$S \in \texttt{Store}$ 是程序执行时的存储空间, 用从 Location 到 Value 的 partial function 表示。$\texttt{dom}(S)$ 表示 $S$ 中已分配的内存地址。$S[l \mapsto v]$ 表示将 $S$ 中 $l$ 对应的值修改为 $v$ 得到的新存储空间在 ref 的求值规则中，我们假设有一种固定的方式来分配一个新的、不在 $S$ 中的内存地址。(无论 $l$ 本来在 $S$ 中是否有定义)。赋值作为一个表达式的结果是一个无意义的 0，相当于 ML 中的 unit。

一些常见的语言构造可以用这里的语言进行模拟。例如，$\texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ u$ 依然可以表示为 $(\lambda x.u)\ t$。而顺序执行 $t; u$ 可以表示为 $\texttt{let}\ x_0\ \texttt{=}\ t\ \texttt{in}\ u$ ($x_0$ 不在 $u$ 中出现)。此外，递归定义也可以用 Landing's Knot 在这门语言中实现。$\texttt{let rec}\ f\ \texttt{=}\ t\ \texttt{in}\ u$ 可以表示为:

$$
\begin{aligned}
&\texttt{let}\ f\ \texttt{=} \\
&\quad \texttt{let self = ref}\ t_{\texttt{dummy}}\ \texttt{in} \\
&\quad \texttt{self :=}\ t[f \mapsto\ !\texttt{self}]; \\
&\quad \texttt{self} \\
&\text{in}\ u
\end{aligned}
$$

其中，$t_{\text{dummy}}$ 是一个类型和 $f$ 一致的无意义的值。

本节的语言的类型规则是 1 中的 STLC 的类型规则的拓展。这里只列出新的规则。注意地址 $l$ 没有对应的类型规则，因为它们不会出现在用户的程序中。

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \texttt{ref}\ t : \texttt{ref}\ A} \qquad \frac{\Gamma \vdash t : \texttt{ref}\ A}{\Gamma \vdash !t : A} \qquad \frac{\Gamma \vdash t : \texttt{ref}\ A \qquad \Gamma \vdash u : A}{\Gamma \vdash (t := u) : \texttt{nat}}$$

本节的目标是定义一个 logical relation $R_A$ 来表示“类型为 $A$ 的好的值/表达式”。$R_A$ 的意义主要在于对类型系统的语义进行额外的描述，不一定能导出很重要的性质。

## 5.2 存储空间的类型与 Kripke logical relation

为本节的语言定义 logical relation 并不是一件轻松的事。语言的特性对 logical relation 提出了如下的要求：

1. 一个内存地址 $l$ 是否是“好”的取决于我们在哪个存储空间 $S$ 中考察它。所以 logical relation 必须把 $S$ 也纳入考虑
2. 类型系统要求赋值不改变一个地址的类型。所以，一个“好”的表达式的 reduction 不应该改变存储空间的类型

为了达成要求 2，我们需要定义一个存储空间的类型。我们需要描述它分配了哪些内存地址、每个内存地址上存的值是什么类型。存储空间的类型 $\Phi$ 定义为从地址到类型的 partial function：

$$\Phi \in \texttt{StoreType} = \texttt{Location} \rightharpoonup \texttt{Type}$$

为了达成要求 1，logical relation 应当定义成 $\mathcal{V}_A^\Phi$ 的样子：一个值是否是“好”的，不仅要看类型，还要看所处的存储空间。现在，我们尝试定义 logical relation $\mathcal{V}_A^\Phi$（值）和 $\mathcal{R}_A^\Phi$（表达式）。首先，从 $\mathcal{V}_A^\Phi$ 开始：

$$\begin{aligned}
\mathcal{V}_A^\Phi &\subseteq \texttt{Value} \\
\mathcal{V}_{\texttt{nat}}^\Phi &= \mathbb{N} \\
\mathcal{V}_{\texttt{ref}\ A}^\Phi &= \{l \mid \Phi(l) = A\} \\
\mathcal{V}_{A \to B}^\Phi &= \{v \mid \forall \Phi' \supseteq \Phi, w \in \mathcal{V}_A^{\Phi'}, v\ w \in R_B^{\Phi'}\}
\end{aligned}$$

其中，$\Phi \subseteq \Phi'$（$\Phi' \supseteq \Phi$）意味着对于任意的 $l \in \texttt{dom}(\Phi)$，$\Phi(l) = \Phi'(l)$。在 $\mathcal{V}_{A \to B}$ 的定义中，参数 $w$ 是在一个不同的存储空间 $\Phi'$ 中给出的。这是因为，一个函数被创建时的存储空间和它被调用时的存储空间不一定相同，例如：

```
let f = ... in
let r = ref 0 in (* 存储空间改变了！ *)
f r (* 参数 r 在改变后的存储空间里才有意义！ *)
```

但是，我们要求 $\Phi \subseteq \Phi'$。这意味着 $f$ 定义时可能用到的地址在被调用时一定都还在，而且类型没有改变。

接下来，我们还需要定义表达式的语义 $R_A^\Phi$。由于语言不一定停机，我们只要求在停机时表达式能给出好的结果。除此之外，我们还需要定义“一个存储空间 $S$ 有类型 $\Phi$”意味着什么，把 logical relation 拓展到 $\Phi$ 上：

$$
\begin{aligned}
R_A^\Phi &\subseteq \texttt{Term} \\
R_A^\Phi &= \{t \mid \forall S \in R_\Phi, (S,t) \Downarrow_A^\Phi \text{ or } (S,t) \Uparrow\} \\
(S,t) \Downarrow_A^\Phi &\text{ iff } \exists \Phi' \supseteq \Phi, S' \in R_{\Phi'}, v \in \mathcal{V}_A^{\Phi'}, (S,t) \rightsquigarrow^* (S',v) \\
(S,t) \Uparrow &\text{ iff } \nexists S', v, (S,t) \rightsquigarrow^* (S',v)
\end{aligned}
$$

$$
\begin{aligned}
R_\Phi &\subseteq \texttt{Store} \\
R_\Phi &= \{S \mid \forall l \in \texttt{dom}(\Phi), S(l) \in \mathcal{V}_A^\Phi\}
\end{aligned}
$$

这里 $(S,t) \Downarrow_A^\Phi$ 表示 $(S,t)$ 在 $\Phi$ 下会停机，并能给出类型为 $A$ 的好的结果。$(S,t) \Uparrow$ 表示 $(S,t)$ 不停机。由于程序在运行过程中可能分配新的 reference，表达式执行的结果可能活在一个不同的存储空间类型 $\Phi'$ 里。但我们要求 $\Phi \subseteq \Phi'$，这意味着程序的执行不能销毁现有的内存地址，也不能改变一个内存地址的类型。

这里的 logical relation 比起前面几节的，多了一个参数 $\Phi$。这其实是一种更一般的 logical relation，叫作 Kripke logical relation。Kripke logical relation 中，有一个“可能的世界”的偏序集 $W$，用于表示语法中不存在的、只在语义中存在的一些外部环境。$W$ 上的偏序关系 $w \leqslant w'$ 称为“可达性”，它表示从世界 $w$ 有可能移动到世界 $w'$。在本节中，$W = \texttt{StoreType}$，偏序关系是 $\Phi \subseteq \Phi'$。$\texttt{StoreType}$ 刻画了存储空间 $S$ 这一语法中不存在的语义环境，而由于“不能销毁地址或改变地址的类型”的约束，如果程序能从 $S : \Phi$ 执行到 $S' : \Phi'$，就一定有 $\Phi \subseteq \Phi'$。这符合可达性关系的直觉。

有了可能性的世界 $W$ 后，一个 Kripke logical relation 就是一个额外以世界 $w$ 为参数的 logical relation $R_w$。Kripke logical relation 需要尊重可达

性：如果一个值在世界 $w$ 是好的，那么在所有能从 $w$ 到达的世界中 $w' \geqslant w$ 中，它也依然应当是好的。

Kripke logical relation 是一个非常通用的构造。几乎所有（如果不是全部）logical relation 都可以看作 Kripke logical relation。上面的 $\mathcal{V}^{\Phi}_{A\to B}$ 的构造，同样是一个更一般的构造——Kripke implication 的特例。本文的后面将会介绍 Kripke logical relation 的其他应用和它的数学含义。本节的后面将专注于继续定义带 reference 的语言的 logical relation 语义。

## 5.3 这是一个循环定义！

在前面的 $R_\Phi$ 的定义中，“$S(l) \in \mathcal{V}^{\Phi}_{A}$”这一条件里提到了 $\Phi$ 自己。这是合理的，因为语言中有像 `ref` (`ref nat`) 这样的嵌套类型的存在，描述这些类型的语义时，可能需要在 $\Phi$ 中跳转任意多次。然而，这种自我指涉也是极其危险的。事实上，上一小节定义的 logical relation 是不存在的、循环定义的。因为：

- $\mathcal{V}^{\Phi}_{A\to B}$ 的定义中用到了 $R^{\Phi}_{B}$
- $R^{\Phi}_{B}$ 的定义中用到了 $R_\Phi$
- 如果存在某个 $l$ 使得 $\Phi(l) = A \to B$，那么 $R_\Phi$ 的定义就需要用到 $\mathcal{V}^{\Phi}_{A\to B}$，形成循环定义！

这一循环定义也可以从另一个角度来观测。在前一小节的定义里，$\Phi$ 给每个地址 $l$ 赋予的是一个**语法的**类型 $A \in \mathtt{Type}$。如果我们参照 1.4 的做法，也可以在 $\Phi$ 中给每个地址 $l$ 赋予一个**语义的**类型。而语义的类型正是像 $R^{\Phi}_{A}$ 这样的集合/predicate。令 `SemType` 表示语义的类型，那么：

$$\begin{aligned}\mathtt{StoreType} &= \mathtt{Location} \rightharpoonup \mathtt{SemType}\\ \mathtt{SemType} &= \mathtt{StoreType} \to \mathcal{P}(\mathtt{Value})\end{aligned}$$

因为 `SemType` 是对每个 $\Phi$ 分别定义的，所以每个类型 $A$ 关联的是一个 $\mathtt{StoreType} \to \mathcal{P}(\mathtt{Value})$ 的函数。但是，这意味着 `StoreType` 和 `SemType` 是互相递归的。而且，`StoreType` 在 `SemType` 的定义中出现在函数参数的位置：这种递归是无法通过归纳来定义的、无解的。

这里定义 logical relation 时出现的循环，经常是与不停机相关联的。例如，无类型的 $\lambda$-calculus 中，如果尝试将 $\lambda$ 解释为函数，那么表达式的语

义 $D$ 就必须满足 $D = D \to D$。这里的 $D$ 在递归中，同样在函数参数的位置提到了自己。而众所周知，无类型的 $\lambda$-calculus 是不停机的。

事实上，Landing's Knot 的本质同样是这里的循环定义。正是因为 `StoreType` 和 `SemType` 之间的这种循环，才导致语言中可以有不停机的程序、可以写出任意的递归。而且，目前这个错误的 logical relation 定义中，循环的部分是函数类型的语义。而 Landing's Knot 恰恰就是通过装有函数的 reference 实现的。

事实上,如果我们对 reference 可以存储的东西做出限制,只允许 reference 中存储一些简单的类型，不允许函数和其他 reference 被存储在 reference 中，那么简单类型的 logical relation 中就不需要提到 $\Phi$，可以直接定义成 $\mathcal{V}_A$，从而整个 logical relation 的定义就可以按照

> 简单类型 → 存储空间的类型 → 其他类型（reference 和函数）

的顺序定义，不再会循环。由于 reference 不再能存储函数，Landing's Knot 被解开了。允许存储函数的 reference 由于上述的复杂性, 常常被称为 higher order reference 或 general reference。但在实际语言中, higher order reference 是很实用的。所以接下来，本节将介绍如何用名为 step indexing 的技巧定义出 general reference 的 logical relation 语义。

## 5.4 用 step indexing 解开 Landing's Knot

本节中的支持 general reference 的语言有 Landing's Knot、可能不停机。它的 logical relation 语义也因此无法简单地定义出。所以，我们必须解开 Landing's Knot。为此，我们可以从“不停机”的角度入手。

假设有一段可能不停机的程序，我们应该如何判断它是否停机？停机问题是不可判定的。我们无法直接得到答案。所以，现实中常见的做法是：让程序跑一段时间，直到我们失去耐心。如果在这段时间内程序停机并给出结果了，那么它的确是停机的，皆大欢喜。但如果它在这段时间内没有给出结果, 那么它有可能真的不停机, 也有可能只是需要更多时间来得出结果。

如果把上面的这个过程形式化，那么首先我们给程序的执行设定一个“步数”的概念，用于代替时间。接下来，我们给程序一个步数限制 $k$，观察它能否在 $k$ 步之内得到一个结果。如果能，那么它停机。但如果不能，它

也依然有可能只是需要更多步数来停机。

只提供一个固定的步数限制的话，用于判断程序是否停机是不够准确的。但是，如果我们能知道程序在**任意的**步数限制下的行为的话，就能得到程序的完整行为：如果一个程序在任意的步数限制 $k$ 下都无法得出结果，那么它不停机。否则，它停机，并能在某个步数限制 $k$ 下给出结果。

步数限制不仅能用于刻画程序的停机行为，它还能用于解开 Landing's Knot，规避循环定义！方法很简单：我们暴力地在 logical relation 的定义的每个地方塞入一个额外的步数限制参数（这就是“step indexing”这一名字的由来）。接下来，只要可能导致不停机的操作都会消耗步数，我们就能通过对步数进行归纳来定义出 logical relation！

按照这一思路，我们可以重新尝试定义 logical relation。为此，首先需要定义一个“步数”的概念。这里，我们用“读取 reference 的次数”作为步数的概念。用 $(S,t) \rightsquigarrow^k (S',t')$ 表示 $(S,t)$ 能在使用 $k$ 次读取 reference 的规则（$(S,E[!l]) \rightsquigarrow (S,E[S(l)])$）的情况下求值到 $(S',t')$。其他步数定义也是可能的，只要保证会引入不停机和循环的操作至少需要一步即可。例如，直接用 reduction 次数作为步数也是可行的。

有了步数的概念，就可以开始重新定义 logical relation 了。现在，每个类型的 logical relation 为形如 $\mathcal{V}_A^{\Phi,k}/R_A^{\Phi,k}$。我们人为地在每个 logical relation 中加入了一个“步数限制”的参数 $k$。我们依然定义三个 logical relation：

$$\mathcal{V}_A^{\Phi,k} \subseteq \texttt{Value} \qquad R_A^{\Phi,k} \subseteq \texttt{Term} \qquad R_\Phi^k \subseteq \texttt{Store}$$

分别用来描述“好”的值、表达式和存储空间。它们的直觉意义如下：为了阐明步数参数 $k$ 的意义，以表达式的 logical relation 为例。$t \in R_A^{\Phi,k}$ 意味着，如果我们在 $\Phi$ 下把 $t$ 当成一个类型为 $A$ 的好的表达式，那么 $k$ **步执行之内**不会产生不好的结果。现在，我们开始填上各个 logical relation 的具体定义。首先，值的 logical relation $\mathcal{V}_A^{\Phi,k}$ 的定义如下：

$$\begin{aligned}
\mathcal{V}_{\texttt{nat}}^{\Phi,k} &= \mathbb{N} \\
\mathcal{V}_{\texttt{ref } A}^{\Phi,k} &= \{l \mid \Phi(l) = A\} \\
\mathcal{V}_{A\to B}^{\Phi,k} &= \{v \mid \forall \Phi' \supseteq \Phi, k' \leqslant k, w \in \mathcal{V}_A^{\Phi',k'}, v\ w \in R_B^{\Phi',k'}\}
\end{aligned}$$

$\mathcal{V}_{\texttt{ref } A}^{\Phi,k}$ 的定义和之前一样。Reference 和步数的互动被放在了 $R_\Phi^k$ 中。$\mathcal{V}_{A\to B}^{\Phi,k}$ 的定义发生了改变，它需要和步数进行互动。定义中的各个条件的意义如下：

| | |
|---|---|
| $\Phi' \subseteq \Phi$ | 函数被创建后、被调用前，程序可能分配了新的 reference |
| $k \leqslant k'$ | 函数被创建后、被调用前，程序可能消耗了一些步数 |
| $w \in \mathcal{V}_A^{\Phi',k'}$ | 函数调用时只剩 $k'$ 步了，所以参数 $w$ 只需要在 $k'$ 步内有效 |
| $v\ w \in R_A^{\Phi,k'}$ | 函数调用时只剩 $k'$ 步了，所以函数调用自身也只需要在 $k'$ 步内有效 |

接下来是表达式的 logical relation $R_A^{\Phi,k}$:

$$
\begin{aligned}
R_A^{\Phi,k} &= \{t \mid \forall S \in R_\Phi^k, (S,t) \Downarrow_A^{\Phi,k} \text{ or } (S,t) \Uparrow^k\} \\
(S,t) \Downarrow_A^{\Phi,k} \text{ iff } &\exists k' \leqslant k, \Phi' \supseteq \Phi, S' \in R_{\Phi'}^{k-k'}, v \in \mathcal{V}_A^{\Phi',k-k'}, \\
&(S,t) \rightsquigarrow^{k'} (S',v) \\
(S,t) \Uparrow^k \text{ iff } &\exists S', t', \\
&(S,t) \rightsquigarrow^{k} (S',t') \text{ and } t' \notin \texttt{Value}
\end{aligned}
$$

定义里用到了两个辅助定义。$(S,t) \Downarrow_A^{\Phi,k}$ 表示 $(S,t)$ 能在 $k$ 步内停机，并得到一个 $\Phi$ 下的、类型为 $A$ 的值。$(S,t) \Uparrow^k$ 则表示 $(S,t)$ 在 $k$ 步之内无法停机。现在，$t \in R_A^{\Phi,k}$ 告诉我们，在**任意**类型为 $\Phi$ 的存储空间 $S$ 中，$t$ 要么在 $k$ 步内无法停机，一旦停机就一定能给出一个好的结果。最后，存储空间自己的语义 $R_\Phi^k$ 如下:

$$
\begin{aligned}
R_\Phi^0 &= \texttt{Store} \\
R_\Phi^{k+1} &= \{S \mid \forall l \in \texttt{dom}(\Phi), S(l) \in \mathcal{V}_A^{\Phi,k}\}
\end{aligned}
$$

大体上，它表达的依然是“每个内存地址存储的值都是好的”这个约束。只是，它还需要处理步数。在还剩 $k+1$ 步时，由于读取 reference 消耗一步，从存储空间中读取出值后就只剩下 $k$ 步了，因此读出的值也只需要在

$k$ 步内是好的。如果只剩下 0 步，那么无法从存储空间中读取值，因此任何存储空间都在 0 步内是好的。

最后，让我们展示这个新的 logical relation 的一些重要性质。加入步数参数后，新的 logical relation 依然是一个 Kripke logical relation。只不过它的可能世界 $W$ 从 $\texttt{StoreType}$ 变成了 $\texttt{StoreType} \times \mathbb{N}$，额外引入了一个步数参数。可达性关系则从 $\Phi \subseteq \Phi'$ 变成了下面的关系：

$$(k, \Phi) \leqslant (k', \Phi') \text{ iff } k' \leqslant k \text{ and } \Phi \subseteq \Phi'$$

这个定义是非常符合直觉的：随着程序的执行，我们到达一个分配了更多内存地址的世界，但已有的内存地址不会被销毁、类型也不会改变。随着程序的执行，剩余的步数只会越来越少。所以我们可以从步数多的世界到达步数少的世界。我们的 logical relation 尊重这个可达性关系。这可以表述成如下的引理：

**Lemma 5.1.** 如果 $(k, \Phi) \leqslant (k', \Phi')$，那么 $\mathcal{V}_A^{\Phi,k} \subseteq \mathcal{V}_A^{\Phi',k'}$，$R_A^{\Phi,k} \subseteq R_A^{\Phi',k'}$。如果 $k' \leqslant k$，那么 $R_\Phi^k \subseteq R_\Phi^{k'}$

*Proof.* 对 logical relation 的定义进行归纳即可。在证明 $A = A' \to B$ 的情况是，会发现如果没有定义里“任意 $\Phi' \supseteq \Phi$、$k' \leqslant k$”的部分，就无法证明这条引理 □

## 5.5 fundamental theorem of logical relation

经过一系列改动，我们终于定义出了一个支持 general reference 的 logical relation。最后一步，就是证明熟悉的 fundamental theorem of logical relation。在此之前必须要做的一项工作，是把 logical relation 拓展到 open term：

$$R_\Gamma^{\Phi,k} = \{\sigma \mid \forall (x : A) \in \Gamma, x[\sigma] \in \mathcal{V}_A^{\Phi,k}\}$$

现在，我们可以证明 fundamental theorem of logical relation 了：

**Theorem 5.2.** 如果 $\Gamma \vdash t : A$，那么对于任意的 $\Phi, k, \sigma \in R_\Gamma^{\Phi,k}$，有 $t[\sigma] \in R_A^{\Phi,k}$

*Proof.* 这个证明比较长。不感兴趣的读者可以跳过，特别感兴趣的读者可以自行证明作为练习。对 $t$ 进行归纳：

- $t = x$。根据 $R_\Gamma^{\Phi,k}$ 的定义显然

- $t = \lambda x.t', A = A' \to B_\circ$ 此时有 $\Gamma, x : A' \vdash t' : B_\circ$ 由于 $t$ 已经是一个值，我们需要证明 $t \in \mathcal{V}^{\Phi,k}_{A' \to B\circ}$。根据定义，假设有 $\Phi' \supseteq \Phi, k' \leqslant k, v_a \in \mathcal{V}^{\Phi,k'}_{A'}$，我们需要证明 $t[\sigma]\ v_a \in R^{\Phi',k'}_B$。对于任意的 $S$，有如下的求值序列：

  $$(S, t[\sigma]\ v_a) = (S, (\lambda x.t')[\sigma]\ v_a) \rightsquigarrow (S, t'[\sigma, x \mapsto v_a])$$

  而且这一过程中没有读取 reference，所以消耗的步数为 0。根据 Lemma 5.1 有 $\sigma \in R^{\Phi',k'}_\Gamma$，所以 $(\sigma, x \mapsto v_a) \in R^{\Phi',k'}_{\Gamma,x:A'\circ}$。根据归纳假设有 $t'[\sigma, x \mapsto v_a] \in R_{\Phi',k'}$，所以 $t[\sigma]\ v_a \in R_{\Phi',k'\circ}$。根据定义这意味着 $t[\sigma] \in \mathcal{V}^{\Phi,k}_{A' \to B}$

- $t = t_f\ t_a$，$\Gamma \vdash t_f : A' \to A$，$\Gamma \vdash t_a : A'_\circ$。根据归纳假设 $t_f \in R^{\Phi,k}_{A' \to A\circ}$ 给定一个初始状态 $S_0 \in R^k_\Phi$，如果 $(S_0, t)$ 能停机，那么完整的求值过程和每步完成后的性质如下：

  $$\begin{array}{lll} & (S_0, t_f\ t_a) & S_0 \in R^k_\Phi \\ \rightsquigarrow^{k_1} & (S_1, v_f\ t_a) & \exists \Phi_1 \supseteq \Phi, S_1 \in R^{k-k_1}_{\Phi_1}, v_f \in \mathcal{V}^{\Phi_1,k-k_1}_{A' \to A} \\ \rightsquigarrow^{k_2} & (S_2, v_f\ v_a) & \exists \Phi_2 \supseteq \Phi_1, S_2 \in R^{k-k_1-k_2}_{\Phi_2} v_a \in \mathcal{V}^{\Phi_2,k-k_1-k_2}_{A'} \\ \rightsquigarrow^{k_3} & (S_3, v) & \exists \Phi_3 \supseteq \Phi_2, S_3 \in R^{k-k_1-k_2-k_3}_{\Phi_3} v \in \mathcal{V}^{\Phi_3,k-k_1-k_2-k_3}_{A} \end{array}$$

  这里第一步的性质来自于对 $t_f$ 的归纳假设。第二步的性质来自于对 $t_a$ 的归纳假设。第三步的性质来自于 $v_f \in \mathcal{V}^{\Phi_1,k-k_1}_{A' \to A}$ 的性质。前一步的性质将被用作初始条件开启下一步。最终：

  - 如果 $t$ 在 $k$ 步内不能停机（$t_f$ 不停机、$t_a$ 不停机，$v_f\ v_a$ 不停机或者 $k_1 + k_2 + k_3 > k$），那么 $t_f\ t_a$ 在 $k$ 步内不会停机，根据定义 $t_f\ t_a \in R^{\Phi,k}_A$。
  - 如果最终上面的整个过程能在 $k$ 步内完成，即 $k_1 + k_2 + k_3 \leqslant k$，那么根据定义有 $(S_0, t_f\ t_a) \Downarrow^{\Phi,k}_A$，所以 $t_f\ t_a \in R^{\Phi,k}_A$

- $t = \texttt{ref}\ t'$，$A = \texttt{ref}\ A'_\circ$。此时有 $\Gamma \vdash t' : A'_\circ$。这里不失一般性地，我们只考虑 $t'$ 已经求值完毕的情况。$t'$ 是任意表达式的情况参照上面 $t = t_f\ t_a$ 的情况处理即可。假设 $t = \texttt{ref}\ v$ 且 $v \in \mathcal{V}^{\Phi,k}_{A'}$，那么对于任意的 $S \in R^k_\Phi$，有 $(S, t) \rightsquigarrow^0 (S[l \mapsto v], l)$，其中 $l \notin \texttt{dom}(S)$。令 $\Phi' = \Phi, l \mapsto A'$，显然 $\Phi' \supseteq \Phi_\circ$ 现在我们需要证明两件事：$S[l \mapsto v] \in \Phi'$，以及 $l \in \mathcal{V}^{\Phi',k}_{\texttt{ref}\ A\circ}$。后者根据定义显然成立。最后只需证明前者即可：

  - 如果 $k = 0$，那么 $S[l \mapsto v] \in R^k_{\Phi'}$ 根据定义一定成立

  - 如果 $k > 0$，那么根据 Lemma 5.1 $v \in \mathcal{V}_{A'}^{\Phi,k} \subseteq \mathcal{V}_{A'}^{\Phi',k-1}$，所以 $S[l \mapsto v] \in R_{\Phi'}^{k}$

- $t = !t'$，$\Gamma \vdash t' : \texttt{ref}\ A$。这里依然只考虑 $t' = v$ 已经是一个值的情况。由于 $v \in \mathcal{V}_{\texttt{ref}\ A}^{\Phi,k}$，$v$ 一定是一个内存地址 $l$。假设有 $S \in R_{\Phi}^{k}$，$t$ 的完整求值过程为：

  $$(S, t) = (S, !l) \rightsquigarrow^{1} (S, S(l))$$

  $k = 0$ 时，由于读取 reference 消耗一步，$!l$ 无法在 $k$ 步内停机，$!l \in R_{A}^{\Phi,k}$ 直接成立。当 $k > 0$ 时，由于 $v \in \mathcal{V}_{\texttt{ref}\ A}^{\Phi,k}$，有 $\Phi(l) = A$、$S(l) \in \mathcal{V}_{A}^{\Phi,k-1}$。根据定义，这意味着 $(S, !l) \Downarrow_{A}^{\Phi,k}$（取 $k' = 1$）。所以 $!l \in R_{A}^{\Phi,k}$

- $t = t_r := t_v$，$A = \texttt{nat}$。此时 $\Gamma \vdash t_r : \texttt{ref}\ A'$，$\Gamma \vdash t_v : A'$。依然只考虑 $t_r$、$t_v$ 都是值的情况，那么 $t = l := v$。给定 $S \in R_{\Phi}^{k}$，$t$ 的求值过程如下：

  $$(S, t) = (S, l := v) \rightsquigarrow^{0} (S[l \mapsto v], 0)$$

  $0 \in \mathcal{V}_{\texttt{nat}}^{\Phi,k}$ 显然成立。现在，我们只需要证明存在 $\Phi' \supseteq \Phi$ 使得 $S[l \mapsto v] \in R_{\Phi'}^{k}$ 即可。由于赋值没有分配新的 reference，这里应该直接取 $\Phi' = \Phi$。现在，如果 $k = 0$，那么 $S[l \mapsto v]$ 显然成立。如果 $k > 0$，根据对 $t_r = l$ 的归纳假设，$l \in \mathcal{V}_{\texttt{ref}\ A'}^{\Phi,k}$，所以 $\Phi(l) = A'$。根据对 $t_v = v$ 的归纳假设和 Lemma 5.1，$v \in \mathcal{V}_{A'}^{\Phi,k} \subseteq \mathcal{V}_{A'}^{\Phi,k-1}$。所以根据定义 $S[l \mapsto v] \in R_{\Phi'}^{k}$

□

## 5.6 logical relation 的直觉意义

在完成了所有证明之后，不妨回头来看看我们证明了什么。本节前面提到过，这里的 logical relation 不一定能直接导出很有用的性质。所以，它应当被看作对类型系统的语义的一个补充和描述。那么，这就要求我们必须从直觉上理解 $t \in R_{A}^{\Phi,k}$ 的意义。

步数限制 $k$ 是我们为了绕开循环定义而人为引入的概念。所以，它在 logical relation 中的出现似乎不是很能让人满意。幸运的是，在 fundamental theorem of logical relation 中，我们证明了对于**任意的** $k$ 都有 $t \in R_{A}^{\Phi,k}$。所以，我们可以定义如下几个不涉及步数的 logical relation：

$$\begin{array}{lll} \mathcal{V}_{A}^{\Phi} = & \{v \mid \forall k, v \in \mathcal{V}_{A}^{\Phi,k}\} & \qquad R_{A}^{\Phi} = \{t \mid \forall k, t \in R_{A}^{\Phi,k}\} \\ R_{\Phi} = & \{S \mid \forall k, S \in R_{\Phi}^{k}\} & \qquad R_{\Gamma}^{\Phi} = \{\sigma \mid \forall k, \sigma \in R_{\Gamma}^{\Phi,k}\} \end{array}$$

从带步数的 fundamental theorem，很容易推出一个不带步数的 fundamental theorem：

**Corollary 5.3.** 对于任意的 $\Gamma \vdash t : A$，$\Phi$ 和 $\sigma \in R^{\Phi}_{\Gamma}$，有 $t[\sigma] \in R^{\Phi}_{A}$

接下来，就是最有趣的部分。如果把定义展开，就能发现不涉及步数的 logical relation 满足如下的性质：

**Theorem 5.4.** 下列式子成立：

$$\begin{aligned}
\mathcal{V}^{\Phi}_{\texttt{nat}} &= \texttt{nat} \\
\mathcal{V}^{\Phi}_{\texttt{ref } A} &= \{l \mid \Phi(l) = A\} \\
\mathcal{V}^{\Phi}_{A\to B} &\subseteq \{v \mid \forall \Phi' \supseteq \Phi, w \in \mathcal{V}^{\Phi'}_{A}, v\ w \in \mathcal{R}^{\Phi'}_{B}\}
\end{aligned}$$

$$\begin{aligned}
R^{\Phi}_{A} &= \{t \mid \forall S \in R_{\Phi}, (S,t) \Downarrow^{\Phi}_{A} \ \ or\ (S,t) \Uparrow\} \\
(S,t) \Downarrow^{\Phi}_{A} &\ \textit{iff}\ \exists \Phi' \supseteq \Phi, S' \in R_{\Phi'}, v \in \mathcal{V}^{\Phi'}_{A}, (S,t) \rightsquigarrow^{*} (S',v) \\
(S,t) \Uparrow &\ \textit{iff}\ \nexists S', v, (S,t) \rightsquigarrow^{*} (S',v)
\end{aligned}$$

$$R_{\Phi} = \{S \mid \forall l \in \texttt{dom}(\Phi), S(l) \in \mathcal{V}^{\Phi}_{A}\}$$

*Proof.* 由于空间原因略去。读者可以自行验证，注意运用“任意步数限制 $k$”的灵活性。注意 $\mathcal{V}^{\Phi}_{A\to B}$ 的性质只是一个子集关系，但方向上，$v \in \mathcal{V}^{\Phi}_{A\to B}$ 比式子右侧我们想要的性质更强。所以这不是一个大问题 □

观察这几条性质可以发现，这几乎就是 5.2 中，有循环定义问题的 logical relation 的定义！5.2 中的 logical relation 是更简单的，它的直觉意义也要清晰得多。所以如果没有循环定义问题，它作为类型系统的一个语义描述是足够好的。然而，通过 step indexing，循环定义的问题得到了解决。而只要加上一个任意步数 $k$ 的条件，步数自身又可以消去，使我们回到原来的符合直觉的、简单的 logical relation！所以，加入了 step indexing 之后的 logical relation 用来描述类型系统的语义，依然是足够好的、符合直觉的。

# 6 使用 Kripke logical relation 证明 STLC 的 normalizion

我们已经看到了很多利用 logical relation 证明类型系统性质的例子。但是，这些例子处理的都是 *closed term*。例如，第一节证明的、STLC 的停机性，是 closed term 的停机性。然而，经过一点小小的改造，logical relation 还可以证明 open term 的性质。本节就将利用 logical relation，证明 STLC 的 normalization。

处理 open term 需要对 logical relation 做出改造。然而，出人意料的是，只需要使用 Kripke logical relation，就能在步做出大改动的情况下让 logical relation 支持 open term。在上一节中，Kripke logical relation 被用于描述 reference 的存储空间这一完全无关的对象。这展示了 Kripke logical relation 超高的泛用性。

## 6.1 open term 的语义与 normalization

大部分时候，我们只关心一门语言在 closed term 下的语义。在运行时，未定义的变量被认为是无意义的。但有时候，我们也需要直接操作 open term，例如：

- 在判定两段程序是否等价时，由于 $\lambda$ 的存在，必须要处理 open term
- 有时（例如在编译器中），我们希望能够将一段程序化简为等价但更简单的程序。化简函数时，同样需要处理 open term

在前面的章节中，我们处理一个 open term $t$ 的方式，是要求只要给 $t$ 中的变量赋予任意“好”的值，$t$ 都能得到好的结果。然而，这种做法有时并不可行，例如：

- 在自动判定两段程序是否等价时，2 中的 logical relation 就难以实现。因为判定算法无法遍历“任意好的 substitution $\sigma$”
- 在编译器化简程序时，变量都没有具体的值。遍历所有可能的值同样不可行

所以，我们需要一种原生支持 open term 的语义。从上面的两种应用中，能够产生两种思路：

- 如果从“等价关系”的角度出发，我们可以定义一个 open term 上的等价关系 $t \equiv u$
- 如果从“化简”的角度出发，我们可以定义一个 open term 上的 reduction $t \rightsquigarrow u$

而把这两种定义方式结合在一起的，就是“normalization”这一操作：

- 对于等价关系来说，normalization 需要给每个程序 $t$ 找到一个等价的“最简单版本” $\texttt{nf}\ t$，称为 normal form。如果 $t \equiv u$（$t$ 和 $u$ 等价），那么 normal form 应当满足 $\texttt{nf}\ t = \texttt{nf}\ u$（$t$ 和 $u$ 的 normal form 是完全相同或 $\alpha$-等价的）。但是，“所有程序都存在对应的 normal form”并不是自动成立的。所以，我们需要证明如下的定理：

  **Theorem.** $\forall t, \exists n \in \texttt{Nf}, t = n$

- 对于 reduction 来说，normalization 就是反复应用 reduction，直到得到的表达式是一个不能再 reduce 的“最简单的”表达式（normal form）。但是，“所有程序的 reduction 都一定停机”同样不是自动成立的。所以，我们需要证明：

  **Theorem.** $\forall t, \exists t', t'$ *irreducible and* $t \rightsquigarrow^* t'$

Normalization 同样也是“判定程序等价”和“化简程序”这两种应用之间的桥梁。假设我们有一个 normalization 算法（无论是基于等价关系定义的还是 reduction 定义的）：

- 要判定两段程序 $t$ 和 $u$ 是否等价，只要算出它们的 normal form，并比较 normal form 是否相等即可
- 要化简一段程序，只需要直接求出它的 normal form 即可

传统上，normalization 往往是基于 reduction 的。此时，我们可以直接从 closed term 语义中借来 reduction 规则。只不过，我们需要稍稍修改 reduction 以使它适配 open term。对于 $\lambda$-calculus 来说，需要允许 reduction 在 $\lambda$ 下发生：如果 $t \rightsquigarrow t'$，那么 $\lambda x.t \rightsquigarrow \lambda x.t'$。

然而，基于 reduction 的 normalization 也有许多问题。例如：

- “求值顺序”的定义不再自然。假如我们尝试把 call-by-value 拓展到 open term 下，那么 $(\lambda x.t)\ v$ 依然有多种 reduction 方式：直接 $\beta$-reduction，或是在 $t$ 中进行 reduction。

  当然，我们依然可以强行规定一种 reduction 顺序。但更多时候，open term 的 reduction 会定义成非确定性的：允许任意的 reduction 顺序，表达式可以有不止一种 reduction 方式。但这么一来，就需要把 normalization 升级成 *strong* normalization：任何求值序列都停机。

  证明 strong normalization 比证明停机性更困难，因为必须处理求值顺序的不确定性。证明过程中需要用到很多 strong normalization 自身的性质，而且这些性质本身并不容易证明

- 并不是所有等价关系都有自然的顺序。之前的章节中，我们考虑的语言都只有 $\beta$-reduction。但假如我们想把 $\eta$ 等价：$t = \lambda x.t\ x\ (x \notin \mathtt{FV}(t))$ 也加入语言中，就必须面对方向选择的问题：

  - $\eta$-expansion，即 $t \rightsquigarrow \lambda x.t\ x$，有更好的性质：每个函数类型的 normal form 都一定是一个 $\lambda$。然而，它需要在 reduction 的定义引入类型：把非函数类型的表达式 reduce 到一个 $\lambda$ 会直接导致类型错误。此外，$\eta$-expansion 会破坏 strong normalization。利用 $\eta$-expansion 可以构造出如下的无穷 reduction：

    $$t\ u \rightsquigarrow_\eta (\lambda x.t\ x)\ u \rightsquigarrow_\beta t\ u \rightsquigarrow \cdots$$

  - $\eta$-contraction，即 $\lambda x.t\ x \rightsquigarrow t$，更容易实现。它不需要类型，也不破坏 strong normalization。然而，它得到的 normal form 的形状没有 $\eta$-expansion 漂亮

而基于等价关系的 normalization 就没有这些问题。它从一开始就没有“reduction”的概念，因而也没有求值顺序的问题。对于 $\eta$-等价，基于等价关系的 normalization 可以直接产生 $\eta$-long （即所有函数类型的 normal form 都是一个 $\lambda$，对应 $\eta$-expansion）的 normal form，因为 normalization 不需要按照“反复应用 reduction”的方式实现。

本节将会介绍基于等价关系的 normalization，因为它更简单、性质更好。现实中，这种基于等价关系的语义定义常见于支持 dependent type 的类型系统中。基于 reduction 的 normalization 证明和本节的证明有许多共通之

处。具体来说，open term 的处理方式、Kripke logical relation 中可能世界的选择、fundamental theorem 的表述，以及 saturated set 的构造都是共通的（下面会逐一介绍这些内容）。只不过，在 logical relation 的定义中需要把等价关系改成 reduction，以及证明 fundamental theorem 时需要用到一些 strong normalization/reduction 自身的性质。

## 6.2 STLC 上的 $\beta\eta$ 等价关系及其 normal form

本节中，我们处理的语言回到了第一节中的 STLC（加上一个内建类型 `Ans`）。语言的语法和类型规则都没有变化。但是，本节中不再使用 reduction 来定义语义，转而使用一个等价关系。此外，本节的语义不止考虑了 $\beta$-reduction，还考虑了 $\eta$-等价：

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x.t)\ u \equiv_B t[x \mapsto u]} \qquad \frac{\Gamma \vdash t : A \to B}{\Gamma \vdash t \equiv_{A\to B} \lambda x.t\ x}$$

$$\frac{\Gamma, x : A \vdash t \equiv u : B}{\Gamma \vdash \lambda x.t \equiv_{A\to B} \lambda x.u} \qquad \frac{\Gamma \vdash t_1 \equiv_{A\to B} t_2 \qquad \Gamma \vdash u_1 \equiv_A u_2}{\Gamma \vdash t_1\ u_1 \equiv_B t_2\ u_2}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \equiv_A t} \qquad \frac{\Gamma \vdash t \equiv_A u}{\Gamma \vdash u \equiv_A t} \qquad \frac{\Gamma \vdash t_1 \equiv_A t_2 \qquad \Gamma \vdash t_2 \equiv_A t_3}{\Gamma \vdash t_1 \equiv_A t_3}$$

这里，$\Gamma \vdash t \equiv_A u$ 表示 “$t$ 和 $u$ 是 $\Gamma$ 下类型为 $A$ 的等价的表达式”。第一行是最重要的两条规则：$\beta$ 和 $\eta$。第二行的规则用于把等价关系从子表达式拓展到大的表达式，作用类似于 reduction 中的 evaluation context。第三行的规则把 $\Gamma \vdash t \equiv_A u$ 变成了一个等价关系。容易证明，如果 $\Gamma \vdash t \equiv_A u$，那么 $\Gamma \vdash t : A$ 且 $\Gamma \vdash u : A$。这里，等价关系定义中提到类型是必要的，因为否则 $\eta$ 规则可能会出现类型错误的情况。

有了等价关系后，我们还需要定义什么是 normal form。在基于 reduction 的语义中，normal form 就是不能继续 reduce 的表达式。然而，等价关系中不存在 reduction 的概念，所以我们必须单独定义 normal form 长什么样。直觉上，我们希望 normal form 满足如下的性质：

- normal form 是 $\beta$-short 的：normal form 中不存在形如 $(\lambda x.t)\ u$ 的 $\beta$-redex
- normal form 是 $\eta$-long 的：每个函数类型的 normal form 都形如 $\lambda x.t$

这里，形如 $\lambda x.t$ 的表达式可能是函数类型的 normal form。然而，它有不能出现在 normal form 内部 $\square\ u$ 的位置。能够出现在 $\square\ u$ 的位置的，只能是自由变量，或是卡在自由变量上的计算。例如 $x$ 或 $x\ t$。这种卡在自由变量上的表达式称为 neutral term/中性表达式。现在，我们可以通过归纳同时定义 normal form 的集合 $\mathtt{Nf}_A^\Gamma$（其中的 normal form 用 $v, w$ 表示）和 neutral term 的集合 $\mathtt{Ne}_A^\Gamma$（其中的 neutral term 用 $n$ 表示）：

$$\frac{(x : A) \in \Gamma}{x \in \mathtt{Ne}_A^\Gamma} \qquad \frac{n \in \mathtt{Ne}_A^\Gamma \qquad v \in \mathtt{Nf}_A^\Gamma}{n\ v \in \mathtt{Ne}_A^\Gamma}$$

$$\frac{c \in \{\mathtt{yes}, \mathtt{no}\}}{c \in \mathtt{Nf}_{\mathtt{Ans}}^\Gamma} \qquad \frac{n \in \mathtt{Ne}_{\mathtt{Ans}}^\Gamma}{n \in \mathtt{Nf}_{\mathtt{Ans}}^\Gamma} \qquad \frac{v \in \mathtt{Nf}_B^{\Gamma, x:A}}{\lambda x.v \in \mathtt{Nf}_{A\to B}^\Gamma}$$

这里，neutral term 都形如 $x\ \vec{t}$，其中 $\vec{t}$ 中的参数都必须是 normal form。$\mathtt{Ans}$ 的 normal form 包含 $\mathtt{yes}$、$\mathtt{no}$ 和全部 neutral term（因为我们在处理 open term）。$A \to B$ 的 normal form 则一定是 $\lambda$。有了 normal form 的定义，就可以表述出 normalization 定理了：

**Theorem** (normalization)**.** 对于任意的 $\Gamma \vdash t : A$，都存在 $v \in \mathtt{Nf}_A^\Gamma$ 使得 $\Gamma \vdash t \equiv_A v$

Normalization 定理只声明了 normal form 一定存在。Normal form 的唯一性需要另外证明：

**Theorem 6.1.** 如果 $v, w \in \mathtt{Nf}_A^\Gamma$ 且 $\Gamma \vdash v \equiv_A w$，那么 $v = w$（$\alpha$-等价，即不考虑函数参数名字的问题）。如果 $n, n' \in \mathtt{Ne}_A^\Gamma$ 且 $\Gamma \vdash n \equiv_A n'$，那么 $n = n'$（同样是 $\alpha$-等价）。

*Proof.* 通过对 $\Gamma \vdash v \equiv_A w / \Gamma \vdash n \equiv_A n'$ 的证明进行归纳，同时证明 normal form 和 neutral term 的唯一性。这里由于空间原因不给出证明 □

## 6.3 open term 的 logical relation

和证明 STLC 停机时一样，尝试通过直接归纳证明 normalization 会因为定理本身太弱而失败。我们需要用一个 logical relation 来表达“好的表达式”的概念，加强定理的表述。Logical relation 可以看作是给类型赋予语义。在证明 STLC 停机时，logical relation 给每个类型赋予一个 **closed value/term 的集合**作为语义。现在我们要证明 open term 的 normalization，所以，logical relation 应该给每个类型赋予一个 **open term 的集合**作为语

义。但是，open term 的类型系统中，除了类型本身还有一个 free variable context！所以，证明 strong normalization 的 logical relation 应该形如 $R_A^\Gamma$。

注意到，$R_A^\Gamma$ 和前一节的 $R_A^\Phi$ 非常相似！事实上，normalization 的 logical relation 也是一个 Kripke logical relation，而它的“可能世界”$W$ 就是 `Context`。`Context` 上也有一个自然的“可达性”关系：

$$\Gamma \subseteq \Gamma' \text{ iff } \forall (x : A) \in \Gamma, (x : A) \in \Gamma'$$

Kripke logical relation 的直觉含义，也和 free variable context 是吻合的：

- 可能世界 $W$ 的直觉含义是“语法中没有的语义环境”。而对于 open term 来说，在类型和表达式自身中确实没有直接存储 free variable context，但在考虑一个 open term 的性质时 free variable context 是必要的
- $\Gamma \subseteq \Gamma'$ 是这里的“可达性”关系。在 open term 的化简过程中，一个表达式确实可能在不同的 free variable context 中被调用。例如，假设 $\Gamma \vdash g : A \to B$，那么 $\Gamma \vdash (\lambda f.\lambda x.f\ x)\ g : A \to B$ 中，$g$ 是在 $\Gamma$ 下定义的，但在化简这个表达式的过程中，$g$ 是在 $\Gamma, x : A$ 下被调用的！所以，open term 在化简时，“世界”$\Gamma$ 确实可能改变。不过，已经定义了的 free variable 不能消失：所以 $\Gamma \subseteq \Gamma'$ 的确就是 open term 的可达性关系
- Kripke logical relation 必须尊重可达性。代入到 free variable context 上，就是 $R_A^\Gamma \subseteq R_A^{\Gamma'}$ 必须成立（$\Gamma \subseteq \Gamma'$）。这也非常符合直觉：增添新的、无关的变量，不应该把好的表达式变成坏的

所以，尽管初看之下有些意外，open term 的 logical relation 确实是一个标准的 Kripke logical relation。现在，我们可以开始具体定义 normalization 的（Kripke）logical relation 了。由于我们想要证明 normalization，这个 logical relation 应当满足 $t \in R_A^\Gamma \Rightarrow \exists v \in \mathtt{Nf}_A^\Gamma, \Gamma \vdash t \equiv_A v$。在此基础上，我们可以复用很多 Kripke logical relation 上通用的构造。例如 5.2 中提到的 Kripke implication：

$$\begin{aligned} R_A^\Gamma &\subseteq \mathtt{Term} \\ R_{\mathtt{Ans}}^\Gamma &= \{t \mid \exists v \in \mathtt{Nf}_{\mathtt{Ans}}^\Gamma, \Gamma \vdash t \equiv_{\mathtt{Ans}} u\} \\ R_{A\to B}^\Gamma &= \{t \mid \forall \Gamma' \supseteq \Gamma, u \in R_A^{\Gamma'}, t\ u \in R_B^{\Gamma'}\} \end{aligned}$$

在 $\texttt{Ans}$ 上，我们只要求一个表达式是有 normal form 的，不需要要求额外的性质。在 $A \to B$ 上，我们要求对于任意的好的参数，函数都能给出好的结果。注意这里参数 $u$ 和函数调用 $t\ u$ 都在某个不同的世界 $\Gamma' \supseteq \Gamma$ 中，原因就像刚才所说的：open term 化简的过程中，“世界”/free variable context $\Gamma$ 可能改变。所以，好的函数必须能够处理这种情况。

注意到，$R_{A\to B}$ 并不直接要求表达式有 normal form，这是因为这一点可以从 $t$ 的性质中导出。我们依然可以证明如下的性质：

**Lemma 6.2.** 如果 $t \in R_A^\Gamma$，那么存在 $v \in \texttt{Nf}_A^\Gamma$，使得 $\Gamma \vdash t \equiv_A v$

*Proof.* 对类型 $A$ 进行归纳：

- $A = \texttt{Ans}$。根据定义显然
- $A = A' \to B$。取 $\Gamma' = \Gamma, x : A'$，$u = x$，那么根据 $R^\Gamma_{A'\to B}$ 的性质有 $t\ x \in R_B^{\Gamma,x:A'}$。根据归纳假设，存在 $v \in \texttt{Nf}_B^{\Gamma,x:A'}$ 使得 $\Gamma, x : A' \vdash t\ x \equiv_B v$。根据 $\eta$ 规则，可得 $\Gamma \vdash t \equiv \lambda x.t\ x \equiv \lambda x.v$。根据 normal form 的定义，有 $\lambda x.v \in \texttt{Nf}^\Gamma_{A'\to B}$

□

这里的 logical relation 在函数类型上的构造和 5.2 中几乎一模一样——尽管这两个 logical relation 中“可能世界”的含义完全不同。这是 Kripke logical relation 泛用性的又一体现。为了证明这里的 logical relation 确实是一个 Kripke logical relation，需要证明如下引理：

**Lemma 6.3.** $\forall \Gamma \subseteq \Gamma', R_A^\Gamma \subseteq R_A^{\Gamma'}$

*Proof.* 对类型 $A$ 进行归纳即可。在类型 $A \to B$ 的情况下，可以看到如果不使用 Kripke implication、不支持不同世界下的参数，就不能证明这条引理 □

最后，还可以证明这里的 logical relation 尊重等价关系：

**Lemma 6.4.** 如果 $t \in R_A^\Gamma$ 且 $\Gamma \vdash t \equiv_A u$，那么 $u \in R_A^\Gamma$

*Proof.* 对 $A$ 归纳即可 □

## 6.4 表述并使用 fundamental theorem of logical relation

构造出 logical relation 后，就只需要证明 fundamental theorem of logical relation 了。然而，这次，fundamental theorem of logical relation 的表述需要一些额外的思考。之前的所有 fundamental theorem of logical relation 的证明中，我们都是先定义了 closed term 的 logical relation，再通过 substitution $\sigma$ 把它拓展到任意 open term。但是，本节中，我们的 logical relation 似乎已经能够处理 open term 了。那么，还需要在 fundamental theorem of logical relation 中加入 substitution 吗? 如果加入，又应该如何加入?

于是，在 fundamental theorem of logical relation 的表述上，出现了两种可能的选择。不妨把两种选择都列出来，进行比较。第一种选择，是直接利用 logical relation 中的“世界”来支持 open term：

**Theorem** (v1). $\forall(\Gamma \vdash t : A), t \in R_A^\Gamma$

第二种选择，是采用 Kripke logical relation 上的通用构造。把 logical relation 中的“世界” $\Gamma$ 和 open term $\Delta \vdash t : A$ 中的 context $\Delta$ 看作不同的东西（尽管它们实际上都是 `Context`）。在这种选择上，首先要把 logical relation 拓展到 substitution 上：

$$R_\Delta^\Gamma = \{\sigma \mid \forall(x : A) \in \Delta, x[\sigma] \in R_A^\Gamma\}$$

$R_\Delta^\Gamma$ 有着非常鲜明的直觉意义：它是一个 open substitution。它给 $\Delta$ 中的所有变量提供了“类型正确”的定义。另一方面，它产生的表达式必须在 $\Gamma$ 下才有意义。换言之，$\sigma \in R_\Delta^\Gamma$ 满足如下的规则：

$$\Delta \vdash t : A \Rightarrow \Gamma \vdash t[\sigma] : A$$

现在，我们可以表述出第二个版本的 fundamental theorem of logical relation 了：

**Theorem** (v2). $\forall(\Delta \vdash t : A), \sigma \in R_\Delta^\Gamma, t[\sigma] \in R_A^\Gamma$

那么，这两个版本的 fundamental theorem 有什么区别、哪个更好呢? 这里需要从两个方面考虑：如何证明 fundamental theorem，以及如何使用 fundamental theorem 进一步证明 strong normalization。上面的 (v1) 使用起来是非常方便的。由于 $R_A^\Gamma$ 中的表达式都有 normal form, 证明了 $t \in R_A^\Gamma$ 也就证明了 $t$ 有 normal form。然而，尝试证明 (v1) 时，会在证明 $t = \lambda x.t'$ 的情况时卡住。我们需要证明对任意好的参数 $u$，$(\lambda x.t')\ u \in R_B^\Gamma$。通过 $\beta$

规则我们可以得到 $(\lambda x.t')\ u \equiv t'[x \mapsto u]$：(v2) 中的 substitution 出现了，而 (v1) 的归纳假设无法处理这个表达式！

问题的根源，在于 open term 的化简中，变量同时扮演着两个不同的角色：它们有时是自由变量，有时则是参数。例如，考虑一个表达式 $\Gamma, x : A \vdash t : B$，变量 $x$ 在 $t$ 中可能是一个自由变量。当我们化简 $\Gamma \vdash \lambda x.t : A \to B$ 时就是如此。但与此同时，$x$ 也有可能是一个参数。在化简 $\Gamma \vdash (\lambda x.t)\ u$ 时就是如此。这时，$x$ 应当被替换为参数 $u$。这正是 (v1) 没有处理的一种情况。而在 (v2) 中，变量的两种角色都能得到处理。“自由变量”的情况出现在 logical relation $R^\Gamma_A$ 中，而 substitution $\sigma \in R^\Gamma_\Delta$ 解决了 $\Delta \vdash t : A$ 中的变量是参数的情况。所以，定理 (v1) 无法直接证出。(v2) 才是正确的选择。

## 6.5 logical relation 的 saturated set 结构

我们还剩下一个问题没有处理：应该如何使用上面的定理 (v2)？为了最终证明 normalization，我们还是要回到 (v1) 的结论：$t \in R^\Gamma_A$。因此，我们需要证明 **(v2) 能推出 (v1)**。幸运的是，由于“任意 substitution $\sigma$”的灵活性，我们可以使用单位 substitution $\mathtt{id}_\Gamma$（对于任意的 $(x : A) \in \Gamma$，$x[\mathtt{id}] = x$）来推出 (v1)。由于 $t[\mathtt{id}] = t$，只需要使用 (v2) 证明 $t[\mathtt{id}] \in R^\Gamma_A$ 即可。

现在，观察 (v2) 的定义，证明 normalization 还剩下最后一步：证明 $\mathtt{id} \in R^\Gamma_\Gamma$。展开定义，我们需要证明对于任意的 $(x : A) \in \Gamma$，$x \in R^\Gamma_A$。也就是说，所有自由变量都是“好”的。直觉上这应当成立，但这一性质证明起来并不显然。因为在 $R_{A\to B}$ 中，表达式除了 strongly normalize 还要满足额外的性质：对好的参数给出好的结果。所以，我们需要把定理从自由变量拓展到所有 neutral term：

**Lemma 6.5.** $\mathtt{Ne}^\Gamma_A \subseteq R^\Gamma_A$

*Proof.* 对 $A$ 进行归纳：

- $A = \mathtt{Ans}$。根据定义 $R^\Gamma_{\mathtt{Ans}} = \mathtt{Nf}^\Gamma_{\mathtt{Ans}} \supseteq \mathtt{Ne}^\Gamma_{\mathtt{Ans}}$
- $A = A' \to B$。假如有 $n \in \mathtt{Ne}^\Gamma_{A'\to B}$，我们证明 $n \in R^\Gamma_{A'\to B}$。假设有 $\Gamma' \supseteq \Gamma$ 和 $u \in R^{\Gamma'}_{A'}$，根据 Lemma 6.2 存在 $v \in \mathtt{Nf}^{\Gamma'}_{A'}$，使得 $\Gamma' \vdash u \equiv_{A'} v$。所以有 $\Gamma' \vdash n\ u \equiv_B n\ v$。根据定义 $n\ v \in \mathtt{Ne}^{\Gamma'}_B$（需要把 $n$ 从 $\Gamma$ 提升到 $\Gamma'$，单独证明 $\mathtt{Ne}^\Gamma_A$ 也是一个 Kripke logical relation 即可）。根据归纳假设 $n\ v \in R^{\Gamma'}_B$。根据 Lemma 6.4 $n\ u \in R^{\Gamma'}_B$。由于 $\Gamma'$ 和 $u$ 是任意的，所以 $n \in R^\Gamma_{A'\to B}$

□

证明了这条引理之后，可以发现我们的 logical relation 有着如下的三明治结构：

$$\mathtt{Ne}^\Gamma_A/_\equiv \subseteq R^\Gamma_A \subseteq \mathtt{Nf}^\Gamma_A/_\equiv$$

其中 $S^\Gamma_A/_\equiv$ 表示 $S^\Gamma_A$ 在 $\equiv$ 下的闭包 $\{t \mid \exists u \in S^\Gamma_A, \Gamma \vdash t \equiv_A u\}$。在几乎（如果不是全部）类型系统的 normalization 证明中，都能找到这种共通的结构。满足这条三明治性质且尊重 $\equiv$ 的 logical relation 被称为 *saturated set*（在基于 reduction 的 normalization 中尊重 $\equiv$ 要替换成某些更复杂的性质）。本节中，我们证明了构造出的 logical relation 有 saturated set 的结构，并借此证明了 fundamental theorem 能推出 normalization。在证明带 polymorphism 的语言的 normalization 时，由于 $\forall\alpha.A$ 的存在，需要讨论“任意类型”。而这里的“类型”就可以定义为一个任意的 saturated set。

## 6.6 证明 fundamental theorem 与 normalization

我们已经构造出了 open term 的 logical relation，并写出了它的 fundamental theorem。通过证明 $R^\Gamma_A$ 是一个 saturated set，我们得到了利用 fundamental theorem 证明 strong normalization 的办法。现在，整个证明就只剩下证明 fundamental theorem 了：

**Theorem 6.6.** $\forall(\Delta \vdash t : A), \sigma \in R^\Gamma_\Delta, t[\sigma] \in R^\Gamma_A$

*Proof.* 对 $t$ 进行归纳即可，和第一节中 closed term 的 fundamental theorem 的证明几乎一模一样：

- $t = x$，根据 $R^\Gamma_\Delta$ 的性质显然
- $t \in \{\mathtt{yes}, \mathtt{no}\}$，$A = \mathtt{Ans}$。根据 $R^\Gamma_{\mathtt{Ans}} = \mathtt{Nf}^\Gamma_{\mathtt{Ans}}$ 的定义显然
- $t = \lambda x.t'$，$A = A' \to B$。此时 $\Delta, x : A' \vdash t' : B$。假设有 $\Gamma' \supseteq \Gamma$ 和 $u \in R^{\Gamma'}_{A'}$，根据定义 $(\sigma, x \mapsto u) \in R^{\Gamma'}_{\Delta, x:A'}$。根据归纳假设 $t'[\sigma, x \mapsto u] \in R^{\Gamma'}_B$。容易证明 $\Gamma \vdash (\lambda x.t')[\sigma]\ u \equiv_B t'[\sigma, x \mapsto u]$。所以根据 Lemma 6.4 $\lambda x.t' \in R^\Gamma_{A' \to B}$
- $t = t_f\ t_a$。此时 $\Delta \vdash t_f : A' \to A, \Delta \vdash t_a : A'$。根据归纳假设 $t_f \in R^\Gamma_{A' \to A}$，$t_a \in R^\Gamma_{A'}$。根据 $R^\Gamma_{A' \to B}$ 的定义有 $t_f\ t_a \in R^\Gamma_B$

□

**Theorem 6.7** (normalization)**.** 如果 $\Gamma \vdash t : A$，那么存在 $v \in \mathtt{Nf}^\Gamma_A$，使得 $\Gamma \vdash t \equiv_A v$

*Proof.* 根据 Lemma 6.5，$\mathtt{id} \in R^\Gamma_\Gamma$。根据 Theorem 6.6，$t = t[\mathtt{id}] \in R^\Gamma_A$。根据 Lemma 6.2 即可得到满足性质的 $v$ □

## 6.7 从证明中提取出 normalization 算法

基于 reduction 的 normalization 的一个优点是，只要证明了 strong normalization，就自动得到了一个正确的 normalization **算法**：按任意顺序反复应用 reduction 规则直到没有规则可以应用即可。在本节基于等价关系的 normalization 中，normal form 的存在性和 normalization 算法的存在性是分开的。我们证明了每个表达式都有对应的（唯一的）normal form，但还没有一个算法能算出这个 normal form，也没有证明这种算法存在。

幸运的是，normalization 的证明和 logical relation 的构造是可以指导 normalization 算法的设计的。我们可以从 normalization 的证明中提取出一个 normalization 算法。而且，这个算法属于一类叫作 *normalization-by-evaluation*（NBE）的算法。NBE 算法比起反复应用 reduction 要高效得多。

那么，要如何从证明中提取出算法呢？在前面的 normalization 证明中，有如下的 saturated set 结构：

$$\mathtt{Ne}^\Gamma_A/_\equiv \subseteq R^\Gamma_A \subseteq \mathtt{Nf}^\Gamma_A/_\equiv$$

夹在中间的 $R^\Gamma_A$ 是一个 `Term` 上的 logical relation。但是，如果我们给 $R^\Gamma_A$ 加上**结构**，给“$t \in R^\Gamma_A$”的证明赋予一个有计算意义的**证据**，就能马上把证明转换成算法。假设 $R^\Gamma_A$ 对应的“证据”叫作“值”的话，saturated set 的结构就会对应下面两个函数：

- 一个把 neutral term 嵌入到值的函数（记作 $\uparrow^\Gamma_A$）
- 一个从值中提取出 normal form 的读回函数（记作 $\downarrow^\Gamma_A$）

Fundamental theorem 的证明则会变成一个“求值”算法（记作 $[\![t]\!]_\sigma$）：

> 给定任何的 $\Delta \vdash t : A$ 和 $\sigma \in R^\Gamma_\Delta$（$R^\Gamma_\Delta$ 会变成一个变量到“值”的映射），**计算出**一个 $R^\Gamma_A$ 中的“值”

而用 fundamental theorem 证明 normalization 的过程，就是用求值算法得到 normalization 算法的过程。对于 $\Gamma \vdash t : A$，首先我们通过嵌入函数 $\uparrow_A^\Gamma$（Lemma 6.5）构建一个初始求值环境：

$$\uparrow \Gamma = x \mapsto \uparrow_A^\Gamma x \qquad (x : A) \in \Gamma$$

然后利用求值算法（fundamental theorem）得到一个值 $[\![t]\!]_{\uparrow\Gamma}$。最后，利用读回函数 $\downarrow_A^\Gamma$（Lemma 6.2）得到 $t$ 对应的 normal form。组合起来，normalization 函数可以定义为：

$$\mathtt{nf}_A^\Gamma\ t = \downarrow_A^\Gamma [\![t]\!]_{\uparrow\Gamma}$$

这一过程也是“normalization by evaluation”这一名字的由来：通过求值实现 normalization。现在，只需要涉及 $R_A^\Gamma$ 对应的“值”（记作 $\mathtt{Value}_A^\Gamma$），并写出三条定理对应的函数即可。$\mathtt{Value}_A^\Gamma$ 的设计，同样和 logical relation 的定义高度相似。对于 $\mathtt{Ans}$，$\mathtt{Value}$ 就是 $\mathtt{Nf}$。对于 $A \to B$，它的 logical relation 定义为“参数是好的能推出结果是好的”，它对应的值就是某种函数：给定好的参数，**算出**好的结果。这里使用闭包（$\lambda$ 表达式 + 求值环境）来表示。现在，值和求值环境可以如此定义：

$$\begin{aligned}
\mathtt{Value}_{\mathtt{Ans}}^\Gamma &= \mathtt{Nf}_{\mathtt{Ans}}^\Gamma \\
\mathtt{Value}_{A\to B}^\Gamma &= \{(\lambda x.t, \sigma) \mid \Delta, x : A \vdash t : B \text{ and } \sigma \in \mathtt{Env}_\Delta^\Gamma\}
\end{aligned}$$

$$\begin{aligned}
\mathtt{Env}_\varnothing^\Gamma &= \{\varnothing\} \\
\mathtt{Env}_{\Delta, x:A}^\Gamma &= \{\sigma, x \mapsto v \mid \sigma \in \mathtt{Env}_\Delta^\Gamma \text{ and } v \in \mathtt{Value}_A^\Gamma\}
\end{aligned}$$

各个函数可以如此定义：

$$\begin{aligned}
\uparrow_A^\Gamma &: \mathtt{Ne}_A^\Gamma \to \mathtt{Value}_A^\Gamma \\
\uparrow_{\mathtt{Ans}}^\Gamma n &= n \\
\uparrow_{A\to B}^\Gamma n &= (\lambda x.n\ x, \uparrow \Gamma) \qquad (x \notin \Gamma)
\end{aligned}$$

$$\begin{aligned}
\uparrow \Gamma &: \mathtt{Env}_\Gamma^\Gamma \\
\uparrow \varnothing &= \varnothing \\
\uparrow (\Gamma, x : A) &= \uparrow \Gamma, x \mapsto \uparrow_A^\Gamma x
\end{aligned}$$

$$
\begin{aligned}
\downarrow^{\Gamma}_{A} &: \mathtt{Value}^{\Gamma}_{A} \to \mathtt{Nf}^{\Gamma}_{A} \\
\downarrow^{\Gamma}_{\mathtt{Ans}}\ v &= v \\
\downarrow^{\Gamma}_{A\to B}\ (\lambda x.t, \sigma) &= \lambda y.\ \downarrow^{\Gamma, y:A}_{B}\ [\![t]\!]_{\sigma, x\mapsto \uparrow^{\Gamma}_{A} y} \qquad (y \notin \Gamma)
\end{aligned}
$$

$$
\begin{aligned}
[\![\_]\!]_{\_} &: \mathtt{Term}^{\Delta}_{A} \to \mathtt{Env}^{\Gamma}_{\Delta} \to \mathtt{Value}^{\Gamma}_{A} \\
[\![x]\!]_{\sigma} &= x[\sigma] \\
[\![\mathtt{yes}]\!]_{\sigma} &= \mathtt{yes} \\
[\![\mathtt{no}]\!]_{\sigma} &= \mathtt{no} \\
[\![\lambda x.t]\!]_{\sigma} &= (\lambda x.t, \sigma) \\
[\![t\ u]\!]_{\sigma} &= [\![t]\!]_{\sigma} \cdot [\![u]\!]_{\sigma}
\end{aligned}
$$

$$
\begin{aligned}
\_ \cdot \_ &: \mathtt{Value}^{\Gamma}_{A\to B} \to \mathtt{Value}^{\Gamma}_{A} \to \mathtt{Value}^{\Gamma}_{B} \\
(\lambda x.t, \sigma) \cdot v &= [\![t]\!]_{\sigma, x\mapsto v}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{nf}^{\Gamma}_{A} &: \mathtt{Term}^{\Gamma}_{A} \to \mathtt{Nf}^{\Gamma}_{A} \\
\mathtt{nf}^{\Gamma}_{A}\ t =& \downarrow^{\Gamma}_{A} [\![t]\!]_{\uparrow\Gamma}
\end{aligned}
$$

下面以 $\varnothing \vdash (\lambda x.\lambda y.x\ y)\ (\lambda y.y) : \mathtt{Ans} \to \mathtt{Ans}$ 为例, 演示算法的运行过程:

$$
\begin{aligned}
&\mathtt{nf}^{\varnothing}_{\mathtt{Ans}}\ (\lambda x.\lambda y.x\ y)\ (\lambda y.y) \\
=& \downarrow^{\varnothing}_{\mathtt{Ans}} [\![(\lambda x.\lambda y.x\ y)\ (\lambda y.y)]\!]_{\uparrow\varnothing} \\
=& \downarrow^{\varnothing}_{\mathtt{Ans}} [\![(\lambda x.\lambda y.x\ y)\ (\lambda y.y)]\!]_{\varnothing} \\
=& \downarrow^{\varnothing}_{\mathtt{Ans}} [\![\lambda x.\lambda y.x\ y]\!]_{\varnothing} \cdot [\![\lambda y.y]\!]_{\varnothing} \\
=& \downarrow^{\varnothing}_{\mathtt{Ans}} (\lambda x.\lambda y.x\ y, \varnothing) \cdot (\lambda y.y, \varnothing) \\
=& \downarrow^{\varnothing}_{\mathtt{Ans}} [\![\lambda y.x\ y]\!]_{x\mapsto(\lambda y.y,\varnothing)} \\
=& \downarrow^{\varnothing}_{\mathtt{Ans}} (\lambda y.x\ y, [x \mapsto (\lambda y.y, \varnothing)]) \\
=& \lambda z_1.\ \downarrow^{z_1:\mathtt{Ans}}_{\mathtt{Ans}} [\![x\ y]\!]_{x\mapsto(\lambda y.y,\varnothing), y\mapsto z_1} \\
=& \lambda z_1.\ \downarrow^{z_1:\mathtt{Ans}}_{\mathtt{Ans}} [\![x]\!]_{x\mapsto(\lambda y.y,\varnothing), y\mapsto z_1} \cdot [\![y]\!]_{x\mapsto(\lambda y.y,\varnothing), y\mapsto z_1} \\
=& \lambda z_1.\ \downarrow^{z_1:\mathtt{Ans}}_{\mathtt{Ans}} (\lambda y.y, \varnothing) \cdot z_1 \\
=& \lambda z_1.\ \downarrow^{z_1:\mathtt{Ans}}_{\mathtt{Ans}} [\![y]\!]_{y\mapsto z_1} \\
=& \lambda z_1.\ \downarrow^{z_1:\mathtt{Ans}}_{\mathtt{Ans}} z_1 \\
=& \lambda z_1.z_1
\end{aligned}
$$

# 7 走向抽象：Kripke 语义

到目前为止，我们看到了许多形态各异的 logical relation。但它们都是用于证明操作语义的性质的 logical relation，事实上，logical relation 中的许多构造，也可以拓展到指称语义上。在 logical relation 中，我们给每个类型赋予了一个值/表达式的集合，在指称语义中，我们则把每个类型的语义泛化为某种数学对象的集合。指称语义有很多非常重要的应用：

- 通过把类型和表达式翻译成数学对象，可以提供理解语言语义的新的角度、复用现有的数学方法来研究语义
- 为了证明语言**不能**做到某些事，例如表达不出某个特定的函数，可以用构造反模型的方法：构造一个特殊的语义模型，其中某些性质不成立或是某些对象不存在。由于语言中成立的在语义模型中也一定成立，反模型的存在意味着语言中没有对应的性质、构造不出对应的对象

指称语义一般不会被归类到 logical relation 下，但指称语义和 logical relation 的构造往往是高度相似的，所以这里对它与 logical relation 的联系进行介绍。本节中的内容需要一点基础的范畴论知识。空间所限，本节不会用一个具体的语言来演示，只会简单地介绍一些常见构造。

## 7.1 前置知识：Lambek correspondence

假设现在我们希望把 STLC 翻译成某种数学对象。一个很自然的想法是，把类型 $A$ 翻译成一个集合 $[\![A]\!]$，把表达式翻译成 $[\![A]\!]$ 中的元素。为了处理 open term 中的自由变量，我们可以使用和 logical relation 一样的办法：在翻译表达式时，接受一个“substitution”，把 context 中的变量映射到对应类型中的数学对象。具体来说，我们把 $\Gamma$ 翻译成一个变量到数学对象的映射的集合 $[\![\Gamma]\!]$，使得对于每个 $\rho \in [\![\Gamma]\!]$ 和 $(x : A) \in \Gamma$，都有 $\rho(x) \in [\![A]\!]$。现在，给定一个映射 $\rho \in [\![\Gamma]\!]$，我们可以把每个表达式 $\Gamma \vdash t : A$ 翻译成一个数学对象 $[\![t]\!]_\rho \in [\![A]\!]$。为了说明翻译的正确性，需要证明如下的 soundness 定理：

**Theorem** (soundness)**.** 如果 $\Gamma \vdash t : A$，那么对于任意的 $\rho \in [\![\Gamma]\!]$，都有 $[\![t]\!]_\rho \in [\![A]\!]$

可以看到，这其实就是 fundamental theorem of logical relation！事实上，logical relation 就是上面描述的翻译过程的一个特例：

| 指称语义 | logical relation |
|---|---|
| $[\![A]\!]$ | $R_A$ |
| $[\![\Gamma]\!] = \{\rho \mid \forall (x : A) \in \Gamma, \rho(x) \in [\![A]\!]\}$ | $R_\Gamma = \{\sigma \mid \forall (x : A) \in \Gamma, x[\sigma] \in R_A\}$ |
| soundness: $[\![t]\!]_\rho \in [\![A]\!]$ | fundamental theorem: $t[\sigma] \in R_A$ |

下一个问题是，logical relation 中的构造，是否也能应用到指称语义上呢？例如，在前面的各个 logical relation 的构造中，函数类型 $A \to B$ 的定义中，都有一条“给定好的参数，得到好的结果”的共同要求。那么，指称语义中是否也会有类似的构造呢？一个很自然的想法是，我们可以把函数类型翻译成集合上函数的集合，把 $\lambda$ 表达式翻译成一个真正的函数：

$$[\![A \to B]\!] = [\![A]\!] \to [\![B]\!] \qquad [\![\lambda x.t]\!]_\rho = a \mapsto [\![t]\!]_{\rho, x \mapsto a}$$

然而，这种翻译方式是不完备的、不够精确的。假设源语言中有一个自然数类型 `nat`，那么指称语义中它最自然的翻译是自然数集 $\mathbb{N}$。然而，在集合论中，自然数到自然数的全体函数的集合 $\mathbb{N} \to \mathbb{N}$ 是不可数的。而全体 STLC 表达式的集合是可数的，因为 STLC 的语法是归纳地定义的。所以，$[\![\texttt{nat} \to \texttt{nat}]\!] = \mathbb{N} \to \mathbb{N}$ 中，只有很少的一部分对象是由 STLC 表达式翻译得到的，剩下的都是与 STLC 无关的对象。

语义上的这种不精确性对语义的实用性有着很大的影响。例如，我们想证明某个特定的 $\mathbb{N} \to \mathbb{N}$ 的函数无法在 STLC 中表达，那么 $[\![\texttt{nat} \to \texttt{nat}]\!] = \mathbb{N} \to \mathbb{N}$ 的语义就派不上用场，因为它不能区分 STLC 能表达的和不能表达的自然数上的函数。因此，我们需要更精确的语义。

既然函数类型不能翻译成函数，它应该被翻译成什么呢？这里，范畴论给出了一个很好的答案。范畴论中，有 exponential object 的概念。它刻画了“函数”和“态射”所具有的性质。函数构成的集合就是集合构成的范畴 `Set` 中的 exponential object。Exponential object $C^B$（相当于函数 $B \to C$）可以通过如下的同构来定义：

$$\texttt{Hom}(A \times B, C) \simeq \texttt{Hom}(A, B^C)$$

这里 $\texttt{Hom}(A, B)$ 表示某个范畴中的对象 $A$ 到 $B$ 之间的态射集合，$A \times B$ 是范畴中的积对象。现在，我们可以尝试把 STLC 翻译到某个范畴 $\mathcal{C}$ 中、把 $A \to B$ 翻译成 $\mathcal{C}$ 中的 exponential object $[\![B]\!]^{[\![A]\!]}$。Context $\Gamma$ 可以利用 $\mathcal{C}$ 中的乘积来翻译：

$$[\![\varnothing]\!] = \mathbf{1} \qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!] \times [\![A]\!]$$

这里，**1** 是 $\mathcal{C}$ 中的 terminal object。接下来，一个表达式 $\Gamma \vdash t : A$，就可以翻译成一个 $[\![t]\!] \in \mathtt{Hom}_{\mathcal{C}}([\![\Gamma]\!], [\![A]\!])$ 的态射。有了翻译的框架后，让我们重新审视 exponential object 的定义。Exponential object 定义中的同构，如果代入 STLC 的翻译的话，就是：

$$\mathtt{Hom}([\![\Gamma, x : A]\!], [\![B]\!]) \simeq \mathtt{Hom}([\![\Gamma]\!], [\![A \to B]\!])$$

现在，这个同构的意义就清晰了起来：左侧到右侧就是从 $\Gamma, x : A \vdash t : B$ 到 $\Gamma \vdash \lambda x.t : A \to B$。右侧到左侧就是从 $\Gamma \vdash t : A \to B$ 到 $\Gamma, x : A \vdash t\ x : B$。同构的两个方向刚好对应了 $\lambda$ 和函数调用、对应函数类型的引入和消去规则。所以，exponential object 的确是翻译 STLC 中函数类型的一个很好的选择。

事实上，有一个名为 Lambek correspondence 的著名结论，说的就是 STLC 和 Cartesian Closed Category （CCC，即任何两个对象之间的乘积和 exponential object 都存在的范畴）是等价的。对于**任意的** CCC $\mathcal{C}$，只要给基础类型和常量在 $\mathcal{C}$ 中找到对应的 object/态射作为语义，就能把完整的 STLC 翻译到 $\mathcal{C}$ 中。STLC 自己也构成一个 CCC，称为**语法范畴**。而 STLC 到某个 CCC $\mathcal{C}$ 的翻译，则会对应一个语法范畴到 $\mathcal{C}$ 的、保持 CCC 结构的函子。Lambek correspondence 提供了一系列的语义模型。前面提到的集合语义也是 Lambek correspondence 的一个特例：取 $\mathcal{C} = \mathtt{Set}$ 即可。

## 7.2 logical relation 的范畴解释

我们已经看到，logical relation 可以看成把类型翻译成集合的指称语义的一个特例。而把类型翻译成集合的语义，又是范畴语义的一个特例。那么，logical relation 中的通用构造，例如“给定好的参数，得到好的结果”，也应该有对应的通用范畴构造。

Logical relation 的特殊性在于，它给每个类型赋予了一个满足某些性质的子集作为语义。在证明 STLC 停机时，每个类型对应一个 closed term/closed value 的子集。在证明 STLC 的 normalization 时，每个类型则对应一个 open term 的子集（证明 normalization 时，context/Kripke world 的处理需要使用 Kripke 语义来描述，后面会看到这一点）。所以，在 logical relation 对应的语义中，每个类型首先要对应“某种东西”的集合（例如 closed term），然后每个类型实际的语义是对应的“某种东西”的集合的一个子集。

假设语法范畴是 $\mathcal{C}$，首先需要有一个“某种东西”的函子 $|\_| : \mathcal{C} \to \mathtt{Set}$。例如，当“某种东西”是 closed term 时，$|A|$ 可以取 $\mathtt{Hom}(\mathbf{1}, A)$。现在，类型 $A$ 上的一个 logical relation 就是 $|A|$ 的一个子集。这在范畴论中可以表示成一个 monomorphism $\bullet \hookrightarrow |A|$。如此一来，全体 logical relation 就构成了一个新的范畴 $\mathtt{Set} \downarrow |\_|$：

- $\mathtt{Set} \downarrow |\_|$ 中的对象是形如 $(R \in \mathtt{Set}, A \in \mathcal{C}, f : R \hookrightarrow |A|)$ 的三元组。直觉上，这是 $|A|$ 上的一个 logical relation：$A$ 是类型，$R$ 和 $f$ 构成了 $|A|$ 的一个子集
- $\mathtt{Set} \downarrow |\_|$ 中，从 $(R, A, f)$ 到 $(R', A', f')$ 的箭头是使得下面的交换图交换的 $(h, \hat{h})$ 二元组：

$$\begin{array}{ccc} R & \xhookrightarrow{f} & |A| \\ \downarrow{\scriptstyle \hat{h}} & & \downarrow{\scriptstyle |h|} \\ R' & \xhookrightarrow{f'} & |A'| \end{array}$$

  直觉上，这是一个从 $A$ 到 $A'$ 的、**STLC 中的**函数 $h$，满足如果 $x \in R$，那么 $|h|(x) \in R'$

现在，我们的得到了一个“全体 logical relation”构成的范畴 $\mathtt{Set} \downarrow |\_|$。一个具体的 logical relation 就是这个范畴中的一个对象。在使用 logical relation 证明类型系统的性质时，我们往往给每个类型找一个对应的 logical relation。所以，STLC 的一个 logical relation 就是一个 $\mathcal{C} \to \mathtt{Set} \downarrow |\_|$ 的函子 $[\![\_]\!]$。每个类型 $A$ 对应的 logical relation 应当是 $|A|$ 的一个子集。所以，我们额外要求 $[\![\_]\!]$ 满足如下的交换图：

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{[\![\_]\!]} & \mathtt{Set} \downarrow |\_| \\ & \searrow & \downarrow{\scriptstyle \pi} \\ & & \mathcal{C} \end{array}$$

其中 $\pi$ 是忘掉 logical relation 的额外性质的函子：$\pi(R, A, f) = A$。

有了用范畴语言表达 logical relation 的框架之后，就可以开始考虑 logical relation 的具体构造了。考虑函数类型 $A \to B$ 对应的 logical relation。由于 logical relation 可以看成范畴语义的一个特例，而 STLC 的范畴语义中，函数类型被翻译为 exponential object，一个很自然的想法是：把 $A \to B$ 翻

译成 $\mathtt{Set}\downarrow|\_|$ 中的 exponential object。那么，$\mathtt{Set}\downarrow|\_|$ 中的 exponential object 长什么样呢？事实上，它就是“给定好的参数，得到好的结果”：

$$(R_B)^{R_A} \subseteq |A \to B|$$
$$(R_B)^{R_A} = \{f \in |A \to B| \mid \forall a \in R_A, |\mathtt{app}|(f, a) \in R_B\}$$

这里简单起见，直接用 $R_A$ 表示 logical relation $R \subseteq |A|$。$\mathtt{app} : (A \to B) \times A \to B$ 是 $\mathcal{C}$ 中的函数调用操作。容易证明，$(R_B)^{R_A}$ 的确就是 $\mathtt{Set}\downarrow|\_|$ 中的 exponential object。这给函数类型的 logical relation 构造提供了新的支持：之前所有函数类型的 logical relation 都长得差不多不是一个巧合，而是结构上必须如此。

最后，让我们考虑 fundamental theorem 的范畴对应。在 logical relation 的范畴表述中，$[\![\_]\!]$ 其实同时包含了 logical relation 的构造和 fundamental theorem 的证明。在构造 $[\![\_]\!]$ 的 object 部分时，我们给出了类型和 context 对应的 logical relation。而在构造 $[\![\_]\!]$ 的态射部分时，给定一个 $\Gamma \vdash t : A$，我们需要构造一个 $[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!]$。展开 $\mathtt{Set}\downarrow|\_|$ 中态射的定义，并加上 $\pi \cdot [\![\_]\!] = \mathtt{id}$ 这一额外条件，就能发现 $[\![t]\!]$ 的存在其实就是 fundamental theorem。这里由于空间原因不展开讨论，读者可以自行验证。

范畴语义还给我们提供了一种一劳永逸地证明 fundamental theorem 的办法。由于我们可以在任意 CCC 中解释 STLC，只要说明 $\mathtt{Set}\downarrow|\_|$ 是一个 CCC，就能得到 $[\![\_]\!] : \mathcal{C} \to \mathtt{Set}\downarrow|\_|$。只要在 logical relation 中把函数解释为 exponential object（给定好的参数，得到好的结果），把 context 解释为乘积（对于 context 中的每个变量，提供一个对应类型的好的值），$[\![\_]\!]$ 就会成为一个免费的 fundamental theorem 的证明。我们也可以利用这一结论反过来指导 logical relation 的设计：性质良好的 logical relation，就应该把函数类型解释成 exponential object。所以，当我们设计 logical relation 时，可以只设计 base type 上的 logical relation，然后在函数类型等复合类型上直接套用一般的构造。这样得出来的 logical relation，往往就是我们想要的。

## 7.3 Kripke 语义与 Kripke logical relation

我们已经看到了普通的 logical relation 如何使用范畴语言表述，以及这样带来的好处。更加一般的 Kripke logical relation 也在范畴语言中有非常自然的对应。在简单的集合指称语义中，我们把每个类型解释成一个集合，即 $\mathtt{Set}$ 中的一个 object。在 Kripke 语义中，每个类型被解释成一

**族**以可能世界 $W$ 为索引的集合，而且要尊重可达性关系。Kripke logical relation/Kripke 语义的这些条件，可以用范畴的语言非常简洁地表达为：

> Kripke 语义中，有一个“可能世界”的范畴 $W$。我们把每个类型解释为一个 $W$ 到 $\mathtt{Set}$ 的函子 $F \in [W^{\mathtt{op}}, \mathtt{Set}]$。换言之，一个 Kripke 语义就是一个函子 $[\![\_]\!] : \mathcal{C} \to [W^{\mathtt{op}}, \mathtt{Set}]$

其中，$W^{\mathtt{op}}$ 是 $W$ 的对偶范畴（把 $W$ 中每个箭头的起点和终点对调）。取对偶是因为范畴论中形如 $W^{\mathtt{op}} \to \mathtt{Set}$ 的函子的性质被研究得更加透彻。读者可以自行验证，当 $W$ 是一个偏序集（每两个对象之间至多有一个箭头，表示 $\leqslant$），并且 $[\![A]\!]$ 把每个箭头 $w \leqslant w'$ 映射到一个子集到父集的嵌入时，这个范畴定义等价于 Kripke logical relation 的定义。

那么，各个具体地类型又应该翻译成什么呢？大部分构造，例如乘积，在函子范畴 $[W^{\mathtt{op}}, \mathtt{Set}]$（常常称为 $W$ 上的 presheaf category，其中的函子称为 prehseaf）中只需要逐点定义即可：

$$(F \times G)(w) = F(w) \times G(w)$$

但函数/exponetial object 的定义是个例外。逐点地定义两个函子之间的 exponential object，就无法定义出 $G^F$ 在态射上的部分，读者可以自行验证这一点。这里出现的问题，和 Kripke logical relation 中函数类型的语义是一样的。函数被创建时的世界 $w$ 和被使用时的世界 $w'$ 可能是不一样的，所以函数收到的参数也不一定来自 $w$，可能来自 $w'$。我们只知道能够从 $w$ 到达 $w'$。所以，函数的语义必须把这一点也考虑在内。同理，presheaf 上的 exponential object $G^F$ 必须对任意可达的世界定义：

$$\begin{aligned} G^F(w) &= [\mathtt{Hom}(-, w) \times F, G] \\ &= \prod_{w' \in W} \mathtt{Hom}(w', w) \times F(w') \to G(w') \end{aligned}$$

如果代入 $W$ 是一个偏序集的情况，那么 $G^F(w)$ 的定义就等价于：

$$G^F(w) = \{f \mid \forall w' \geqslant w, x \in F(w'), f(x) \in G(w')\}$$

可以看到，这正是 Kripke implication！所以，范畴语言也给 Kripke implication 的构造提供了新的支持：函数类型上性质良好的 Kripke 语义，一定是 Kripke implication！

有了 Kripke 语义之后，只需要把 logical relation 的范畴 $\mathtt{Set} \downarrow |\_|$ 中的 $\mathtt{Set}$ 替换成 $[W^{\mathtt{op}}, \mathtt{Set}]$，就能得到 Kripke logical relation 的范畴表述。同样地，我们可以用范畴语言表述出 fundamental theorem，并利用语法范畴的性质一劳永逸地证明 fundamental theorem of *Kripke* logical relation。$[W^{\mathtt{op}}, \mathtt{Set}] \downarrow |\_|$ 中的 exponetial object 如下（假设 $W$ 是一个偏序集）：

$$(R_B)^{R_A}(w) = \{f \in |A \to B|(w) \mid \forall w' \geqslant w, a \in R_A^{w'}, |\mathtt{app}|(w')(\uparrow_w^{w'} f, a) \in R_B^{w'}\}$$

其中 $\uparrow_w^{w'}$ 是 $|i|$（$i : w' \to w$ 是 $w \leqslant w'$ 的证据）。在 Kripke logical relation 中 $\uparrow_w^{w'}$ 是子集到父集的嵌入。可以看到，这正是 Kripke logical relation 中函数类型对应的 logical relation。

## 7.4 Kripke 语义的完备性

在简单的集合语义中，我们把类型解释为集合。这也可以看作是在 $\mathtt{Set}$ 这个范畴/逻辑框架中研究语言的语义。但是，研究语言的语义使用的逻辑框架不一定要是 $\mathtt{Set}$。理论上，我们可以换成任何性质足够丰富的范畴 $\mathcal{T}$。范畴论中，有一类性质和 $\mathtt{Set}$ 类似的范畴，叫作 *elementary topos*。在一个 Elementary topos 内部，支持高阶逻辑和类似于 $\mathtt{Set}$ 的取子集、取幂集等操作。所以，把 $\mathtt{Set}$ 换成其他的 elementary topos，理论上也能研究语言的语义。

除了 $\mathtt{Set}$ 之外，另一类非常常见的 elementary topos 就是 presheaf category $[W^{\mathtt{op}}, \mathtt{Set}]$：Kripke 语义！所以，Kripke 语义可以看成是把研究语义的逻辑框架从 $\mathtt{Set}$ 换成了 presheaf category $[W^{\mathtt{op}}, \mathtt{Set}]$。Kripke 语义比起普通的集合语义更加精确。因为 $\mathtt{Set}$ 中有一些 $\mathtt{Set}$ 特有的、在其他 elementary topos 中不一定成立的性质。例如，在 $\mathtt{Set}$ 中，排中律是成立的。但在 presheaf category 中，排中律就不一定成立。众所周知，STLC 等价于直觉主义自然演绎。所以，很多时候，当我们想要研究 STLC 中能否定义出某个表达式，或是借助 STLC 研究某个命题在直觉主义逻辑中是否可证时，排中律成立的 $\mathtt{Set}$ 就无法使用，需要选择其他的逻辑框架：例如 presheaf category，即使用 Kripke 语义。

除了 $\mathtt{Set}$ 和 presheaf category 之外，还有很多其他 elementary topos/逻辑框架。但是，在这之中，presheaf category 有着非常好的性质：它是完备的。我们可以把语言的语法自身表示成一个 Kripke 语义，这个语义正是 yoneda embedding：

$$\mathtt{y}(A) \in [\mathcal{C}^{\mathtt{op}}, \mathtt{Set}]$$
$$\mathtt{y}(A) = \mathtt{Hom}(-, A)$$

展开定义，有 $\mathtt{y}(A)(\Gamma) = \{t \mid \Gamma \vdash t : A\}$。根据 Yoneda Lemma，$\mathtt{y}$ 是 fully faithful 的。换言之，用 $\mathtt{y}$ 把 $\mathcal{C}$ 嵌入到 $[\mathcal{C}^{\mathtt{op}}, \mathtt{Set}]$ 中没有损失任何信息。由此可以知道，Kripke 语义不仅比集合语义更准确、更好，还是最精确的一种语义。语言中的任何性质，都能通过某种 Kripke 语义加以证明。

# References