关于 Maple Algebra 的这一路

枫聆 (maplegra)

2023 年 5 月 7 日

目录

Logic

Classic Logic

Annotation 1.1. Double negation 和 excluded middle 是 logical equivalent, 关于它们有两个通俗解释:

- Double negation: 如果你想证明 P 为 $true$, 只需要说明 P 不是 $false$ 即可.
- Excluded middle: 任意一个命题 P, 它不是 $true$ 就是 $false$.

Higher-order logic

Annotation 1.2. Second-order logic 在 first-order logic 的基础上, 给其中的谓词也加上了量词. 给谓词加量词意义是什么呢? 谓词实际上可以理解为 varibles 的 properties, 所以相当于你给 variable's properties 也加上了量词. 形如:

$$\exists P.\ P(b)$$

这里的 P 可以看做一个谓词变量 (predicate variable). 从语义上来说 P 可以理解为集合, 更进一步说就是满足某种 property 的 varibles 组成的 set, 所以你对 P 施加一个量词也就是在考虑不同的集合, 此时上面的 formula 可以非正式地理解为存在某个集合 P 包含 b.

Quantifier Elimination

Annotation 1.3. Fourier-Motzkin algorithm 适用于 prenex form 下内部 formula 是一个 conjunction 且里面的每个 conjuncts 只能是一个 literals i.e., $\forall x_1, x_2, \cdots, x_n.\ x_1 > a_1 \wedge x_2 > a_2 \wedge \cdots \wedge x_n > a_n$. 我们将这个算法记为 π, 那么只要我们把任意的 formula 都转换称 DNF,

Annotation 1.4. David Monniaux (CAV'10) 中提到可以利用 SMT-solver 来把任意 quantifer-free formula 转换成 DNF, 这样我们对其中 DNF 中的每一个 clause 都可以分开来使用 Fourier-Motzkin algorithm. 它具体的做法就是, 对于给定的 formula F, 扔给 SAT-solver, 如果是 satisfiable 的, 那么 SAT-solver 就会给出一个 solution, 这个 solution 实际对应一个 conjunction C (由 F 中 atoms 构造的, 如果 solution 某个 variable 使得对应的 atom 是 true, 那么就把这个 atom 丢到 conjunction 中, 反之把 negation of atom 丢到 conjunction 中) 使得 $C \to F$. 这样我们再把 $F \wedge \neg C$ 扔到 SMT-solver 中, 类似一直操作, 直到 unsat.

这种方式存在一个问题, 如果给定 $F \equiv (x_1 = 0 \vee x_1 = 1) \wedge \cdots \wedge (x_n = 0 \vee x_n = 1)$, 那么将产生 2^n 个 conjuncts. 而对于 $\exists x_2, \cdots, x_n.\ F$, 它的 quantifier free 形式却只是 $x_1 = 0 \vee x_1 = 1$ (logically equivalent quantifier-free formula). 显然有点不太 efficient, 这个时候我们不直接用 $F \wedge C$, 而用 $F \wedge \neg\pi(\exists x_2, \cdots, x_n.\ C)$ 作

为新一轮的查询. 例如这里设 $C \equiv (x_1 = 0 \wedge x_2 = 0 \wedge \cdots \wedge x_n = 0)$, 那么 $\pi(C) \equiv x_1 = 0$, 这样我们只需要 call SAT-solver 两次就可以得到我们想要的 DNF.

David Monniaux 提出的优化方法远不止于此，其核心在于对 nested quantifiers 如何处理. 如果此时给定 $F \equiv \exists x.\forall y.\exists z.\ z \geq 0 \wedge ((x \geq z \wedge y \geq z) \vee y \leq 1 - z)$, 显然这个 F 是 valid 的. 我们从内向外一层层去掉 quantifiers.

- 设最里面的 quantifier-free formula 为 F_3, 它并不是 DNF, 尝试转化为 DNF. 假设扔给 SMT-solver 得到 solution $x = 0 \wedge y = 0 \wedge z = 0$, 对应的 conjunction 为 $C_1 \equiv z \geq 0 \wedge x \geq z \wedge y \geq z \wedge y \leq 1 - z$, 那么 $\pi(\exists z.\ C_1) \equiv x \geq 0 \wedge y \geq 0 \wedge y \leq 1$. 再把 $F_3 \wedge \neg\pi(C)$ 丢到 SMT-solver 中得到 solution $x = 0 \wedge y = -1 \wedge z = 2$, 对应的 conjunction 为 $C_2 \equiv z \geq 0 \wedge x \geq z \wedge y \leq 1 - z$, 那么 $\pi(\exists z.\ C_2) \equiv x \geq 0 \wedge y \leq 1$. 如此继续，设 $F_2 \equiv \exists z.F_3$, 最后有 $F_2 \equiv \bigvee \pi(\exists z.\ C_i) \equiv x \geq 0 \vee y \leq 1$.
- 设 $F_1 = \forall y.\ F_2$, 那么显然有 $F_1 \equiv x \geq 0$.
- 最后 $\exists x.F_1 \equiv true$.

注意到如果我们只取 quantifier-free F_2 中的 conjunct $x \geq 0$, 使得 $F_2' \equiv x \geq 0$, 对 $\exists x.\forall y.\ F_2'$ 做 quantifier elimination，最后也能得到 $true$ 这个结果. 这意味着我们不用在最里面那个阶段费尽心思地构造完整的 DNF. 也就是说用 “比较强的 formula” 得到了一个结果，那么显然 “弱一点的 formula” 也是有这个结果的. 但是只有当这个结果是 true 的时候，这么做才有意义，不然出现 false 的时候，你并不能说明它是 false 的, 其他情况给不出原 formula 准确的 quantifier-free form.

我还是想记录这个算法的全过程推理，因为花了不少精力，将来忘记了就比较可惜. 算法如图1, 其每个关键的步骤细节如下:

- 首先是关于这里 $F_0, F_1, \cdots F_n$ 的定义: F_0 是一个形如 $\forall_{B_0}\exists_{B_1}\forall_{B_2}\cdots F$ 的 prenex. 其中 $B_0, B_1, B_2, \cdots$ 就是把相邻地相同的 quantifiers 合起来. 对于任意 $i \geq 0$, $F_i = \forall\neg F_{i+1}$, 注意这里是从 F_0 往后构造 F_i 的. 例如给定 $\forall x.\exists y.\forall z\ G$, 它对应的 F_i 分别为

$$
\begin{aligned}
F_0 &= \forall x.\exists y.\forall z\ G \\
F_1 &= \forall y.\exists z.\ \neg G \\
F_2 &= \forall z.\ G \\
F_3 &= \neg G
\end{aligned}
$$

- 这里 SMT-Test(C, F) 用于判断 $C \wedge F$ 是否是 satisfiable，它的返回值为 (b, C'), 其中 b 是一个 boolean value 表示是否 satisfiable. 当 $b = true$ 时，有 atoms$(C') \subseteq$ atoms(F), 且满足 $C' \Rightarrow F$ and $C \wedge C'$ is satisfiable. 这一点很容易办到，参考上面由 solution 生成对应 conjunction 的过程; 当 $b = false$ 时，$C = false$.

- 其中 $M_0, M_1, \cdots, M_{n-1}$ 初始化为 $true$, 满足对任意的 $i < n$ 有 $F_i \Rightarrow M_i$. 注意在 Q-Test 的每一步都会保持这个条件.

- 最关键的 Q-Test(i, C), 设它的返回值为 (b, C'), 其中 b 表示 $F_i \wedge C$ 表示是否 satisfiable. (性质 1) 如果 $b = true$, 有 $\mathrm{FV}(C) = \mathrm{FV}(F_i)$, 且满足 $C' \Rightarrow F_i$ and $C \wedge C'$ is sat. 那这个算法和 quantifier elimination 有什么关系呢? 考虑 Q-Test$(0, true)$, 当它第一个返回值为 $true$ 的时候, 就有 $C' \Rightarrow F_0$, 并且 C' 是 quantifier-free 的. 这个时候我们就得到了一个比如强的 formula C', 拿它可以进一步去测试 $\forall F_0$ 或者 $\exists F_0$ 就有意义了. 最大的缺点是给不出准确原完整的 quantifier-free form. 也就是我们无法保证 $F_0 \Rightarrow C'$.

  前面我们把算法基本核心描述了一下, 我们下面一步一步来看:

  - 当 $i = n$, 如果 $F_n \wedge C$ is unsat, 那么 $F_n \wedge C \Rightarrow false$ 且 $false \Rightarrow F_n$, 即返回值 $(false, false)$. 如果 $F_n \wedge C$ is sat, $C' \Rightarrow F_n$ 是显然的, 还对 C' 做了一次泛化 (generalization) 找到弱一点 C'' 依然符合条件. 因为我们希望最终得到的那个强的 formula 能更弱一点, 尽量贴近 F_n.
  - 当 $i < n$ 时, 看下面这个 while 循环. 当 $b' = false$ 时, 由 $C \wedge M_i$ is unsat 可以马上得到 $C \wedge F_i$ is unsat, 因为 $F_i \Rightarrow M_i$ (在后面唯一修改 M_i 的地方, 我们会证明这个关系在 M_i 被修改之后依然保持). 因此 Q-Test(i, C) 返回 $(false, false)$, 性质 1 是显然保持的.

    当 $b' = true$ 时, 此时由 $C \wedge M_i$ is sat 还不能确定 $C \wedge F_i$ is sat. 同时是否有 $C' \Rightarrow F_i$ 也不能确定, 想确定它可以按照下面思路

    $$\begin{aligned} & C' \Rightarrow F_i \text{ is valid} \\ \equiv\ & \neg(C' \Rightarrow F_i) \text{ is unsat} \\ \equiv\ & C' \wedge \neg F_i \text{ is unsat} \\ \equiv\ & C' \wedge \exists_{B_i} F_{i+1} \text{ is unsat (by } F_i = \forall_{B_i} \neg F_{i+1}) \end{aligned}$$

    这里有 $FV(C') \subseteq FV(\exists_{B_i} F_{i+1})$, 那么这里最后只需要确定 $C' \wedge F_{i+1}$ 是否是 unsat 的, 直接调用 Q-Test$(i+1, C')$ 即可. 如果其返回值 $b'' = false$, 直接返回经过前面的分析是满足性质 1 的. 这里也还做一下 generalization, 也是希望最终得到的那个强的 formula 能更弱一点, 尽量贴近 F_i.

    如果其返回值 $b'' = true$, 那么我们不能确定 $C' \Rightarrow F_i$. 但是我们得到了一个 C'' 使得 $C'' \Rightarrow F_{i+1}$ and $C' \wedge C''$ is satisfiable. 就是这个 C'' 使得 $C' \wedge \exists_{B_i} F_{i+1}$ is sat, 如果我们未来想得到一个新的 C' 满足 $C' \Rightarrow F_i$, 就要尽量把 C'' 排除出去. 因此把 $\neg\pi(\exists_{B_i} C'')$ 加到 M_i 中. 这里我们需要确认一下 $F_i \Rightarrow M_i$ 是否依然保持. 由于 $C'' \Rightarrow F_{i+1}$, 所以有 $\exists_{B_i} C'' \Rightarrow \exists_{B_i} F_{i+1}$, 它的逆否命题为 $\forall_{B_i} \neg F_{i+1} \rightarrow \neg\exists_{B_i} C''$, 其左边是等价于 F_i, 右边等价于 $\neg\pi(\exists_{B_i} C'')$.

Algorithm 2 Q-TEST(i, C): satisfiability testing for $F_i \wedge C$

Require: (i, C) such that $0 \leq i \leq n$, $\mathrm{FV}(C) \subseteq \mathrm{FV}(F_i)$

if $i = n$ **then**
  $(b', C') :=$ SMT-TEST(C, F_n)
  if b' is false **then**
    return (false, false)
  else
    return (true, GENERALIZE$(C', K \mapsto \neg \mathit{first}(\text{SMT-TEST}(K, F_n))))$
  end if
else
  while true do
    $(b', C') :=$ SMT-TEST(C, M_i)
    if b' is false **then**
      return (false, false) {Since $F_i \Rrightarrow M_i$, then $C \wedge F_i$ is unsatisfiable too}
    else
      $(b'', C'') :=$ Q-TEST$(i + 1, C')$
      if b'' is false **then**
        $\{C' \wedge F_{i+1}$ is unsatisfiable$\}$
        return (true, GENERALIZE$(C', K \mapsto \neg \mathit{first}(\text{Q-TEST}(i+1, K))))$
      else
        $\{C''$ is such that $\mathrm{FV}(C'') \subseteq \mathrm{FV}(F_{i+1})$, $C'' \Rrightarrow F_{i+1}$ and $C' \wedge C''$ satisfiable. Thus $\exists_{B_i} C'' \Rrightarrow \exists_{B_i} F_{i+1}$, whence $F_i = \forall_{B_i} \neg F_{i+1} \Rrightarrow \neg \exists_{B_i} C'' \equiv \neg \pi_i(C'')\}$
        $M_i := M_i \wedge \neg \pi_i(C'')$
      end if
    end if
  end while
end if

Ensure: The return value is a pair (b, C'). b is a Boolean stating whether $F_i \wedge C$ is satisfiable. If b is true, then C' is a conjunction of literals such that $\mathrm{FV}(C') \subseteq \mathrm{FV}(F_i)$, $C' \Rrightarrow F_i$, and $C \wedge C'$ is satisfiable.

$x \mapsto v$ denotes the function mapping x to v; *first* denotes the function mapping a couple to its first element.

图 1: Q-Test

Misc

Definition 1.5. A set of logical connectives is called functionally complete if every boolean expression is equivalent to one involving only these connectives.

Definition 1.6. 简而言之，如果任意的 logical formula 都可以只用给定的一些 logical connectives 来构造一个等价的新的 logical formula, 我们就说这些这些 logical connectives 组成的 system 是 functionally complete.

Example 1.7. 列举一些在 classic logic 下 functionally complete 的 system:

- $\{\downarrow\}$, $\{\uparrow\}$, 其中 $\downarrow$ 表示 $A \downarrow B = \neg(A \vee B)$, $\uparrow$ 表示 $A \uparrow B = \neg(A \wedge B)$.
- $\{\vee, \neg\}$, $\{\wedge, \neg\}$, etc.
- $\{\wedge, \vee, \rightarrow\}$, etc.

Language

Standard Semantics

Annotation 2.1. Operational semantics 关注给定一个 initial state 会得到怎样的 final state, 每一个 CFG's node 对应一个 transfer function $f : \mathcal{S} \to \mathcal{S}$

$$\lambda s.\, (\mu(s), next).$$

其中 f 表示对 state s 的 update, next 表示下一个将要进入的 node. 那么 operational semantics 的一个解释过程可以用递归的形式定义:

$$interp = \lambda s.\, \lambda n.\, \textbf{if } isExitNode(n) \textbf{ then } s \textbf{ else let } (s', n') = f_n(s) \textbf{ in } interp\ s'\ n'$$

如果我们希望得到一个准确的定义，即不希望等式两边都包含 $interp$，那么我可以利用一个 trick

$$semantics = fix(\lambda F.\, \lambda s.\, \lambda n.\, \textbf{if } isExitNode(n) \textbf{ then } s \textbf{ else let } (s', n') = f_n(s) \textbf{ in } F\ s'\ n')$$

你仔细观察一下这个 fixpoint 就是 $interp$ 本身.

Collecting Semantics

Annotation 2.2. Collecting semantics 和 operational semantics 不同的是，每一个 CFG's node 可能对应了一个 set of states, 而不是单一的 state 了. 因为 collecting semantics 从一开始就把所有可能的 initial states 作为了一个 set 来考虑. 因此此时的 transfer function 为 $f : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$:

$$f_{n \to m} = \lambda S.\, \{\, s' \mid s \in S \text{ and } f_n(s) = (s', m) \,\}$$

Language equation and Arden's Rule

Theorem 2.3. The set $A^* \cdot B$ is the smallest language that is a solution for X in the linear equation

$$X = A \cdot X + B$$

where X, A, B are sets of string and $+$ stands for union of languages. Moreover, If the set A does not contain the empty word, then the solution is unique.

Annotation 2.4. Arden's rule can be used to help convert some finite automatons to regular expressions.

Equivalence of Program

Graph Ismorphism

Accepted Language Equivalentce

Annotation 3.1. [4] Chapter 1.

Bisimulation and Observation Equivalence

Definition 3.2. A labelled transition system (LTS) is a tuple $(S, \Lambda, \rightarrow)$ where S is set of states, Λ is set of labels, and $\rightarrow$ is relation of labelled transitions (i.e., a subset of $S \times \Lambda \times S$). A $(p, \alpha, q) \in \rightarrow$ is written as $p \xrightarrow{\alpha} q$.

Annotation 3.3. TODO: categorical semantics: F-coalgebra

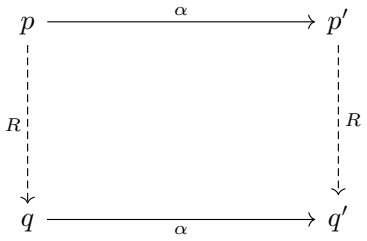
Definition 3.4. [1]Let $T = (S, \Lambda, \rightarrow)$ be a labelled transition system. The set of traces $Tr(s)$, for $s \in S$ is the minimal set satisfying

- $\varepsilon \in Tr(s)$.
- $\alpha\ \sigma \in Tr(s)$ if $\{\, s' \in S \mid s \xrightarrow{\alpha} s' \text{ and } \sigma \in Tr(s') \,\}$.

Definition 3.5. Two states p, q are trace equivalent iff $Tr(p) = Tr(q)$.

Definition 3.6. (Simultation) Given two labelled transition system $(S_1, \Lambda, \rightarrow_1)$ and $(S_2, \Lambda, \rightarrow_2)$, relation $R \subseteq S_1 \times S_2$ is a simulation iff, for all $(p, q) \in R$ and $\alpha \in \lambda$ satisfies

$$\text{for any } p \xrightarrow{\alpha}_1 p', \text{ then there exists } q' \text{ such that } q \xrightarrow{\alpha}_2 q' \text{ and } (p', q') \in R$$

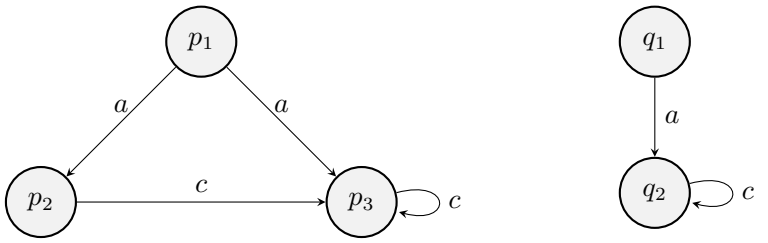


Definition 3.7. We say q simulates p if there exists a simulation R includes (p, q) (i.e., $(p, q) \in R$), written $p < q$.

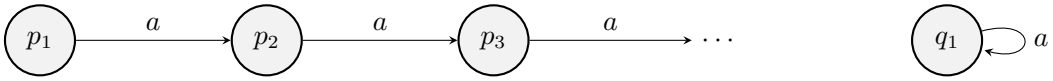
Definition 3.8. (Bisimultation) Given two labelled transition system $(S_1, \Lambda, \to_1)$ and $(S_2, \Lambda, \to_2)$, relation $R \subseteq S_1 \times S_2$ is a bisimulation iff both R and its converse $\overline{R}$ are simulations, for all $(p, q) \in R$ and $\alpha \in \Lambda$ satisfies

$$\text{for any } p \xrightarrow{\alpha}_1 p', \text{ then there exists } q' \text{ such that } q \xrightarrow{\alpha}_2 q' \text{ and } (p', q') \in R$$
$$\text{for any } q \xrightarrow{\alpha}_2 q', \text{ then there exists } p' \text{ such that } p \xrightarrow{\alpha}_1 p' \text{ and } (p', q') \in R$$

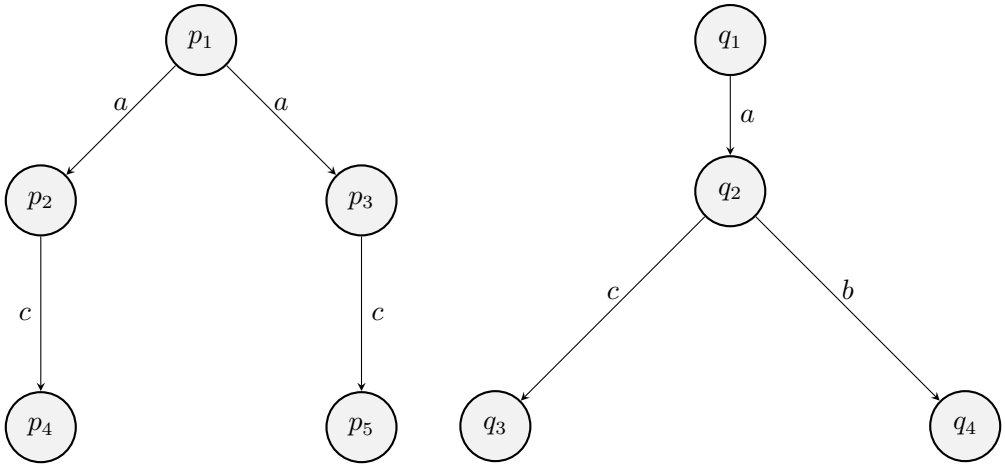
Example 3.9. 一些 bisimulation 的例子



关于上面两个 transition system 的 bisimultaion 为 $R = \{(p_1, q_1), (p_2, q_2), (p_3, q_2)\}$. 还有一个比较有点特别的例子



如果关于上图这样 bisimulation R 存在，那么 $(p_i, q_1) \in R$ for every i. 再看一个不是 bisimulation 的例子



这里不满足 $(p_3, q_2) \notin R$.

Definition 3.10. (Bisimilarity) Given two states p and q in S, p is bisimilar to q, written $p \sim q$, if and only if there is a bisimulation R such that $(p, q) \in R$.

Definition 3.11. The bisimilarity relation $\sim$ is the union of all bisimulations.

Lemma 3.12. The bisimulation has some properties:

- The identity relation id is a bisimulation(with two same LTS).
- The empty relation $\perp$ is a bisimulation.
- (closed under union) The $\bigcup_{i\in I} R_i$ of a family of bisimulations $(R_i)_{i\in I}$ is a bisimulation.

Lemma 3.13. [2] The bisimilarity relation $\sim$ is equivalence relation (i.e., reflexivity, symmetry, transitivity).

证明. 其中 reflexivity, symmetry 是比较显然的. Transitivity 稍微麻烦一点, 我们用 relation composition 定义新的 relation$R_3 = R_1; R_2$, 此时有 $(p, q) \in R_3$, 因此只要证明 R_3 is bisimulation 足够了. 取任意一个 $(p_1, q_1) \in R_3$, 那么按照 R_3 的定义, 存在 $(p_1, r_1) \in R_1$ 和 (r_1, q_1). 由 $p_1 \sim r_1$ 那么对于任意的 $p_1 \xrightarrow{\alpha} p_1'$, 存在 $r_1 \xrightarrow{\alpha} r_1'$ 满足 $(p_1', r_1') \in R_1$. 再由 $r_1 \sim q_1$, 存在 $q_1 \xrightarrow{\alpha} q_1'$ 满足 $(r_1', q_1') \in R_2$. 于是按照 R_3 的定义也有 $(p_1', q_1') \in R_3$. 再由 R_2 is bisimulation, 从 $(r_1, q_1) \in R_2$ 按照上述的思路往回证明即可, 最终 R_3 is bisimulation. □

Definition 3.14. [3] An LTS is called deterministic if for every state p and action α, there is at most one state q such that $p \xrightarrow{\alpha} q$.

Lemma 3.15. In a deterministic LTS, two states are bisimilar if and only if they are trace equivalent,

$$s_1 \sim s_2 \iff Tr(s_1) = Tr(s_2)$$

证明. 先证 $\Rightarrow$, 设满足 $s_1 \sim s_2((s_1, s_2) \in R$ and R is bisimultaion), 设 $\sigma_{s_1} \in Tr(s_1)$, 其中 σ_{s_1} 为 sequence $(\alpha_i)_{i\in I}$ where I is a indexed famliy. 由于 $s_1 \sim s_2$, 那么对于 $s_1 \xrightarrow{\alpha_1} s_1'$, 存在 $s_2 \xrightarrow{\alpha_1} s_2'$, 于是 $(s_1', s_2') \in R$, 根据 σ 长度做 induction 可以证明 $\sigma_{s_1} \in Tr(q)$. 再反过来证明 $\sigma_{s_2} \in Tr(s_2)$ 也同样有 $\sigma_{s_2} \in Tr(s_1)$. 最终 $Tr(s_1) = Tr(s_2)$.

对于 $\Leftarrow$, 我们可以用 $Tr(s_1) = Tr(s_2)$ 构造一个 bisimulation, 定义 relation R 为

$$Tr(s_1) = Tr(s_2) \iff (s_1, s_2) \in R.$$

只要能证明 R bisimulation 即可. 首先我们来说明在 deterministic 限制下一个比较好性质: 若 $Tr(s_1) = Tr(s_2)$ 且当 $s_1 \xrightarrow{\alpha} s_1', s_2 \xrightarrow{\alpha} s_2'$, 那么 $Tr(s_1') = Tr(s_2')$. 这样对于任意地 $(s_1, s_2) \in R$, 它们 accept 相同 action 对应的 transition $(s_1', s_2') \in R$. 因此 $s_1 \sim s_2$. □

Definition 3.16. (Weak Bisimultation) Given two labelled transition system $(S_1, \Lambda, \rightarrow_1)$ and $(S_2, \Lambda, \rightarrow_2)$, relation $R \subseteq S_1 \times S_2$ is a bisimulation iff both R and its converse $\overline{R}$ are simulations, for all $(p, q) \in R$ and $\alpha \in \Lambda \cup \{\tau\}$ satisfies

$$\text{for any } p \xrightarrow{\alpha}_1 p', \text{ then there exists } q' \text{ such that } q \xrightarrow{\tau* \ \alpha \ \tau*}{}^{*}_{2} q' \text{ and } (p', q') \in R$$

$$\text{for any } q \xrightarrow{\alpha}_2 q', \text{ then there exists } p' \text{ such that } p \xrightarrow{\tau* \ \alpha \ \tau*}{}^{*}_{1} p' \text{ and } (p', q') \in R$$

where $\rightarrow^*$ is multi-transition.

Annotation 3.17. 对于 LTS 的一些想法:

- 如果你想用 transition system 来做 reasoning 可以考虑把它和 Kripke frame 联系起来，同时要构造一些 modality 来设计方便做 reasoning 的 calculus.
- (*bisimulation proof method*) 对于两个特别的 states 来说，我们应该如何找到这样 bisimulation 来满足 $(p, q) \in R$?
- 对于两个特别的 LTS 来说，我们怎样以 bisimulation 思考它们是否 equivalent? bisimulation 的最初定义应该叫做 strong bisimulation, 它建立的是一种 strong equivalence, 而 weak bisimulation 建立是一种 observation equivalence.

Annotation 3.18. TODO: CCS(calculus of communicating systems)[4] and mCRL2 [3].

Symbolic Execution

Predicate transformer

Definition 4.1. A predicate transformer p is a function

$$p : \text{FOL} \times \text{stmts} \to \text{FOL}$$

where FOL are first-order logic formulas and stmts are program statements.

Definition 4.2. Given a FOL Q and program statement c, the weakest precondition is a FOL $\text{wp}(Q, c)$ satisfies the following conditions

- $\{\text{wp}(Q, c)\}c\{Q\}$.
- for any FOL P such that $\{P\}c\{Q\}$, $P \Rightarrow \text{wp}(Q, c)$.

Example 4.3. For assumption

$$\text{wp}(Q, \text{assume } e\} = e \to Q,$$

想想 deduction theorem 或者 modus ponens, 没有比这个更弱的.

Definition 4.4. Given a FOL Q and program statement c, the strongest postcondition is a FOL $\text{sp}(P, c)$ satisfies the following conditions

- $\{P\}c\{\text{sp}(Q, c)\}$.
- for any FOL Q such that $\{P\}c\{Q\}$, $\{\text{sp}(Q, c)\} \Rightarrow Q$.

Example 4.5. For assignment, there is standard nation. One possible is

$$\text{sp}(P[e/x], x \coloneqq e) = P.$$

Another one is

$$\text{sp}(P, x \coloneqq e) \equiv \exists v'.v = e[v'/v] \wedge P[v'/v].$$

For example,

$$\text{sp}(x \geq i, x \coloneqq x + k) \equiv (\exists x'.x = x' + k \wedge x' \geq i) \equiv (x - k \geq i).$$

第二种方式显的更加的自然, 对做符号执行是比较友好的.

Some Reasoning

Example 4.6. Symbolic reachability analysis 这是来自 [5] 的一个小例子, 我们尝试用 dynamic symbolic execution 做一些 reasoning.

```
#define VALVE_KO(status) status == -1
#define TOLERANCE 2
extern int size;
extern int valvesStatus[];

int getStatusOfValve(int i){
    if(i < 0 || i >= size){
        printf ("ERROR");
        exit(EXIT_FAILURE);
    }
    int status = valvesStatus[i];
    return status;
}

int checkValves(int wait1, int wait2) {
    int count, i;
    while(wait1 > 0) wait1--;
    count = 0, i = 0;
    while(i < size){
        int status = getStatusOfValve(i);

        if(VALVE_KO(status)) {
            count++;
        }
        i++;
    }

    if(count > TOLERANCE)
        printf ("ALARM");
    }
    while(wait2 > 0) wait2--;
        return count;
```

[5] 提到了一个 symbolic reachability analysis, 它和我们常见的 symbolic execution 是不一样的, 它可以看做给定

一个 postcondition 沿着 control flow 往后推. 这种方法在解决一些 branch condition indirectly related to input, 可能会有一些帮助. 例如 L29 所在 branch condition, 它并不是直接依赖 input. 如果我们将这个 while 展开，那么每次在某条路径上做 symoblic execution 到 L28 时, count 都是一个 concrete value, 如果想尝试在 L28 这里开分支是做不到的.

例如我们想进入 L29 所在的 branch, 那么 one-step induction 如下:

```
// P_28 = count > 2
{L28: count > 2}
// Q_28 = true
```

可以看到 precondition 是 weakest 的, 后面推导依然保持这个性质. 继续往后推导我们需要尝试得 resolve 掉 L19-L26 的 while, 这里可能就有 infinitely many paths, 例如执行 $0, 1, 2, \cdots$ 次这个 loop. 顺着这个思路来选择路径往后做 symbolic execution, 路径直到 function entry 为结束.

```
//Path_1: L15-> L16 -> L17 -> L18 -> L19 -> L28

// P_18 = count > 2 ∧ i ≥ size ∧ 0 > 2 ∧ 0 > size ≡ false
{L18: count = 0, i = 0; }
// Q_18 = count > 2 ∧ i ≥ size
// P_19 = count > 2 ∧ i ≥ size
{L19: i >= size}
// Q_19 = count > 2
// P_28 = count > 2
{L28: count > 2}
// Q_28 = true
```

在 P_{18} 这里得到了一个 contradiction, 这就意味着上面选择的 path 是 infeasible 的，那么到这里我们就不能往后再继续推理了. 现在我们给用 $Ln_a, Ln_b, \cdots$ 的形式来表示对同一 statement Ln 的多次执行.

```
//Path_2: ··· -> L19_b -> L20_b -> L22_b -> L23_b -> L25_b -> L19_a -> L28

...
// P_19b = count > 1 ∧ i = size − 1 ∧ i ≥ 0 ∧ valvesStatus[i] = −1 ∧ i < size
{L19b: i < size}
// Q_20b = count > 1 ∧ i = size − 1 ∧ i ≥ 0 ∧ valvesStatus[i] = −1
// P_20b = (count > 1 ∧ i ≥ size − 1 ∧ i ≥ 0 ∧ i < size ∧ valvesStatus[i] = −1) ≡
// (count > 1 ∧ i = size − 1 ∧ i ≥ 0 ∧ valvesStatus[i] = −1)
{L20b: int status = getStatusOfValve(i);}
// Q_20b = count > 1 ∧ i ≥ size − 1 ∧ status = −1
```

```
// P_{22b} = count > 1 ∧ i ≥ size − 1 ∧ status = −1
{L22b: status == -1}
// Q_{22b} = count > 1 ∧ i ≥ size − 1
// P_{23b} = (count + 1 > 2 ∧ i ≥ size − 1) ≡ (count > 1 ∧ i ≥ size − 1)
{L23b: count++}
// Q_{23b} = count > 2 ∧ i ≥ size − 1
// P_{25b} = (count > 2 ∧ i + 1 ≥ size) ≡ (count > 2 ∧ i ≥ size − 1)
{L25b: i++; }
// Q_{25b} = count > 2 ∧ i ≥ size
// P_{19a} = count > 2 ∧ i ≥ size
{L19a: i >= size}
// Q_{19a} = count > 2
// P_{28} = count > 2
{L28: count > 2}
// Q_{28} = true
```

上面就是执行了最后一次循环并且在这次循环中进入了 L23 所在的 branch，主要需要注意一下 P_{20b} 这里设计到了 inter-analysis.

Craig Interpolation

Definition 4.7. Let $P \to Q$ be a valid propositional formula. A Craig Interpolation for P and Q is a formula C that satisfies the following conditions:

- $P \to C$ and $C \to Q$ are valid.
- All the variables in C also appear in both P and Q.

Theorem 4.8. (Craig Interpolation Theorem) Let $P \to Q$ be a propositional formula, where P and Q share at least one atomic proposition. Then there exists a formula C containing only variable symbols in both P and Q such that $P \to C$ and $C \to Q$.

证明. 我们用 $\text{atoms}(P)$ 和 $\text{atoms}(Q)$ 分别表示 P 和 Q 中的 variable symbols(atomic proposition). 这里对 $|\text{atoms}(P) - \text{atoms}(Q)|$ 做 induction, 其中 $\text{atoms}(P) - \text{atoms}(Q)$ 表示出现在 P 中但不在 Q 中的 variable symbols.

Base case: 当 $|\text{atoms}(P) - \text{atoms}(Q)| = 0$, 我们可以让 $C = P$ 作为一个 interpolation, 显然 $P \to P$ is valid and $P \to Q$.

Inductive hypothesis: 假设 $|\text{atoms}(P) - \text{atoms}(Q)| = n$ 时原命题成立.

Inductive case: 当 $|\text{atoms}(P) - \text{atoms}(Q)| = n + 1$ 时, 我们取 $\alpha \in |\text{atoms}(P) - \text{atoms}(Q)|$. 我们定义 $P_{\alpha \mapsto \top}$ 表示将 P 中所有 α 替换成 $\top$ 得到的 formula, 类似地我们用 $P_{\alpha \mapsto \bot}$ 表示将 P 中所有 α 替换成 $\bot$ 得到的 formula. 显然我们有 $P \equiv P_{\alpha \mapsto \top} \vee P_{\alpha \mapsto \bot}$. 根据 inductive hypothesis 我们找到一个关于 $P_{\alpha \mapsto \top} \vee P_{\alpha \mapsto \bot} \to Q$ 的 interpolation C. 显然 C 也是 $P \to Q$ 的一个 interpolation. □

Annotation 4.9. 值得注意上面的 proof 仅仅在 proposition logic 下证明了 Craig interpolation 了. 这也是为什么我们可以将上面的 α 分别用 $\top$ 和 $\bot$ 替换, 因为 α 表示的实际是 atomic proposition, 对于 atomic proposition 而言它只有 $\top$ 和 $\bot$.

那么比较自然的问题就是 first-order logic 上怎么证明? 先回答两个 first-order formula φ 和 ψ 需要 shared 什么? 若

$$\forall x.F(x) \to \exists y.F(y).$$

[6] 里面提到了 non-logical symbols, 那么 first-order logic 中的 non-logical symbols 到底是什么呢? 在 wiki 中它被定义为 predicates, functions, and constants. 我们来分别思考一下这些 symbols 在上面的 inductive step 中应该怎么被处理?

1. 如果存在 constant $c_1 \in \text{atoms}(P) - \text{atoms}(Q)$, 此时我们可以用新一个 fresh variabe v_1 来替换这个 c_1, 得到一个新的 formula $\exists v_1.P_{c_1 \mapsto v_1}$. 显然 $P \to \exists v_1.P_{c_1 \to v_1}$, 这里就可以继续用 inductive hypothesis 了.
2. 如果存在 function $f_1 \in \text{atoms}(P) - \text{atoms}(Q)$, ???

3. 如果存在 predicate $p_1 \in \text{atoms}(P) - \text{atoms}(Q)$, ???

看来并不 trivial. 这里需要再想想.

Theorem 4.10. (Satisfiability form) Let A and B be proposition formula. A Craig Interpolation for A and B is a formula that satisfies the follow:

- $A \wedge B$ is unsatifiable.
- $A \to C$.
- $C \wedge B$ is unsatifiable.

Annotation 4.11. 如何理解上面的 satisfiability form 呢? 首先 $A \wedge B$ is unsat, 那么则有 $(\neg A \vee \neg B) \equiv (A \to \neg B)$ is valid. 再 $C \vee B$ is unsat, 那么则有 $(\neg C \vee \neg B) \equiv (C \to \neg B)$ is valid. 显然 C 是关于 A 和 $\neg B$ 的一个 interpolation.

在使用中我们会通常中 A 和 B 分别表示两个 set of clauses, 也就是对 A 和 B 进行 normalization 先.

Definition 4.12. A proof of unsatisifability Π for a set of clause C is a root-tree (V_Π, E_Π), where V_Π is a set of clauses, such that for every vertex $c \in V_\Pi$:

- if c is a empty clause, then c is the unique root.
- if c is resolvent of c_1 and c_2, then edge (c, c_1) and (c, c_2) are both in E_Π.
- c is leaf that is caluse in C otherwise.

Annotation 4.13. 这个 proof 实际就是做 resolution 得到 empty clause 一个过程, 是很自然的一个从下到上的一棵树, 当然也有定义把 empty clasue 当做 unique leaf 的形式. 无论哪种情况, 只要能理解 resolution 过程就行. 这里简单把 resolution 再写一遍.

$$\frac{p \cup C_1 \quad \neg p \cup C_2}{C_1 \cup C_2}$$

其中 literal p 对应的 variable 称为 pivot variable.

Definition 4.14. Given a set of clauses C, we say a variable is global if it appears in all clauses in C, and local to one clause c in C if it appear only in c. Given any clause $c_i \in C$, we denote by $g(c_i)$ the disjunction of the global literals in c_i and by $l(c_i)$ the disjunction of literals local to c_i

Annotation 4.15. 注意上面 variable 和 literal 的描述.

Theorem 4.16. (linear-time construction) Let (A, B) be a pair of caluse sets and let Π be a proof of unsatisfiability of $A \cup B$. For all vertices $c \in V_\Pi$, let $p(c)$ be a boolean formula, such that

- if c is leaf, then
  - if $c \in A$ then $p(c) = g(c)$,
  - else $p(c)$ is the $\top$.
- else, let c is resolvent of c_1 and c_2 and let v be pivot variable of this resolution.
  - if v is local to A, then $p(c) = p(c_1) \vee p(c_2)$,
  - else $p(c) = p(c_1) \wedge p(c_2)$.

Then $p(\bot)$ is an interpolant for (A, B) where $\bot$ is the root of Π.

Annotation 4.17. 这个证明感觉不是那么显然, 花了 2-3 天想了一下. 先从直觉出发, 我们先设 A 和 B 是没有 common clauses, 再假设 proof Π 里面属于 A 的 leaves 为 $\{\alpha_1, \alpha_2, \cdots, \alpha_m\}$, 属于 B 的 leaves 为 $\{\beta_1, \beta_2, \cdots, \beta_n\}$. 我们可以重排整个 proof Π:

- 先对 $\{\alpha_1, \alpha_2, \cdots, \alpha_m\}$ 做 resolution, 直到无法继续. 得到 $\{\alpha_1', \alpha_2', \cdots, \alpha_i'\}$
- 再对 $\{\alpha_1', \alpha_2', \cdots, \alpha_i'\}$ 和 $\{\beta_1, \beta_2, \cdots, \beta_n\}$ 一起做 resolution.

上述想法实际对应了这样一个操作，如果 Π 中存在这样一个片段:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{v_2 \cup c_4} \quad \overset{\mathcal{E}}{\neg v_2 \cup c_5}}{v_1 \cup c_2}\ v_1 \notin l(A) \quad \overset{\mathcal{F}}{\neg v_1 \cup c_3}}{c_1}\ v_1 \in l(A)$$

我们就把下面这个 resolution 往上移动，变成下面这样

$$\dfrac{\dfrac{\overset{\mathcal{D}}{v_1 \cup (v_2 \cup c_4 \backslash v_1)} \quad \overset{\mathcal{F}}{\neg v_1 \cup c_3}}{v_2 \cup (c_4 \backslash v_1 \cup c_3)}\ v_1 \in l(A) \quad \overset{\mathcal{E}}{\neg v_2 \cup c_5}}{c_1}\ v_2 \notin l(A)$$

$$\dfrac{\overset{\mathcal{D}}{v_2 \cup c_4} \quad \dfrac{\overset{\mathcal{E}}{v_1 \cup (\neg v_2 \cup c_5 \backslash v_1)} \quad \overset{\mathcal{F}}{\neg v_1 \cup c_3}}{\neg v_2 \cup (c_5 \backslash v_1 \cup c_3)}\ v_1 \in l(A)}{c_1}\ v_2 \notin l(A)$$

上面两个略微不同的形式依赖于 c_1 到底是在 c_4 里面还是在 c_5 里面. 显然这个往上移动操作是可以做到的，也就是说现在整棵树依然是 (A, B) 一个 proof, 我们设其为 Π', 我们接着来研究一下两个 proof 最后得到的 $p(\bot)$ 有什么关系，我们设 Π' 对应 $p'(\bot)$. 因为上述只是一个局部变换，没有新增任何结点，只有两个结点对应的 boolean formula 发生了变化，对于整个 proof 而言只是以 c_1 为根结点的子树发生了变化，所以我们只需要看一下 $p(c_1)$ 和 $p'(c_1)$ 到底是什么关系. 我们先设 $p(v_2 \cup c_4) = b_1, p(\neg v_2 \cup c_5) = b_2, p(\neg v_1 \cup c_3) = b_3$. 那么显然 $p(c_1) = (b_1 \wedge b_2) \vee b_3$. 对于 $p'(c_1)$ 我们有两个不同的结果:

- $v_1 \in c_4$: $p_1'(c_1) = b_1 \wedge (b_3 \vee b_2) \equiv (b_1 \wedge b_3) \vee (b_1 \wedge b_2)$
- $v_2 \in c_5$: $p_2'(c_2) = (b_1 \vee b_3) \wedge b_2 = (b_1 \wedge b_2) \vee (b_2 \wedge b_3)$

显然有 $p_1'(c_1) \to p(c_1)$ 和 $p_2'(c_1) \to p(c_1)$. 这可以说明什么呢? $p'(\bot) \to p(\bot)$, 很可惜这种方式只能得到上述定理的一个弱的形式. 换句话就是我们只能得到一个比较强的 interpolation, 我们一般更在乎是尽量弱的 interpolation, 这样它的结构或许更加的简洁.

换个思路我们重新开始思考, 既然上面那个方向转换不行, 我们换个方向. 我们这样重排 proof $\prod$:

- 先对 $\{\beta_1, \beta_2, \cdots, \beta_n\}$ 做 resolution, 直到无法继续. 得到 $\{\beta_1', \beta_2', \cdots, \beta_j'\}$.
- 再对 $\{\beta_1', \beta_2', \cdots, \beta_j'\}$ 和 $\{\alpha_1, \alpha_2, \cdots, \alpha_m\}$ 一起做 resolution.

但是这里有一个小问题就是 pivot variable 为 $v_1 \notin l(A)$ 的 resolutions 并不完全是 B 里面的 resolution, 当 v_1 是 global variables 的时候它实际是属于 A 与 B 之间的 resolution. 这里其实就暗示了我们需要对上面的第二步再细化一步, 这个问题留到后面. 我们先看看反方向移动有什么性质. 如果 Π 中存在这样一个片段:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{v_2 \cup c_4} \quad \overset{\mathcal{E}}{\neg v_2 \cup c_5}}{v_1 \cup c_2}\ v_1 \in l(A) \quad \overset{\mathcal{F}}{\neg v_1 \cup c_3}}{c_1}\ v_1 \notin l(A)$$

我们把下面 resolution 还是往上移动, 变成下面这样:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{v_1 \cup (v_2 \cup c_4 \backslash v_1)} \quad \overset{\mathcal{F}}{\neg v_1 \cup c_3}}{v_2 \cup (c_4 \backslash v_1 \cup c_3)}\ v_1 \notin l(A) \quad \overset{\mathcal{E}}{\neg v_2 \cup c_5}}{c_1}\ v_2 \in l(A)$$

$$\dfrac{\overset{\mathcal{D}}{v_2 \cup c_4} \quad \dfrac{\overset{\mathcal{E}}{v_1 \cup (\neg v_2 \cup c_5 \backslash v_1)} \quad \overset{\mathcal{F}}{\neg v_1 \cup c_3}}{\neg v_2 \cup (c_5 \backslash v_1 \cup c_3)}\ v_1 \notin l(A)}{c_1}\ v_2 \in l(A)$$

实际就是把前面的 $\notin$ 和 $\in$ 互换. 此时有 $p(c_1) = (b_1 \vee b_2) \wedge b_3$. 对于 $p'(c_1)$ 分别有

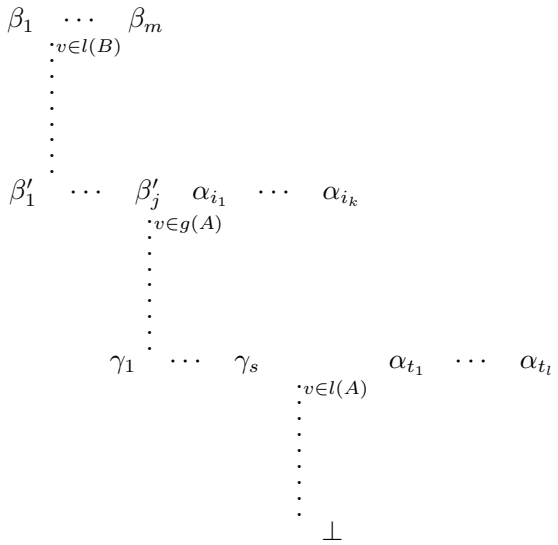
- $c_1 \in c_4$: $p_1'(c_1) = (b_1 \wedge b_3) \vee b_2$.
- $c_1 \in c_5$: $p_2'(c_2) = b_1 \vee (b_3 \vee b_2)$.

显然此时我们有了希望看到的结论 $p(c_1) \to p'(c_1)$, 等价于我们已经有了 $p(\bot) \to p'(\bot)$ 这已经算是成功的基础了). 当你不断对 proof Π 做这个操作, 直到无法继续我们设这个新的 proof 为 Π'. 现在我们来看一下 Π' 结构是怎样的. 现在 Π 最上面的 resolution 都是 pivot variable 都满足 $v \notin l(A)$, 前面也提到了这样的 resolution 包

含了 pivot variable $v \in l(B)$ 和 $v \in g(A)$ 两种情况. 我们再来排一次，把 pivot variable $v \in l(B)$ 放在最前面. 也就是如出现下述 proof 片段:

$$\dfrac{\dfrac{\overset{\mathcal{D}}{v_2 \cup c_4} \quad \overset{\mathcal{E}}{\neg v_2 \cup c_5}}{v_1 \cup c_2}\ v_1 \in g(A) \quad \overset{\mathcal{F}}{\neg v_1 \cup c_3}}{c_1}\ v_1 \in l(B)$$

我们把下面 resolution 往上移动, 也会得到上述类似两种形式, 这里不在累述了. 最重要是此时 $p'(c_1) = p''(c_1)$, 因为这里都是 conjunction. 这就是说明这种移动并不会影响对应结点的 boolean formula, 也就是说依然有 $p(c_1) \to p''(c_1)$. 我们对 Π' 不断做这个操作，知道无法继续我们设这个新的 proof 为 Π''. 我们可以简单画一下 Π'' 的结构:



整个结构分为三个层次，是我们通过前两次移动策略得到的. 我们主要看一下中间整个层次. 首先可以确定是 $\{\beta'_1, \cdots, \beta'_j\}$ 里面所有 variables 都是在 $g(A)$ 中, 此时我们如果考虑把 $\alpha_{i_1}, \cdots, \alpha_{i_k}$ 分布替换成 $g(\alpha_{i_1}), \cdots, g(\alpha_{i_k})$. 此时第二个层次的结构是不会发生变化的，因为它里面做的 resolution 的 pvoit variables 也都在 $g(A)$. 并且你会发现 $\gamma_1, \cdots, \gamma_s$ 会被消成 $\bot$. 这样我们得到又得到了一个新的更短的 proof Π'''，我们也自然地得到了一个 interpolation $g(\alpha_{i_1}) \wedge \cdots \wedge g(\alpha_{i_k})$. 我们证明了一个比较弱的 interpolation，那么原来比它强的 interpolation 也自然是存在的，这样我们整个证明就结束了.

证明. Formalization for above sketch. □

Annotation 4.18. 上面的 theorem 最重要地就是告诉我们计算 interpolation 不需要第二次用到 SAT-solver, 并且整个计算过程是线性的.

Definition 4.19. (Craig interpolant over path) Given formula $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ is unsatisfiable. Let sequence of formulas $\hat{A}_0, \hat{A}_1, \cdots, \hat{A}_n$ is defined as follows.

- $\hat{A}_0 = true$ and $\hat{A}_n = false$.
- for all $1 \leq i \leq n$, $\hat{A}_{i-1} \wedge A_i \rightarrow \hat{A}_i$.
- for all $1 \leq i < n$, $\hat{A}_i \in (\mathcal{L}(A_1 \wedge \cdots \wedge A_i) \cup \mathcal{L}(A_{i+1} \wedge \cdots \wedge A_n))$ where $\mathcal{L}(\phi)$ denote well-formed formula over the vocabulary of ϕ.

Graph as Proof

Annotation 4.20. 当我们想证明 program 中某个点 unreachable 的时候，我们需要说明可以抵达这个点的所有路径都是 infeasible 的. 那么我们应当以怎样的形式给出这个证明呢? 我们可以从 CFG 构造出另外一个 graph, 借助这个 graph 我们可以很直接说明这些路径确实不存在, 这里有两种手法:

- 直接在 graph 上显式地去掉这些路径.
- graph 上显式的把这些路径标出来，并且说明这些路径不存在.

无论哪种方法都需要扩展原 CFG. 我们可以形式化一点，设 CFG 为 $G_1 = (V_1, E_1)$ 和新的 graph 为 $G_2 = (V_2, E_2)$. 我们需要两个 basic map:

- vertex map: $f_v : V_2 \to V_1$.
- edge map: $f_e : E_2 \to E_1$.

这两个 map 在这里指明 G_2 上 vertex 和 edge 都在 G_1 中, 因为我们希望 G_2 上的路径都是在 G_1 中的. 值得注意的是存在 $f_v(v) = f(w), v \neq w \in V_2$. 这是因为希望 G_2 可以表示独立的路径，这使得我们需要把两条不同路径上的重复结点分开，这样的重复结点虽然对应 G_1 中同一个结点但是在 G_2 是 distinct 的.

无论哪种的方法，我们都是无法避免去选路径，然后沿着路径去分析的. 这里我们继续以符号化分析为起点，给定一条从 entry noode 到 target node 的路径 $\pi = (\epsilon, a_0, a_1, \cdots, a_n)$, 它对应 path condition 是 normal form 为 $P_\pi = A_0 \wedge A_1 \wedge A_n$. 我们再建立一个对应关系, 当 $i > 0$ 时, a_{i-1} 执行 a_i 都之前都满足需要 A_i. 其中 a_0 是真正的 entry node, 额外给出 ϵ 为了可以在开始的时候方便地给一个 initial condition(函数调用传参). 这里有两种情况:

- P_π is sat, 那么我们可以直接出给反例.
- P_π is unsat, 那么我们就可以排除这条路径.

后面我们主要研究如何不断的排除这条路径.

Example 4.21. (Navie analysis) 我们通常是在 CFG 类似的 program graph 上进行分析，显然我们分析的路径都是这个 CFG 上存在的路径. 当某条路径是 infeasible 时，我们可以想办法把它从 CFG 上抹去，这就是最初的直觉. 图2中的 φ 为图中路径的 prefix path condition, 消除这条路径并不能简单的把 c 这条边去掉, 我们需要 copy 结点然后加上 annotation, 这个 annotation 的含义是路径条件满足它的时候才能继续往前. 因此这里带 φ 的 copy 结点是没有以 c 为 label 的 outgoing edge. 这样在 annotation + copy 的模式下我们就去掉了 infeasible path.

还有一个问题是我们应该选择 copy 哪一个结点, 通常我们选择那个结点是和 frontier 相关的，frontier 是一种特殊的边. 它连接的我们已知可达的结点和还没有抵达的结点, 这里我们还不知道它是否可达. 当我们经过进

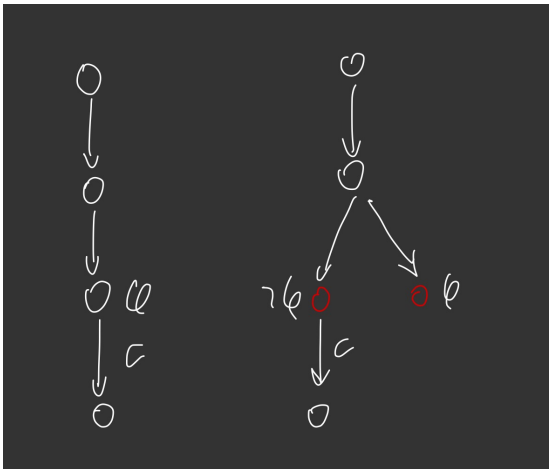


图 2: Refinement by node copying

一步计算之后发现它不可达, i.e., 一步扩展可达路径的路径. 此时这个不可达结点的前驱可达结点，就可以称为我们的 copy 对象. 可以看见这个 φ 是满足一定性质的，即 φ is sat and $\varphi \wedge \psi_c$ is unsat, 这是有道理的. 我们不能任意选择 infeasible path 中间的结点来 copy, 然后删掉一些边. 你必须要保证从删掉这些边产生的路径都是 infeasible 才行.

最后一个值得探讨的问题, 这个 ϕ 往往是比较复杂的, 把它作为 annotation 实际上不太方便的, 因为当你抵达这个点的时候, 你需要判断它是否满足 ϕ 再做进一步打算. 利用 Craig interpolation 我们就可以简化这个 ϕ, 得到一个 $\hat{\phi}$. 在满足 $\hat{\phi}$ 的情况下也是不可达的, 这里弱一点 $\hat{\phi}$ 显然要比 ϕ 好.

上面的这种方法存在冗余的问题. 举一个完整的分析例子, 图3对应一个带 if 的简单程序, 在 if 的出口有一个关于 error 的 transition，假设 error 这个点是 unreachable 的，其他的点都是 reachable 的. 其中绿色的路径表示我们每次分析的 CFG 上关于 error 的一条路径, 这里我们用了 7 步才判定图中 error 这个点是 unreachable 的. 我们分别来看一下过程:

1. 此时 π 取不经过 if 的路径. 这里 φ_1 is sat and $\varphi_1 \wedge \varphi_e$ is unsat. 那么 e 对应的这条边就是这里的 froniter, 这里我们 copy φ 所对应的结点. 我们设 copy 对象为结点 v, copy 的步骤如下:

   - copy v 得到 v', 包括 v 的所有前驱都需要也指向 v'; v' 只需要指向所有 v 的后继.
   - 去掉 v 上的 froniter.

   v 上标注 annotation φ_1, 而 v' 上标注 annotation $\neg\varphi_1$. 于是得到了 (2).

2. 此时 π 取 (2) 中经过 if 的路径. 这里 $\neg\varphi_1 \wedge \varphi_2 \wedge \varphi_e$ is unsat. 那么 e 对应的这条边就是这里的 frontier, 这里我们 copy $\neg\varphi_1 \wedge \varphi_2$ 对应的结点, 得到 $\varphi_1 \vee \neg\varphi_2$. 这里我写错成 $\neg\varphi_1 \wedge \neg\varphi_2$. 理论上这个条件也没什么问题, 所以这里貌似还要多几步才行, 最终得到应该会是一个 $\neg\varphi_1 \wedge (\varphi_1 \vee \neg\varphi_2)$.

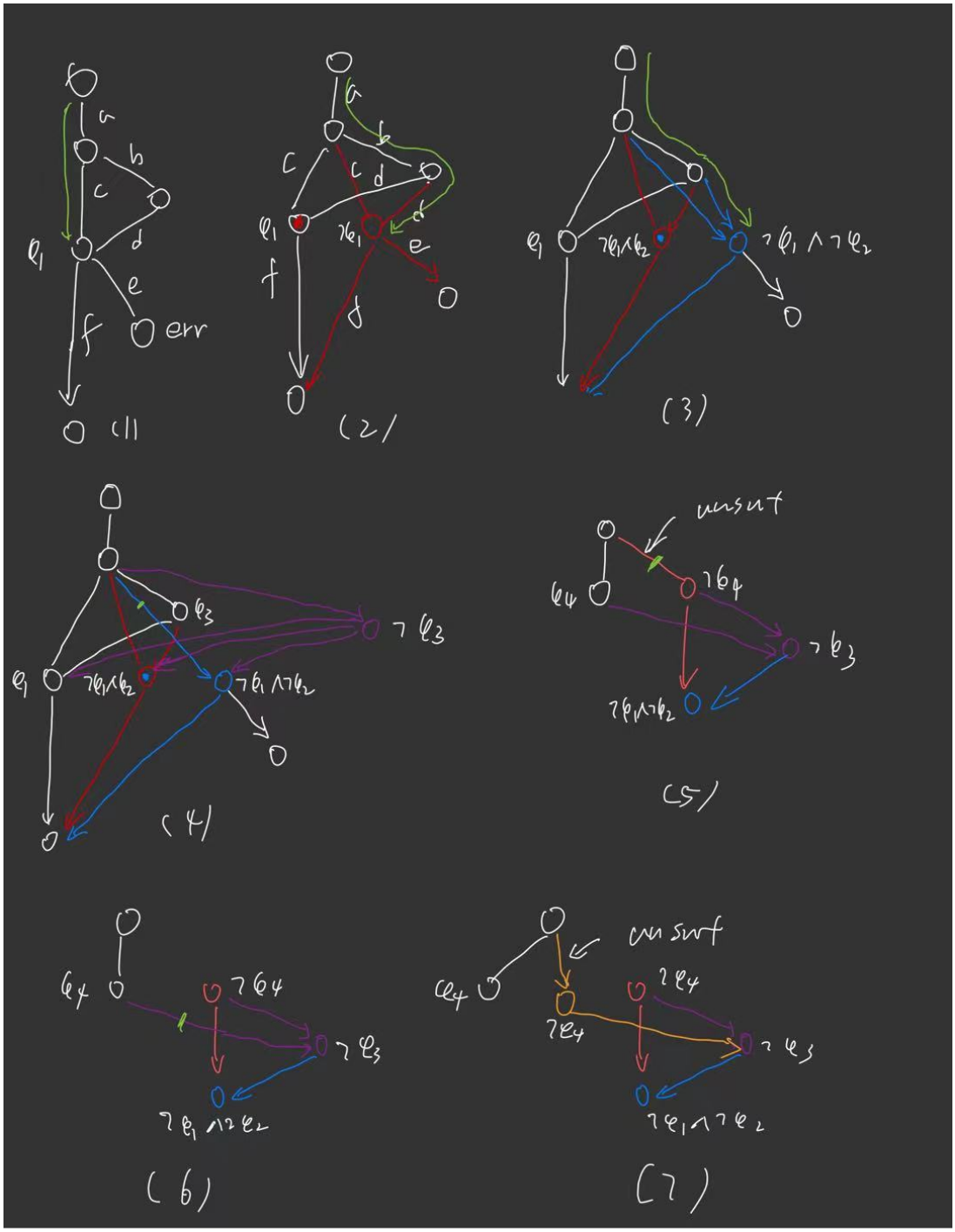


图 3: Refinement by node copying

其余过程类似. 它显然对应了我们前面提到的第一种方法, 最后图上不存在从开始结点到 error 结点的显式路径. 但是分析过程是非常 impractical 的, 相关图的扩展几乎是成指数型的, 并且在分析过程中会发现有一些分析步骤是冗余的. 最后总结一下这种方法内在本质:

- 不断地寻找 frontier, 切割 frontier 的一段. 新增的 edges 可以作为新 frontier，这些新增的 edges 包括了切割 froniter 一段的前驱, 这样我们实际总是在压缩前置路径. 直到 frontier 一端为 entry node, 那么此时可选的路径就是只能是 frontier 本身. 如果依然是 unsat, 那么就可以证明确实不存在这样的一条从 entry node 出发的路径.

- 实际处理的路径要比真实存在的路径要多的. 我们想一个极端情况图4, 假设原 CFG 上只有一条指向 target 的路径.

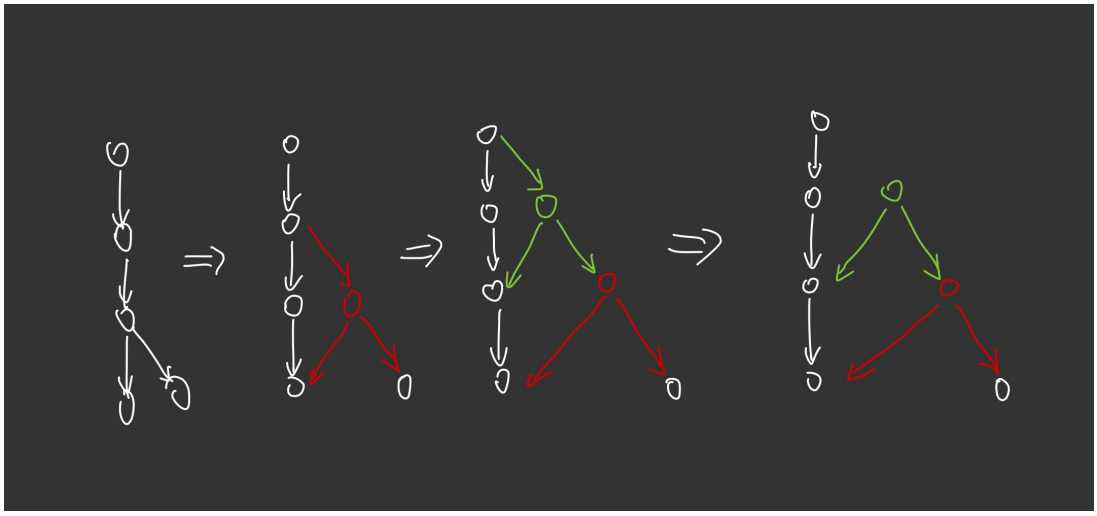


图 4: Refinement for single path

可以看到这里处理的路径个数和路径的长度是关系的, 准确来说如果 CFG 上实际存在 n 条候选错误路径, 那么实际可能分析 $O(2^n)$ 路径. 这个直觉来源于你如果 copy 一个结点，那么你就会将通过这个点的所有路径数乘 2.

- 准确来说我们总是 strengthen 某个结点上的条件，这个 strengthen 过程就是通过新增一个 conjunct 再取反. i.e, $\neg(\neg\varphi_1 \wedge \varphi_2)$. 这里的直觉是什么呢? 我们知道一个 predicate 可以将整个程序状态空间一分为二, 满足它的状态和不满足它的状态, 对应这个 predicate 本身和它的取反形式. 例如当 φ_1 对应的状态无法 reach 到某个结点, 那么可能 reach 这个结点的状态只能在 $\neg\varphi_1$ 中了. 接着我们分析了一条新路径在对应结点产生 φ_2, 此时原本这个结点上还有一个 $\neg\varphi_1$, 那么实际的条件就是 $\neg\varphi_1 \wedge \varphi_2$. 可以看出来我们在 $\neg\varphi_1$ 基础上

又 strengthen 了一次，如果这个条件依然不行，那么再取反分割结点. 两次取反你会发现 φ_1 又回来了, 但是我们之前已经排除了 φ_1. 这个方案告诉你取反并不是一种比较好的办法，它太弱了.

我们可以直接新增一个取反的后 conjunct, 不再直接取反, i.e., $\neg\varphi_1 \wedge \neg\varphi$. 这相当于就是把已经探索过的路径条件都给收集起来了.

- 每次只会对一个结点加 annotation, 即 frontier 上的一段. 这样的后果就是我们得重复计算路径条件, 参考图4.

Annotation 4.22. 前面第一个例子里面用到了给结点加 label 的概念, 我们可以给它形式化一下, 设 label map $f_l : V_2 \to L$. 那么上面的例子中 L 为 logic formulas $\mathcal{L}(\Sigma)$, 于是 G_2 上每一个结点都对应一个 formula, 没有显式标注的 label 的结点, 都默认 label 为 true. 后面记录两种对上述 navie 方法的改进:

- Unfolding.
- Lazy annotation.

Definition 4.23. Given label map $f_l : V_2 \to \mathcal{L}(\Sigma)$ of graph $G_2 = (V_2, E_2)$ and $u, w \in V_2$. We say u is covered by w, if

- $f_v(u) = f_v(w)$ and
- $f_l(u) \supset f_l(w)$ ($\supset$ w.r.t implies).

Annotation 4.24. 当 label 作为所在结点需要满足的状态时，u is covered by w 就意味着 w 可以 reachable 的结点，那么 u 也一定是可以 reachable 的. 按照这个思路我们就可以尝试去优化前面的分析方法.

Example 4.25. Example 4.21中实际在划分某一结点的所有程序状态, 将不可达的程序状态都从 CFG 上剥离出去了.

Model Checking

Bounded Model Checking

Interpolation and SAT-Based Model Checking

Annotation 5.1. 遵循 unrolling 的策略，给定一个初始状态 I_0, check 它所有 k 可达的状态. 这里我们可以利用 interpolation 的想法给出一个关于 one-step 可达状态的 over-approximation, 让我们在不需要增加 k 的情况下尽可能地往前分析. 这个想法的根本是如果在某个比较大的状态集中我们的 checking 通过了，显然那个被盖住的真正的状态集也是 safe 的. 同时配合 bounded model checking, 增大 bounder k, 我们也可以消除 over-approximation 带来的 failure checking.

Procedure FiniteRun($M = (I, T, F), k > 0$)

```
1  if I ∧ F is satisfiable then
2  |  return true
3  end
4  R ← I
5  while true do
6  |  M' ← (R, T, F)
7  |  A ← CNF(PREF_1(M'), U_1)
8  |  B ← CNF(SUFF_0^k(M'), U_2)
9  |  if A ∧ B is satisfiable then
10 |  |  if R = I then
11 |  |  |  return true
12 |  |  else
13 |  |  |  abort
14 |  |  end
15 |  else
16 |  |  Let Π be a proof of unsatisfiability of A ∧ B
17 |  |  P ← ITP(Π, A, B)
18 |  |  R' = P⟨W/W_0⟩
19 |  |  if R' ⇒ R then
20 |  |  |  return false
21 |  |  end
22 |  |  R ← R ∧ R'
23 |  end
24 end
```

Annotation 5.2. 上面的算法完整的描述利用 interpolation 在 k-bound 下做的 checking，目的是找到满足某种 property 的状态. 其中 $M = (I, T, F)$ 为一个 target automata, I 和 F 分布为 initial state predicate 和 final state predicate, T 是描述状态转移的一个 predicate. SAT-based BMC 的目标为

$$I(W_0) \wedge \left(\bigwedge_{0 \le i < k} T(W_i, W_{i+1}) \right) \wedge \left(\bigvee_{j \le i \le k} F(W_i) \right)$$

是否 satisfiable? 简而言之, 能否从初始状态 k 步可达的状态中找到一个满足 F 的状态. 在 while 之前先判断了是否存在 0 步的情况. 其中 R 表示当前所有可达的状态，初始化为 I. 在 while 内，会首先创建一个 new

automata, 将 R 作为新的状态，去做新一轮的 k-bounded checking. 这里将上面的目标分为了两个部分，前置路径和后置路径. 其中 A 表示 1 步前置路径

$$R(W_{-1}) \wedge T(W_{-1}, W_0),$$

B 表示 k 步后置路径

$$\left(\bigwedge_{0\le i\le k} T(W_i, W_{i+1})\right) \wedge \left(\bigvee_{j\le i\le k} F(W_i)\right)$$

注意这里变成了 $k+1$ 步. 如果在第一次循环下就有 $A \wedge B$ is sat, 那么这个时候 checking 就结束了, 因为这里的 R 此时并不是 over-approximation. 反之则直接 abort 了, 需要重新调整 k 才行, 因为我们不能保证这里是误报还是真的 counterexample. 如果 $A \wedge B$ is unsat, 那么这个时候我们就可以 induce 出来一个 interpolant P 满足 $A \to P$ and $P \wedge B$ is unsat. 接着我们需要将 P 中的 prime variables 变回一般的 state space 得到 R'. 此时从 R' 出发的 k 步状态都是不满足 F 的, 并且从 R 出发的 $k+1$ 步状态都是不满足 F 的. 此时的 R' 可以理解为关于 R 出发 1 步可达状态的一个 over-approximation. 因此下一轮 R 应该为 $R \vee R'$, 这个新的 R 保持了从 R 出发的 k 步状态都是不满足 F，因此需要往后考虑一下 $k+1$ 步. 所以这个 k 实际是限制 R' 的步长的. 最重要地是理解算法中的红色部分，如果有 $R' \Rightarrow R$, 那么 R 就是一个 inductive invariant. 这是因为 $I \Rightarrow R$ 是 trivial 的，再根据 $R \wedge T \to R'$ (R' is interpolant)，显然有 $R \wedge T \to R$. 当 R 是一个 inductive invariant 的时候，再配合 $R \wedge F$ is unsat, 可以知道所有 reachable states 都是不满足 F, 我们的 checking 也结束了.

上面实际更像更像 $k+1$ BMC. 我们来考虑一下细节，先考虑直接划分 k BMC 得到 A, B

$$\underbrace{I(W_0) \wedge T(W_0, W_1)}_{A} \wedge \underbrace{\left(\bigwedge_{0<i<k} T(W_i, W_{i+1})\right) \wedge \left(\bigvee_{j\le i\le k} F(W_i)\right)}_{B}$$

假设 C 是关于 $A \wedge B$ 一个 interpolant. 到底应该如何理解 C 是个什么呢? 问题的关键是 A 和 B 实际呈现是什么东西. A 应该是 characterize 从初始状态一步可达状态，这样 $A \to C$ 就意味着从初始状态一步可达状态都满足 C. 而 B 应该是 characteriz 从 $k-1$ 可达最终状态 ($\neg P$) 的状态, 这样 $C \wedge B$ is unsat 就意味着满足 C 的状态都至少不能在 $k-1$ 步到底最终状态.

IC3

Annotation 5.3. IC3 的核心就是 strengthening property 直到 inductive 或者遇到反例无法用 unreachability 剔除. 这种想法把常规的 abstraction 和 refinement 过程糅杂地不那么明显了. 比较难理解是 clauses 传递过程, 正确的视角是把一个 clause 当做一个 predicate，整个 IC3 依然是在 predicate abstraction 的范畴下. “向前传递”实际上对应了 post operation, 而新增 clause 实际是在 refine 原本的 abstract model 消除反例.

对于一个给定的 transition system $S : (\bar{x}, I, T)$ 和一个 safety property P. 整个过程将构造一串 formulas $F_0 = I, F_1, F_2, \cdots, F_k$ 表示对应 $0, 1, \cdots, k$ 可达状态的 over-approximation，它们满足下述性质:

1. $I \Rightarrow F_0$;
2. $F_i \Rightarrow F_{i+1}$ for $0 \leq i < k$;
3. $F_i \Rightarrow P$ for $0 \leq i \leq k$;
4. $F_i \wedge T \Rightarrow F'_{i+1}$ [1] for $0 \leq i < k$.

一旦出现 clauses(F_i) = clauses(F_{i+1}), 那么 F_i 就是我们要找的那个 strengthening clauses such that $F_i \wedge P \wedge T \Rightarrow F_i \wedge P$.

接下来看看这串 F_i 是如何构造的.

- Base case: check the satisfiability of $I \wedge \neg P$ ($I \Rightarrow P$ is true e.q. to $I \wedge \neg P$ is unsat) and $I \wedge T \wedge \neg P'$. 这是对 0-step 和 1-step 的一个 check.
- Inductive case: 当 $k > 0$ 时, 假设 $F_0, F_1, \cdots, F_k$ 满足上述 4 个性质. 现在来尝试构造 F_{k+1}. 首先 check $F_k \wedge T \Rightarrow P'$ 是否存在.
  - 如果 $F_k \wedge T \Rightarrow P'$. 此时让 $F_{k+1} = P$, 显然当前 sequence 依然保持上述 4 个性质. 另外还要做一个操作, 将 $F_0, F_1, \cdots, F_k$ 中的 clauses "向前移动": 对于任意 $c \in F_i$ $(0 \leq i \leq k)$, 如果 $F_i \wedge T \Rightarrow c'$ 并且 $c \notin$ clauses(F_{k+1}), 那么让 $F_{k+1} \leftarrow F_{k+1} \wedge c$. 如果这个操作过程中发现存在 clauses$(F_i)$ = clauses(F_{i+1}), 那么找到了 strengthening clauses F_i.
  - 如果 $F_k \wedge T \not\Rightarrow P'$. 那么这里一定存在一个 state $s \models F_k$ 使得它的下一个状态并不满足 P. 这里下一步的做法就是尝试把 s 剔除, 就是要证明 s 是 unreachable. 也就是证明 $\neg s$ 是 inductive invariant. 首先确定 $0 \leq i \leq k$ 中最大的 i 使得 $\neg s$ is inductive relative to F_i, 即 $i \Rightarrow \neg s$ and $F_i \wedge \neg s \Rightarrow \neg s$. 如果这样的 i 不存在, 即 $\neg s$ is not even inductive relative to F_0, 那么显然 s 就无法排除了, 这就意味着 P is not inductive invariant. 如果这样的 i 存在, 再对 s 做一次 inductive generalization, 找到 mininal subclause $c \subseteq \neg s$ that is inductive relative to F_i. 接着将 c 作为一个 conjunct 依次放到 $F_0, \cdots, F_{i+1}$ 上. 首先把 c 放到 $F_0, \cdots, F_i$ 上并不会违反上述四个性质, 这一点很容易 check. 其次最后是 F_{i+1}, 这是因为 $F_i \wedge \neg s \wedge T \Rightarrow \neg s'$, 配合性质 (4), 可以得到 $F_i \wedge \neg s \wedge T \Rightarrow F'_{i+1} \wedge \neg s'$.

    如果 $i \geq k-1$, 显然 s 已经被排除在 F_k. 比较难处理的是 $i < k-1$ 的情况, 此时并没有把 s 从 F_k 中去掉. 这里需要先思考一下为什么 $\neg s$ 会是在 F_i 上 inductive 而不是 F_{i+1}? 这里可以形式化一下这个问题:

$$F_{i+1} \wedge \neg s \wedge T \not\Rightarrow \neg s$$

$$F_i \wedge \neg s \wedge T \Rightarrow \neg s$$

[1] 这里是 F'_{i+1} 而不是 F_{i+1}, 如果写成后者, 这里就会与 (2) 出现歧义. F'_{i+1} 表示将 F_{i+1} 中的 variables $\{x_i\}$ 都换成带 prime 的 variables $\{x'_i\}$

那么这里必然存在一个状态 t 满足

$$t \models F_{i+1} \text{ and } t \not\models F_i \text{ and } (t, s) \in T.$$

也就是说存在一个 F_{i+1} 中的状态 t，它下一个状态满足 s. 如果能将这样的 t 从 F_{i+1} 都剔除去，那么也会有 $\neg s$ is inductive relative to F_{i+1}. 现在的目标就是让 $\neg t$ is inductive relative to F_i (注意这里不是 $i+1$), 和前面对 s 的处理是高度重合的. 但这里有一个好的性质是当 $i > 0$ 时，$\neg t$ 至少在 F_{i-1} 是 inductive 的. 这是因为根据前面的条件，这里有 $F_i \Rightarrow \neg t$, 所以这里的难度小于对 s 的处理. 接着我们继续考虑使用这样的方法处理 $i+2, i+3, \cdots, k$, 最终 $\neg s$ 会在 F_{k-1} 上 inductive，这样就把 s 从 F_k 中剔除了. 我们又回到了考虑 $F_k \wedge T \Rightarrow P'$ 的情况.

现在我们的条件应该是这样的

$$\begin{cases} s \models F_k \\ (s, u) \in T \\ u \not\models P \end{cases}$$

确定一个小 claim: 当 $k > 2$ 时，$\neg s$ is inductive relative to F_{k-2}. 假设 $\neg s$ is not inductive relative to F_{k-2}, 则

$$F_{k-2} \wedge \neg s \wedge T \Rightarrow s.$$

再配合一下性质 (4), 显然我们得到

$$F_{k-2} \wedge \neg s \wedge T \Rightarrow s \wedge F_{k-1}$$

这就意味着 s 也在 F_{k-1} 中，这和 $F_{k-1} \wedge T \Rightarrow P$ 是矛盾的. 上面这个 claim 意味着我们找那个最大值 i 似乎也不太麻烦, 甚至有点一击必中的感觉.

Annotation 5.4. 我觉得比较好的理解 IC3 的顺序应该是先看看 understanding IC3, 理解里面介绍的 incremental construction 是什么，再看看 IC3 之前的作者另一个工作 FSIS 里面的 inductive generalization 思想. 最后再来主攻 IC3.

Martain Löf Type Theory

Universe and Families

Definition 6.1. A universe is type whose elements are types.

Annotation 6.2. 为了避免 Russell's paradox 在集合论中尴尬，可以定义一组分层的 universe:

$$\mathcal{U}_0 : \mathcal{U}_0 : \mathcal{U}_2 : \cdots$$

其中每一个 $\mathcal{U}_i$ 是 $\mathcal{U}_{i+1}$ 中的元素. 同样允许当 $A : \mathcal{U}_i$, 也有 $A : \mathcal{U}_{i+1}$. 但是这里可能有一个小问题就是这样使得 A 的类型不唯一了. 当某个 $\mathcal{U}$ 被确定了之后，它包含的 types 通常称为small types.

Definition 6.3. Given type A, the functions $B : A \to \mathcal{U}$ whose codomain is a universe are called families of type A.

Annotation 6.4. family 是一个比较重要的 component, 它用于描述 a collection of types varying over a given type A.

Example 6.5. 设 $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$, 其结果表示恰好只存在 n 个元素它们的类型为 $\mathrm{Fin}(n)$, 也就是说" 满足" $\mathrm{Fin}(n)$ 的元素只有 n 个, 即 $0_n : \mathrm{Fin}(n), 1_n : \mathrm{Fin}(n), \cdots, (n-1)_n : \mathrm{Fin}(n)$. 给出一个 agdastdlib 里面的构造

$$\mathrm{zero} : \mathrm{Fin}\ (\mathrm{suc}\ n)$$

$$\mathrm{suc} : (i : \mathrm{Fin}\ n) \to \mathrm{Fin}\ (\mathrm{suc}\ n)$$

每个 type Fin n 都有一元素 zero, 而对于每一个 $i_n : \mathrm{Fin}\ (n)$ 都可以得到一个 $i_{n+1} : \mathrm{Fin}\ (n+1)$.

Example 6.6. 一个 constant type family 为 $B : \mathcal{U}$.

Dependent Function Types(Π-types)

Definition 6.7. Given a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, then $\Pi_{(x:A)} B(x) : \mathcal{U}$ is type of dependent functions whose domain is element x of type A and codomain is element of type $B(x)$. There is a alternative notation for this type, such as $\Pi(x : A).B(x)$.

Annotation 6.8. dependent funtion 和 type family 的区别是什么? 前者的函数输出 (function result) 为一个 term，后者的输出是一个 type.

Example 6.9. 取6.5中的 family $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$, 我们可以构造一个 dependent function $\mathrm{Fmax} : \Pi_{n:\mathbb{N}} \mathrm{Fin}(n+1)$, 其结果表示非空集合 $\mathrm{Fin}(n+1)$ 中" 最大值" (或者说最后一个元素) , 即 $\mathrm{Fmax}(n) :\equiv n_{(n+1)}$[2]

[2] :≡ 表示 function definition

Example 6.10. 当 family B 是一个 constant family 时，有 $\Pi_{(x:A)}B(x) \equiv A \to B$[3]，即 non-dependent function type. 这也说明 dependent function type 是一种 general form.

Example 6.11. 如果一个 dependent function 接受一个 type 作为参数，此时我们可以称它为 polymorphic function, 例如 polymorphic identity function $\mathrm{id} : \Pi_{A:\mathcal{U}} A \to A$.

Example 6.12. 如果一个 dependent functuon 同样可以接受多个参数，例如 polymorphic swap function $\mathrm{swap} : \Pi_{A:\mathcal{U}}\Pi_{B:\mathcal{U}}\Pi_{C:\mathcal{U}}(A \to B \to C) \to (B \to A \to C)$, 它对应的 function definition 为

$$\mathrm{swap}(A, B, C, g) :\equiv \lambda b.\, \lambda a.\, g(a)(b).$$

swap 可以用来交换 function 里面两个 arguments.

Dependent Pair Types

Annotation 6.13. 这里有一个 uniqueness principle，可以简单地理解为先用一次 introduction rule 构造 $a : A$, 再用一次 elimination rule 给 $a : A$" 拆开", 然后再用一次 introduction rule 我们又可以得到 $a : A$. 例如给定 $x : A \times B$, 那么它的 uniqueness principle 就是 $(\pi_1(x), \pi_2(x)) \equiv x$. 注意这是一个命题, 它表示所有 $x : A \times B$, 其 x 都是一个 ordered pair, 所以了这个 definitional equality 是需要证明的，因此也可以叫做 propositional uniqueness principle.

Annotation 6.14. 另外一个需要说明的是这里我们并没有规定 product type 就必须是一个 pair, 也就是说这个东西并没有作为规则写到 type theory 中. 当然你要是说 introductionl rule 里面似乎已经规定了, 那确实是，但是需要证明.

Definition 6.15. 对于任意的 function $g : A \to B \to C$, 我们都可以定义一个 function $f : A \times B \to C$ 使得

$$f((a, b)) :\equiv g(a)(b)$$

f is well-defined when we specify its values on pairs.

Definition 6.16. Two projection functions:

$$\pi_1 : A \times B \to A$$
$$\pi_2 : A \times B \to B$$

其中

$$\pi_1((a, b)) :\equiv a$$
$$\pi_2((a, b)) :\equiv b$$

[3] $\equiv$ 表示 definitional equality

Annotation 6.17. 上面两个 *projections* 中的 $g_1 : A \to B \to A$ 和 $g_2 : A \to B \to B$ 分别为

$$g_1 :\equiv \lambda x.\, \lambda y.\, x$$
$$g_1 :\equiv \lambda x.\, \lambda y.\, y$$

Definition 6.18. Given a product type $A \times B$. Define function $\mathrm{rec}_{A\times B}$ as the recursor for $A \times B$ with type

$$\mathrm{rec}_{A\times B} : \Pi_{C:\mathcal{U}}(A \to B \to C) \to A \times B \to C$$

and defining equation

$$\mathrm{rec}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b).$$

Example 6.19. 此时我们可以用 $\mathrm{rec}_{A\times B}$ 来定义 π_1 和 π_2:

$$\pi_1 :\equiv \mathrm{rec}_{A\times B}(A, \lambda a.\, \lambda b.\, a)$$
$$\pi_2 :\equiv \mathrm{rec}_{A\times B}(B, \lambda a.\, \lambda b.\, b)$$

有一个比较特殊的 nullary product type $\mathbf{1}$, 它只有一个 inhabitant 我们可以用 $\star : \mathbf{1}$ 表示. 它也有一个对应的 recursor $\mathrm{rec}_{\mathbf{1}} : \Pi_{C:\mathcal{U}} C \to \mathbf{1} \to C$, 它的 defining equation 为

$$\mathrm{rec}_{\mathbf{1}}(C, c, \star) :\equiv c$$

这是因为你从 $\mathbf{1}$ 中不能获取任何信息.

可以看到 recursion 实际是一个更为 general elimination rule (The recursion pinciple for catesian products is the fact that we can define a function $f : A \times B \to C$ as above by giving its value on pairs).

Annotation 6.20. 关于 product types 的 dependent functions $f : \Pi_{x:A\times B} C(x)$, 其中 $C : (A \times B) \to \mathcal{U}$. 我们也使用一个 elimination rule: 给定任意的 function $g : \Pi_{x:A}\Pi_{y:B} C((x, y))$, 我们可以定义 dependent function $f : \Pi_{x:A\times B} C(x)$.

$$f((x, y)) :\equiv g(x)(y).$$

Definition 6.21. A judgmental equality or definitional equality is the notion of equality which is defined to be a judgement, we write it as $a \equiv b : A$ or simply $a \equiv b$.

Definition 6.22. A proposition equality is a proposition, so it is also a type such that $a =_A b$, and to prove the proposition is to find term of type $a =_A b$.

Example 6.23. 现在我们来证明 product type 只能是一个 pair, 我们尝试构造这样一个 dependent function $\mathrm{uniq}_{A\times B} : \Pi_{x:A\times B}((\pi_1(x), \pi_2(x)) =_{A\times B} x)$, 通过 defining equation

$$\mathrm{uniq}_{A\times B}((a, b)) :\equiv \mathrm{refl}_{(a,b)}.$$

其中 $\mathrm{refl}_x : x =_A x$ for any $x : A$ 是一个 well-typed element. 那么 $C(x) \equiv (\pi_1(x), \pi_2(x)) =_{A\times B} x)$, 那么当 $x :\equiv (a, b)$, 我们可以得到

$$C(x) :\equiv (\pi_1(((a, b)), \pi_2((a, b))) =_{A\times B} (a, b)$$

显然有

$$(\pi_1(((a, b)), \pi_2((a, b))) \equiv (a, b)$$

因此 $\mathrm{refl}_{(a,b)} : (\pi_1(((a, b)), \pi_2((a, b))) = (a, b)$ 还是 well-typed. 所以关于 $\mathrm{uniq}_{A\times B}$ 整个构造是对的，如何进一步说明 x 确实是一个 pair 呢? 难度从 f 是 well-defined? 推出其所有结果都是 well-typed, 然后 x is equal to a pair.

Annotation 6.24. 如果你把前面的 dependent function f 当做一个 predicate 来描述所有类型为 product type 的元素的某种 property，同时把 (ordered) pair 当做 product type 的 canonical element. 那么就意味 f 只需要作用到这些 canonical elements 上就够了, 这样就可以导出 induction.

Definition 6.25. Given a product type $A \times B$. Define a function ind as the induction for $A \times B$ with the type

$$\mathrm{ind}_{A\times B} : \Pi_{C:A\times B\to\mathcal{U}}(\Pi_{x:A}\Pi_{y:A}C((x, y))) \to \Pi_{x:A\times B}C(x)$$

and the defining equation:

$$\mathrm{ind}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b).$$

Annotation 6.26. 这个就是更加 general 的 elimination rule 了, 如果前面的 recursion rule 叫做 non-dependent eliminator, 那么这个 induction 就可以叫做 dependent eliminator.

Example 6.27. 关于 unit type $\mathbf{1}$ 的 induction 为

$$\mathrm{ind}_{\mathbf{1}} : \Pi_{C:\mathbf{1}\to\mathcal{U}}C(\star) \to \Pi_{x:\mathbf{1}}C(x)$$

with the defining equation

$$\mathrm{ind}_{\mathbf{1}}(C, c, \star) :\equiv c.$$

通过 $\mathrm{ind}_{\mathbf{1}}$ 我们也可以知道 unit type 只有对应的一个元素，这就是 unit type 的 uniqueness principle. 因为我们构造一个 dependent function

$$\mathrm{uniq}_1 : \Pi_{x:\mathbf{1}}x = \star$$

with defining equation

$$\mathrm{uniq}_{\mathbf{1}} :\equiv \mathrm{ind}_{\mathbf{1}}(\lambda x.\, x = \star, \mathrm{refl}_\star)$$

Definition 6.28. Given a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, the dependent pair type is written as $\sum_{(x:A)} B(x)$.

Example 6.29. 当 B 是一个 constant 的时候, 此时 $\Sigma_{(x:A)}B$ 就是 product type.

Definition 6.30. Given a function $g : \Pi_{x:A} B(x) \to C$, we can define a function out of a Σ-type $f : (\Sigma_{x:A} B(x)) \to C$ via the defing equation

$$f((a, b)) = g(a)(b)$$

Example 6.31. 关于 Σ-type 两个 non-dependent projection function 定义如下: the first projection function

$$\pi_1 : (\Sigma_{x:A} B(x)) \to A$$

with the defining equation

$$\pi_1((a, b)) :\equiv a.$$

The second projection function

$$\pi_2 : \Pi_{p:(\Sigma_{x:A} B(x))} B(\pi_1(p))$$

这里我们还无法构造 π_2，因为我们还不知道 Σ-type 的 dependent function 如何构造.

Definition 6.32. Given a function $g : \Pi_{a:A} \Pi_{b:B(a)} C((a, b))$, we can define a dependent function $f : \Pi_{p:\Sigma_{x:A} B(x)} C(p)$ with defining equation

$$f((a, b)) :\equiv g(a)(b)$$

Definition 6.33. The induction for Σ-type $\Sigma_{x:A} B(x)$ is

$$\mathrm{ind}_{\Sigma_{x:A} B(x)} : \Pi_{C:(\Sigma_{x:A} B(x) \to \mathcal{U})} (\Pi_{a:A} \Pi_{b:B(a)} C((a, b))) \to \Pi_{p:(\Sigma_{x:A} B(x))} C(p)$$

with defining equation.

$$\mathrm{ind}_{\Sigma_{x:A} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

Example 6.34. Assume $R : A \to B \to \mathcal{U}$, then type-theoretic axiom of choice is

$$\mathrm{ac} : \Pi_{x:A} \Sigma_{y:B} R(x, y) \to \Sigma_{f:A \to B} \Pi_{x:A} R(x, f(x))$$

with defining equation

$$\mathrm{ac}(g) :\equiv (\lambda x.\, \pi_1(g(x)), \lambda x.\, \pi_2(g(x))).$$

将 Π 看做 $\forall$, 而将 Σ 看做 $\exists$, 就比较好理解了.

Coproduct Type

Definition 6.35. A coproduct type is formed as $A + B$ with introductions rules

$$\frac{a : A}{\mathrm{inl}(a) : A + B}\ +\textsc{Intro}_1 \qquad \frac{b : A}{\mathrm{inr}(b) : A + B}\ +\textsc{Intro}_2$$

Definition 6.36. The recursion principle of coproduct type is

$$\mathrm{rec}_{A+B} : \Pi_{C:\mathcal{U}}(A \to C) \to (B \to C) \to A + B \to C$$

with defining equation

$$\mathrm{rec}_{A+B}(C, g_0, g_1, \mathrm{inl}(a)) :\equiv g_0(a)$$
$$\mathrm{rec}_{A+B}(C, g_0, g_1, \mathrm{inr}(b)) :\equiv g_1(b)$$

where $g_0 : A \to C$ and $g_1 : B \to C$.

Definition 6.37. The induction principle of coproduct type is

$$\mathrm{ind}_{A+B} : \Pi_{C:A+B\to\mathcal{U}}(\Pi_{x:A}C(\mathrm{inl}(x))) \to (\Pi_{y:B}C(\mathrm{inr}(y))) \to \Pi_{z:A+B}C(z)$$

with defining equation

$$\mathrm{ind}_{A+B}(C, g_0, g_1, \mathrm{inl}(a)) :\equiv g_0(a)$$
$$\mathrm{ind}_{A+B}(C, g_0, g_1, \mathrm{inr}(b)) :\equiv g_1(b)$$

where $g_0 : \Pi_{x:A}C(\mathrm{inl}(x))$ and $g_1 : \Pi_{y:A}C(\mathrm{inr}(y))$.

Concrete Construction

Definition 6.38. The constructors for $\mathbb{N}$ are

$$0 : \mathbb{N}$$
$$\mathrm{succ} : \mathbb{N} \to \mathbb{N}$$

Definition 6.39. Primitive recursion of $\mathbb{N}$ is

$$f(0) :\equiv c_0$$
$$f(\mathrm{succ}\ n) :\equiv c_s(n, f(n))$$

where $f : \mathbb{N} \to C$, $c_0 : \mathbb{N}$ and $c_s : \mathbb{N} \to C \to C$.

Annotation 6.40. 注意上面用到了 primitive 这个词，它意味着这个 recursion 可以盖住所有 $f : \mathbb{N} \to C$ 的构造. 如果想要构造 $\mathbb{N}$ 上的加法, 我们只需要这个 C 换成 $N \to N$, 那么此时 $c_0 : \mathbb{N} \to \mathbb{N}$，而 $c_s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$. 此时 f 我们用 add 来表示，那么 $\mathrm{add}(n)$ 实际就是 plus operator $n+$, $\mathrm{add}(0) :\equiv c_0$ 表示即 $0+$, 而 $\mathrm{add}(\mathrm{succ}\ n) = c_s(n, add(n))$ 表示 $n + 1+$.

Definition 6.41. Recursor of $\mathbb{N}$ is

$$\mathrm{rec}_{\mathbb{N}} : \Pi_{C:\mathcal{U}} \to C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$$

with defining equation

$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0$$
$$\mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}\ n) :\equiv c_s(n, \mathrm{rec}_{\mathbb{N}}(C, c_0, c_s, n))$$

Definition 6.42. Induction principle of $\mathbb{N}$ is

$$\mathrm{ind}_{\mathbb{N}} : \Pi_{C:\mathbb{N}\to\mathcal{U}} C(0) \to (\Pi_{n:\mathbb{N}} C(n) \to C(\mathrm{succ}\ n)) \to \Pi_{n:N} C(n)$$

with defining equation

$$\mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, 0) :\equiv c_0$$
$$\mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, \mathrm{succ}\ n) :\equiv c_s(n, \mathrm{ind}_{\mathbb{N}}(C, c_0, c_s, n))$$

Programming Language Definition

Annotation 6.43. Given the type $\Sigma_{x:A} P(x)$, we can regard it as the type of all elements $x : A$ such that $P(x)$.

Algebraic Structure Definition

Example 6.44. A type of magma is

$$\mathrm{Magma} : \Sigma_{A:\mathcal{U}} A \to A \to A.$$

Magma 可以刻画那些带有 binary operator 的代数结构. 其中 A 实际就代表了 a set of elements, 我们可以通过 π_1 拿到. 而其 binary operator, 我们可以通过 π_2 拿到.

Example 6.45. A type of pointed-magma is

$$\mathrm{PointedMagma} : \Sigma_{A:\mathcal{U}} (A \to A \to A) \times A.$$

相比于 Magmas, 这里多出了一些点 $x : A$, 例如我们常常需要的 identity element.

Example 6.46. A type of boolean is $\mathbf{2} : \mathcal{U}$ is intended to have exactly two elements $0_2, 1_2 : \mathbf{2}$. 我们可以用 coproduct type 配合 unit type 来定义它, i.e., $\mathbf{1} + \mathbf{1}$, 这里就正好有两个元素. 当然了也可以显式地直接定义 $\mathbf{2}$.

依赖 boolean type 的 if-then-else 对应的 recursion principle 为

$$\mathrm{rec}_{\mathbf{2}} : \Pi_{C:\mathcal{U}} C \to C \to \mathbf{2} \to C$$

with defining equation

$$\mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 0_2) = c_0$$
$$\mathrm{rec}_{\mathbf{2}}(C, c_0, c_1, 1_2) = c_1$$

Identity Types

Definition 6.47. Given two elements $ab : A$, there is type $a =_A b : \mathcal{U}$ called *identity type* or *equality type* in the same universe.

Definition 6.48. The only introduction rule for identity types is a dependent function

$$\mathbf{refl} : \Pi_{a:A}(a =_A a)$$

called reflexivity.

Definition 6.49. (Indiscernibility of identicals) For every family $C : A \to \mathcal{U}$, there is a function

$$f : \Pi_{x,y:A}\Pi_{p:x=_A y}\ C(x) \to C(y)$$

such that $f(x, x, \mathbf{refl}_x) :\equiv \mathbf{id}_C(x)$.

Annotation 6.50. Indiscernibility of identicals 描述当两个东西是相同的时候, 当其中一个满足某种性质的时候, 另外一个也应当满足这个性质. 在任何使用它们的场景下都是无法区分它们的.

Lemma 6.51. (Principle of substitution) To prove that every x, y with $x = y$ has property $P(x, y)$, it suffices to prove that every pair x, x (for which $x = x$) has property $P(x, x)$

证明. By Indiscernibility of identicals for $x = y$, we know $P(x, x) \to P(x, y)$ (fixed the first argument). □

Annotation 6.52. MLTT 中也有一个类似 path induction 的 identify type 的 elimination rule.

Homotopy Type Theory

Annotation 7.1. *Topologically, ...*

Definition 7.2. Topologically, a identity type is a path.

Path Induction

Annotation 7.3. The family of identity types is freely generated by the reflexivity terms.

Definition 7.4. (Path induction) Given a family $C : \Pi_{x,y:A}(x =_A y) \to \mathcal{U}$ and a function $c : \Pi_{x:A} C(x, x, \mathbf{refl}_x)$, there is a function

$$f : \Pi_{x,y:A} \Pi_{p:x=_A y} C(x, y, p)$$

such that $f(x, x, \mathbf{refl}_x) :\equiv c(x)$. It corresponds a single function

$$\mathbf{ind}_{=A} : \Pi_{C:\Pi_{x,y:A}(x=_A y)\to\mathcal{U}} \left(\Pi_{x:A} C(x, x, \mathbf{refl}_x)\right) \to \Pi_{x,y:A} \Pi_{p:x=_A y} C(x, y, p)$$

with the equality $\mathbf{ind}_{=A}(C, c, x, x, \mathbf{refl}_x) :\equiv c(x)$.

Annotation 7.5. Given a identity type family C, to prove that every pair x, y with path $p : x = y$ has property $C(x, y, p)$, it suffices to prove that every pair x, x with loop $\mathbf{refl}_x$ has property $C(x, x, \mathbf{refl}_x)$.

Annotation 7.6. 辅助理解 path induction, 思考 $1 + 3 = 3 + 1$ 的证明过程. 这里应该有一些类似的证明，例如

$$1 + 3 = 0 + 4 = 4 = 4 = 1 + 3 = 2 + 2 = 3 + 1$$

但是无论怎样我们都无法直接用 $\mathbf{refl}_{1+3}$ 或者 $\mathbf{refl}_{3+1}$ 来一步到位，而是连续地使用了 **refl**. 这里看起来 identity type 是 inductive defined, 注意这并不对! 这里 $1 + 3 = 3 + 1$ 是一个 non-dependent identity type family $C(x, y, p) :\equiv 1 + 3 = 3 + 1$, 因此还是 identity type family 是 inductive defined. 可以更 general 一点，来证明 $1 + x = x + 1$, 我们也可以借助 identity type，构造一个 family $C(x, y, p) :\equiv 1 + x = y + 1$.

Definition 7.7. (Based path induction) Fix an element $a : A$, we have a dependent function

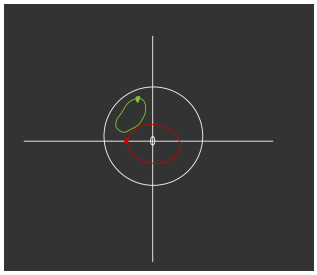
$$\mathbf{ind}'_{=A} : \Pi_{a:A} \Pi_{C:\Pi_{x:A}(a=_A x)\to\mathcal{U}} C(a, \mathbf{refl}_a) \to \Pi_{x:A} \Pi_{p:a=_A x} C(x, p)$$

with the equality

$$\mathbf{ind}'_{=A}(a, C, c, a, \mathbf{refl}_a) :\equiv c.$$

Annotation 7.8. 注意这里不能同时固定 $x, y : A$. 特别地，换句话说我们不能只通过 $C(\mathbf{refl}_x)$ 来证明 $C(p)$ with $p : x =_A x$. 因为 HoTT 允许 x, x 之间的 paths 不只有 reflexivity.

从拓扑的观点来看，一个两端固定的绳子可能不能连续形变到同一个点 (constant path (路径只在某一点上)). 例如给定挖掉圆心的圆 $\{(x, y) | 0 < x^2 + y^2 < 2\}$. 我们来考虑两种情况



绿色的环显然是可以连续形变到这个环上的某个点, 但是红色的环不行. 因为中间的这个洞阻碍了连续形变. 如果运行这个环的某个端点可变, 显然我们可以一直收缩这个端点到另外一个端点. 连续形变相当于在其过程中产生了很多的 paths, 这些 paths 都是" 连续" 的或者 homotopical.

Lemma 7.9. Path induction and based path induction are equivalent.

证明. ($\Leftarrow$) 是简单的, 即

$$\begin{aligned}\mathrm{eq}_l :&(\mathbf{ind}_2 : (\Pi_{a:A}\Pi_{C:\Pi_{x:A}(a=_A x)\to\mathcal{U}}\ C(a,\mathbf{refl}_a) \to \Pi_{x:A}\Pi_{p:a=_A x}C(x,p)))\\ \to&(\mathbf{ind}_1 : (\Pi_{C:\Pi_{x,y:A}(x=_A y)\to\mathcal{U}}\ (\Pi_{x:A}C(x,x,\mathbf{refl}_x)) \to \Pi_{x,y:A}\Pi_{p:x=_A y}C(x,y,p)))\end{aligned}$$

with the equality $\mathrm{eq}_l(\mathbf{ind}_2, C, c, x, y, p) = \mathbf{ind}_2(x, C(x), c(x), y, p)$.

反过来就有点不太明显了, 即要证

$$\begin{aligned}\mathrm{eq}_r :&(\mathbf{ind}_1 : (\Pi_{C:\Pi_{x,y:A}(x=_A y)\to\mathcal{U}}\ (\Pi_{x:A}C(x,x,\mathbf{refl}_x)) \to \Pi_{x,y:A}\Pi_{p:x=_A y}C(x,y,p)))\\ \to&(\mathbf{ind}_2 : (\Pi_{a:A}\Pi_{C:\Pi_{x:A}(a=_A x)\to\mathcal{U}}\ C(a,\mathbf{refl}_a) \to \Pi_{x:A}\Pi_{p:a=_A x}C(x,p)))\end{aligned}$$

这个时候没法直接用 $\mathbf{ind}_1$. 这里我们需要利用 path induction 去额外证明一个 property

$$f : \Pi_{x,y:A}\Pi_{p:x=y}\Pi_{C:\Pi_{z:A}(x=_A z)\to\mathcal{U}}\ C(x,\mathbf{refl}_x) \to C(y,p).$$

使得 $D : \Pi_{x,y:A}(x = y) \to \mathcal{U}$ with $D(x, y, p) = \Pi_{C:\Pi_{z:A}(x=_A z)\to\mathcal{U}}C(x,\mathbf{refl}_x) \to C(y,p)$. 这里 D 就是 $x = y$ 的一个 family, 我们可以利用 path induction 来证明

$$f : \Pi_{x,y:A}\Pi_{p:x=y}D(x,y,p)$$

显然有 $D(x,x,\mathbf{refl}_x) = \mathbf{id}_{C(x,\mathbf{refl}_x)}$. 因此我们有 $\mathrm{eq}_r(\mathbf{ind}_2, a, C, c, x, p) = f(a, x, p, C, c)$.

这里还需要注意是如果 eq_r 里面规定 $C : \Pi_{x:A}(a = x) \to \mathcal{U}_i$, 那么必须需要 $D : \Pi_{x,y:A}(x = y) \to \mathcal{U}_{i+1}$. 因为 D 的参数是 $c : C$, 由此可能会产生类型为 $\mathcal{U}_i$ 的元素, 盖住它们需要用更高一层的 universe. □

Annotation 7.10. 如何使用 path induction 呢? 也就是说如何用 path induction 证明一些 function f ?

如果我们 type family $D : \Pi_{x,y:A}(x = y) \to \mathcal{U}$ 和 $d : \Pi_{a:A} D(a, a, \mathbf{refl}_a)$, 那么我们就可以得到一个 dependent function

$$\mathbf{ind}_{=A}(D, d) : \Pi_{x,y:A}\Pi_{p:x=y} D(x, y, p)$$

with the definition equality $\mathbf{ind}_{=A}(D, d, a, a, \mathbf{refl}_a) :\equiv d(a)$.

首先 f 应当是关于 every x, y with $p : x = y$ 的一个 proprety, 此时就可以用 D 来抽象这个 property. 并且 $D(x, y, p)$ 应当是描述 property 中结果的那个 type. 有可能此时 $D(x, x, \mathbf{refl})$ 还需要继续用 path induction, 也就是无法直接构造 d. 在下面一个小节中可以慢慢体会这个过程.

Types are higher groupoids

Lemma 7.11. For every type A and every $x, y : A$ there is a function

$$(x = y) \to (y = x)$$

denoted $p \mapsto p^{-1}$, such that $\mathbf{refl}_x^{-1} \equiv \mathbf{refl}_x$ for each $x : A$. We call p^{-1} the inverse of p.

证明. The corresponding type for the proposition is

$$f : \Pi_{A:\mathcal{U}}\Pi_{x,y:A}(x = y) \to (y = x).$$

Let $D : \Pi_{x,y:A}(x = y) \to \mathcal{U}$ be the type family defined by $D(x, y, p) :\equiv (y = x)$. Then we have an element $d :\equiv \lambda x.\, \mathbf{refl}_x : \Pi_{x:A} D(x, x, \mathbf{refl}_x)$. Thus, the induction principle for identity types gives us an element $\mathbf{ind}_{=A}(D, d, x, y, p) : (y = x)$. □

Lemma 7.12. For every type A and every $x, y, z : A$ there is a function

$$(x = y) \to (y = z) \to (x = z)$$

written $p \mapsto q \mapsto p \cdot q$, such that $\mathbf{refl}_x \cdot \mathbf{refl}_x \equiv \mathbf{refl}_x$ for any $x : A$. We call $p \cdot q$ the concatenation or composite of p and q.

证明. The corresponding type for the proposition is

$$\Pi_{A:\mathcal{U}}\Pi_{x,y,z:A}\ (x = y) \to (y = z) \to (x = z)$$

Let $D_1 : \Pi_{x,y}(x = y) \to \mathcal{U}$ be the family defined by $D_1(x, y, p) \equiv \Pi_{z:A}\Pi_{q:y=z}(x = z)$. Now $D(x, x, \mathbf{refl}_x) = \Pi_{z:A}\Pi_{q:x=z}(x = z)$. Let $D_2 : \Pi_{x,z:A}\Pi_{q:x=z} \mathcal{U}$ be the family defined by $D_2(x, z, q) :\equiv x = z$. Now $D_2(x, x, \mathbf{refl}_x) \equiv (x = x)$, then we have $d_2 : \Pi_{x:A}\ D_2(x, x, \mathbf{refl}_x)$ defined by $d_2(x) = \mathbf{refl}_x$.

By induction principle for identity type we have $\mathbf{ind}_{=A}(D_2, d_2) : \Pi_{x,z:A}\Pi_{p:x=z} x = z$. Then we have $d_1 : \Pi_{x:A} D_1(x, x, \mathbf{refl})$ defined by $d_1(x) = \mathbf{ind}_{=A}(D_2, d_2)$. Again by induction principle for identity type we have $\mathbf{ind}_{=A}(D_1, d_1) : \Pi_{x,y:A}\Pi_{p:x=y}\Pi_{z:A}\Pi_{q:y=z}(x = z)$. Since p and q is unused, then $\mathbf{ind}_{=A}(D_1, d_1) : \Pi_{x,y:A}(x = y) \to \Pi_{z:A}(y = z) \to (x = z)$. Finally we have $\mathbf{ind}_{=A}(D_1, d_1) : \Pi_{x,y,z:A}(x = y) \to (y = z) \to (x = z)$. □

Definition 7.13. A ∞-groupoid consists of a collection of objects, and then a collection of morphisms between objects, and then morphisms between morphisms, and so on. Morphisms at each level have identity, composition, and inverse operations.

Remark 7.14. "equality" , "homotopical" , and "higher-groupoid" points of view on what we have done so far.

Equality	Homotopy	∞-Groupoid
reflexivity	constant path	identity morphism
symmetry	inversion of paths	inverse morphism
transitivity	concatenation of paths	composition of morphisms

Lemma 7.15. (2-path) Suppose $A : \mathcal{U}$, that $x, y, z, w : A$ and that $p : x = y$ and $r : z = w$. We have the following:

- $p = p \centerdot \mathbf{refl}_y$ and $p = \mathbf{refl}_x \centerdot p$.
- $p^{-1} \centerdot p = \mathbf{refl}_y$ and $p \centerdot p^{-1} = \mathbf{refl}_x$.
- $(p^{-1})^{-1} = p$.
- $p \centerdot (q \centerdot r) = (p \centerdot q) \centerdot r$.

证明. 我们在这里只证明 associativity. 它对应的 type 为

$$\Pi_{x,y,z,w:A}\Pi_{p:x=y}\Pi_{q:y=z}\Pi_{r:z=w}\ p \centerdot (q \centerdot r) = (p \centerdot q) \centerdot r$$

使得 $D_1 : \Pi_{x,y:A}\Pi_{p:x=y}\ \mathcal{U}$ defined by

$$D_1(x, y, p) = \Pi_z \Pi_{q:y=z}\Pi_{w:A}\Pi_{r:z=w}\ p \centerdot (q \centerdot r) = (p \centerdot q) \centerdot r$$

此时 $D_1(x, x, \mathbf{refl}_x) = \Pi_{z,w:A}\Pi_{q:x=z}\Pi_{r:z=w}\ \mathbf{refl}_x \centerdot (q \centerdot r) = (\mathbf{refl}_x \centerdot q) \centerdot r$.

再使得 $D_2 : \Pi_{x,z:A}\Pi_{q:x=z} \to \mathcal{U}$ defined by

$$D_2(x, z, q) = \Pi_{w:A}\Pi_{r:z=w}\ \mathbf{refl}_x \centerdot (q \centerdot r) = (\mathbf{refl}_x \centerdot q) \centerdot r.$$

此时 $D_2(x, x, \mathbf{refl}_x) = \Pi_{w:A}\Pi_{r:x=w}\ \mathbf{refl}_x \centerdot (\mathbf{refl}_x \centerdot r) = (\mathbf{refl}_x \centerdot \mathbf{refl}_x) \centerdot r$.

最后使得 $D_3 : \Pi_{x,w:A}\Pi_{r:x=w}\mathcal{U}$ defined by

$$D_3(x, w, r) = \mathbf{refl}_x \centerdot (\mathbf{refl}_x \centerdot r) = (\mathbf{refl}_x \centerdot \mathbf{refl}_x) \centerdot r$$

此时 $D_3(x, x, \mathbf{refl}_x) = \mathbf{refl}_x \centerdot (\mathbf{refl}_x \centerdot \mathbf{refl}_x) = (\mathbf{refl}_x \centerdot \mathbf{refl}_x) \centerdot \mathbf{refl}_x$. 因此 $d_3 : \Pi_x D_3(x, x, \mathbf{refl}_x)$ defined by $d_3(x) = \mathbf{refl}_{\mathbf{refl}_x}$, 注意这里是 2-path.

最后用两次 path induction, 构造 d_2 和 d_1, 和一次 path induction 证明原命题. □

Lemma 7.16. (Right whiskering) Suppose $a, b : A$ and $p, q : a = b$ and $\alpha : p = q$ and $r : b = c$, then we can define

$$a \centerdot_r r : p \centerdot r = q \centerdot r$$

证明. By induction on r, it suffices to assume r as $\mathbf{refl}_b$, then $a \centerdot_r \mathbf{refl} : p \centerdot \mathbf{refl}_b = q \centerdot \mathbf{refl}_b$ can be defined by

$$a \centerdot_r \mathbf{refl}_b :\equiv \mathbf{ru}_p{}^{-1} \alpha \mathbf{ru}_q$$

where $\mathbf{ru}_p : p = p \centerdot \mathbf{refl}_b$ and $\mathbf{ru}_q : q = q \centerdot \mathbf{refl}_b$. □

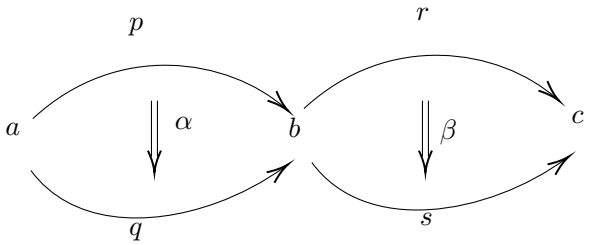
Lemma 7.17. (Left whiskering) Suppose $b, c : A$ and $r, s : b = c$ and $\beta : r = s$ and $q : a = b$, then we define

$$q \centerdot_l \beta : q \centerdot r = q \centerdot s$$

Annotation 7.18. 前面两个 whiskering lemma 告诉我们 2-path 可以直接 concact 相同 path 的, 来产生一个新的 2-path, 这就带来了一些等号上好的计算性质.

Lemma 7.19. (Horizontal composition) Suppose $a, b, c : A$ and $p, q : a = b$ and $r, s : b = c$ and $\alpha : p = q$ and $\beta : r = s$. Then we can define

$$\alpha \star \beta : p \centerdot r = q \centerdot s$$



证明. We can define $\alpha \star \beta : p \centerdot r = q \centerdot s$ by

$$\alpha \star \beta :\equiv (\beta \centerdot_r r) \centerdot (q \centerdot_l \beta).$$

□

Definition 7.20. A pointed type (A, a) is a type $A : \mathcal{U}$ together with a point $a : A$, called its basepoint.We write $\mathcal{U}_\bullet :\equiv \Sigma_{A:\mathcal{U}} A$ for the type of pointed types in the universe $\mathcal{U}$.

Definition 7.21. Given a pointed type (A, a), we define the loop space of (A, a) to be following pointed type:

$$\Omega(A, a) :\equiv ((a =_A a), \mathbf{refl}_a).$$

An element of it will be called a loop at a. For $n : \mathbb{N}$, the n-fold iterated loop space $\Omega^n(A, a)$ of a pointed type (A, a) is defined recursively by:

$$\Omega^0(A, a) :\equiv (A, a)$$
$$\Omega^{n+1}(A, a) :\equiv \Omega^n(\Omega(A, a)).$$

An element of it will be called an n-loop or an n-dimensional loop at a.

Annotation 7.22. Loop space 可以看做所有点 a 上 loops, 这样的 loops 可以用 $a =_A a$ 来表示，特别地有一个特殊的 loop 叫做 $\mathbf{refl}_a$.

Functions are Functors

Lemma 7.23. Suppose that $f : A \to B$ is a function. Then for any $x, y : A$ there is an operation

$$\mathbf{ap}_f : (x =_A y) \to (f(x) =_B f(y)).$$

Moreover, for each $x : A$ we have $\mathbf{ap}_f(\mathbf{refl}_x) \equiv \mathbf{refl}_{f(x)}$. We often write $f(p)$ as $\mathbf{ap}_f(p)$.

Annotation 7.24. $\mathbf{ap}_f$ 可以看做 the application of f to a path or the action on paths of f.

Lemma 7.25. For functions $f : A \to B$ and $g : B \to C$ and paths $p : x =_A y$ and $q : y =_A z$, we have

- Functoriality : $\mathbf{ap}_f(p \centerdot q) = \mathbf{ap}_f(p) \centerdot \mathbf{ap}_f(q)$.
- Reflectivity : $\mathbf{ap}_f(p^{-1}) = \mathbf{ap}_f(p)^{-1}$.
- Functor composition : $\mathbf{ap}_g(\mathbf{ap}_f(p)) = \mathbf{ap}_{g\circ f}(p)$.
- Identity functor : $\mathbf{ap}_{\mathbf{id}_A}(p) = p$.

Type Families are Fibrations

Annotation 7.26. 如果给定一个 dependent function $f : \Pi_{x:A} B(x)$ 和 $p : x = y$, 那么我们是否可以直接构造 $f(x) = f(y)$? 显然不行，因为 $B(x)$ 和 $B(y)$ 可能并不是同一个类型. 所以这里有一些 gap, 这些 gap 可以通过探究 p 本身蕴含的一些东西来解决.

Lemma 7.27. (Transport) Suppose that P is a type family over A and that $p : x =_A y$. Then there is a function $p_* : P(x) \to P(y)$. We often write $\mathbf{transport}^P(p, -) : P(x) \to P(y)$.

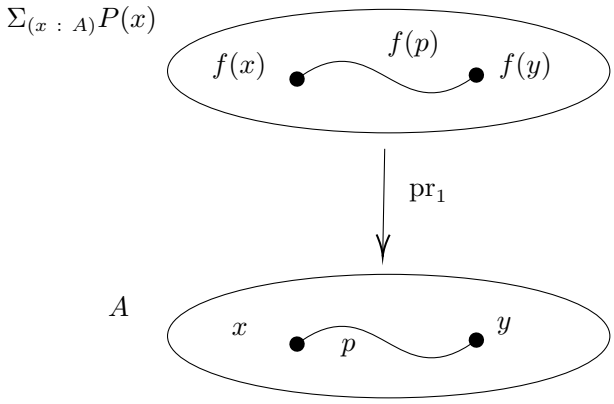
证明. The corresponding type is

$$f : \Pi_{P:A\to\mathcal{U}}\Pi_{x,y:A}(x = y) \to (P(x) \to P(y)).$$

By induction, it suffices to assume p is $\mathbf{refl}_x$, then $f(P, x, x, \mathbf{refl}_x) = \mathbf{refl}_{P(x)}$. □

证明. 这个 **transport** 和 indistinguiability of identicals 实际上一样的，都在说如果 $x = y$，那么 $P(x)$ 成立当且仅当 $P(y)$ 成立, 其中 P 可以看做关于 type A 的一个 property.

从拓扑的观点来看，可以将 $P : A \to \mathcal{U}$ 看做 a fibration with base space A, with P(x) being the fiber over x, and with $\Sigma_{x:A}P(x)$ being the total space of the fibration, with first projection $\Sigma_{x:A}P(x) \to A$. 如果存在 $u : P(x)$, 那么 path $p : x = y$ 就可以对应 total space 中的一个 path $(x, u) = (y, p_*(x)))$. 这个过程称为 path lifting.



□

Lemma 7.28. (Path lifting property). Let $P : A \to \mathcal{U}$ be a type family over A and assume we have $u : P(x)$ for some $x : A$. Then for any $p : x = y$, we have

$$\mathbf{lift}(u, p) : (x, u) = (y, p_*(x)).$$

in $\Sigma_{x:A}P(x)$, such that $\mathbf{pr}_1(\mathbf{lift}(u, p)) = p$.

Definition 7.29. Given a type family $P : A \to \mathcal{U}$, a dependent function $f : \Pi_{x:A}P(x)$ is a section of the fibration P. If such f is exists, we say there is fiberwise happened.

Annotation 7.30. 在给定一个 section $f : \Pi_{x:A} P(x)$ 的情况下, 我们可以定义一个 non-dependent function $f' : A \to \Sigma_{x:A} P(x)$ defined by $f'(x) :\equiv (x, f(x))$. 这样我们再把 **ap** 用在 f' 上了, 那么 $f'(p) : f'(x) = f'(y)$. 但是 type $f'(x) = f'(y)$ 并不能很好的显示这个 total space 上 path 是基于哪个 $p : x = y$ 的, 也就是我们希望 p 也出现最后我们得到的那个 equality type 中. 于是就有下面的定义.

Lemma 7.31. (Dependent map) Suppose $f : \Pi_{x:A} P(x)$, then we have a map

$$\mathbf{apd}_f : \Pi_{p:x=y}(p_*(f(x))) =_{P(y)} f(y))$$

证明. By induction, it suffices to assume p is $\mathbf{refl}_x$, then $\mathbf{apd}_f(\mathbf{refl}_x) = \mathbf{refl}_{f(x)}$. □

Annotation 7.32. 我们让 $u : P(x)$ 和 $v : P(y)$, 那么 $p_*(u) = v$ 就可以看做"a path form u to v lying over $p : x = y$".

Lemma 7.33. (Constantly transport) If $P : A \to \mathcal{U}$ is defined by $P(x) :\equiv B$ for a fixed $B : \mathcal{U}$, then for any $x, y : A$ and $p : x = y$ and $b : B$ we have a path

$$\mathbf{transportconst}_p^B(b) : \mathbf{transport}^P(p, b) = b.$$

Annotation 7.34. 有了 **transportconst** 之后, 对应任意的 $x, y : A$, $p : x = y$ 和 $f : A \to B$, 我们可以构造一个 function

$$\mathrm{eq}_1 : f(x) = f(y) \to p_*(f(x)) = f(y)$$

with definition equality

$$\mathrm{eq}_1(\mathbf{ap}_f(p)) = \mathbf{transportconst}_p^B(f(x)) \centerdot \mathbf{ap}_f(p).$$

反过来, 我们可以用 $\mathbf{transportconst}^{-1}$ 来构造 function

$$\mathrm{eq}_2 : p_*(f(x)) = f(y) \to f(x) = f(y)$$

with definition equality

$$\mathrm{eq}_2(\mathbf{apd}_f(p)) = \mathbf{transportconst}_p^{B^{-1}}(f(x)) \centerdot \mathbf{apd}_f(p).$$

接着我们可以证明 $\mathrm{eq}_1(\mathbf{ap}_f(p)) = \mathbf{apd}_f(p)$.

Homotopies and Equivalences

Definition 7.35. Let $f, g : \Pi_{x:A} P(x)$ be two sections of a type family $P : A \to \mathcal{U}$. A homotopy from f to g is a dependent function of type

$$(f \sim g) :\equiv \Pi_{x:A}(f(x) = g(x)).$$

Annotation 7.36. 一个 homotopy $f \sim g$ 可以从几个方面来理解:

- 可以看做是由很多 continous paths $f(x) = f(y)$ 构成的.
- 把 f, g 看做两个 functors, 那么 $f \sim g$ 是一个 natural transformation.

Lemma 7.37. Homotopy is an equivalent relation on each dependent function type $\Pi_{x:A} P(x)$.

Lemma 7.38. (Natural transformation) Suppose $H : f \sim g$ is a homotopy between dependent functions $f, g : \Pi_{x:A} P(x)$ and let $p : x = y$. Then we have

$$H(x) \centerdot g(p) = f(p) \centerdot H(y).$$

That is below diagram is commutative

$$\begin{array}{ccc} f(x) & \xlongequal{f(p)} & f(y) \\ {\scriptstyle H(x)}\Big\| & & \Big\|{\scriptstyle H(y)} \\ g(x) & \xlongequal[g(p)]{} & g(y) \end{array}$$

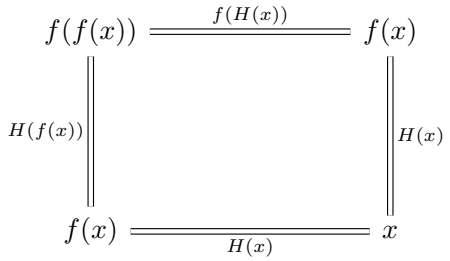
证明. By induction, it suffices to assume p is $\mathbf{refl}_x$, then $H(x) \centerdot \mathbf{refl}_{g(x)} = \mathbf{refl}_{f(x)} \centerdot H(x)$, which both sides are $H(x)$. □

Corollary 7.39. Let $H : f \sim \mathbf{id}_A$ be a homotopy, with $f : A \to A$. Then for any $x : A$ we have

$$H(f(x)) = f(H(x)).$$

Here $f(x)$ denotes the ordinary application of f to x, while $f(H(x))$ denotes $\mathbf{ap}_f(H(x))$.

证明. We know $H(x) : f(x) = x$, let p be $H(x)$. By naturality of H, we have following commutative diagram:



This is $f(H(x)) \centerdot H(x) = H(f(x)) \centerdot H(x)$. We can whisker by $H(x)^{-1}$ to cancel $H(x)$, obtaining

$$f(H(x)) = f(H(x)) \centerdot H(x) \centerdot H(x)^{-1} = H(f(x)) \centerdot H(x) \centerdot H(x)^{-1} = H(f(x)).$$

□

Definition 7.40. For a function $f : A \to B$, a quasi-inverse of f is a triple (g, α, β) consisting of a function $g : B \to A$ and homotopies $\alpha : f \circ g \sim \mathbf{id}_B$ and $\beta : g \circ f \sim \mathbf{id}_A$. We often use $\mathbf{qinv}(f)$ to denote the type:

$$\mathbf{qinv}(f) \equiv \Sigma_{g:B\to A}((f \circ g \sim \mathbf{id}_B) \times (g \circ f \sim \mathbf{id}_A)).$$

Example 7.41. For any $p : x =_A y$ and $z : A$, the function

$$(p \centerdot -) : (y =_A z) \to (x =_A z)$$

has a quasi-inverse given by $(p^{-1} \centerdot -) : (x =_A z) \to (y =_A z)$.

如何理解这个例子呢? 首先得把它们两个 definition 先给出来. 这里我们让 $\mathbf{concact} : \Pi_{A:\mathcal{U}}\Pi_{x,y,z:A}\ (x = y) \to (y = z) \to (x = z)$. 那么这里有

$$(p, -) :\equiv \mathbf{concact}(A, x, y, z, p)$$
$$(p^{-1}, -) :\equiv \mathbf{concact}(A, y, x, z, p^{-1})$$

那么接着来考虑它们的 composition

$$(p, -) \circ (p^{-1}, -)(r) \equiv \mathbf{concact}(A, x, y, z, p, \mathbf{concact}(A, y, x, z, p^{-1}, r))$$

其中 $r : x =_A z$. 再接下来需要证明

$$\mathbf{concact}(A, x, y, z, p, \mathbf{concact}(A, y, x, z, p^{-1}, r)) = r$$

通过 path induction, 我们证明存在一个 $\mathbf{refl}_{x=_A x}$ 对应上述类型.

参考文献

[1] Introduction to labelled transition systems.
http://wiki.di.uminho.pt/twiki/pub/Education/MFES1617/AC/AC1617-2-LTS.pdf

[2] An Introduction to Bisimulation and Coinduction.
https://homes.cs.washington.edu/~djg/msr_russia2012/sangiorgi.pdf

[3] Labelled transition systems.
https://www.mcrl2.org/web/user_manual/articles/lts.html

[4] A Calculus of Communicating Systems. Robin Milner.

[5] Baluda, Mauro, Giovanni Denaro, and Mauro Pezzè. "Bidirectional symbolic analysis for effective branch testing." IEEE Transactions on Software Engineering 42.5 (2015): 403-426

[6] https://www.logic.at/lvas/185255/ml-07-4in1.pdf