关于 Maple Algebra 的这一路

枫聆 (maplegra)

2022 年 11 月 22 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.

Misc

Definition 1.3. A set of logical connectives is called functionally complete if every boolean expression is equivalent to one involving only these connectives.

Definition 1.4. 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 1.5. 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

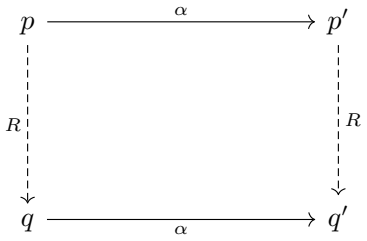
Definition 3.4. [1]Let $T = (S, \Lambda, \rightarrow)$ be a labelled transition system. The set of traces $Tr(s)$, for $s \in S$ is the minimal set satisfying

- $\varepsilon \in Tr(s)$.
- $\alpha\ \sigma \in Tr(s)$ 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

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

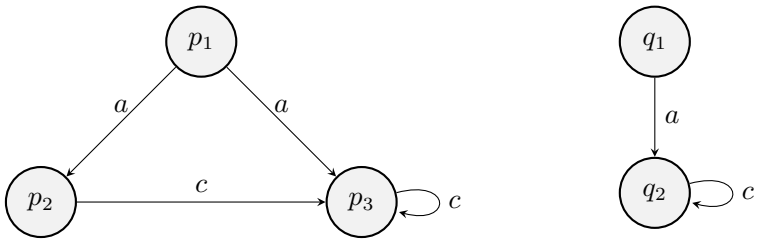


Definition 3.7. We say q simulates p if there exists a simulation R includes (p, q) (i.e., $(p, q) \in R$), written $p < q$.

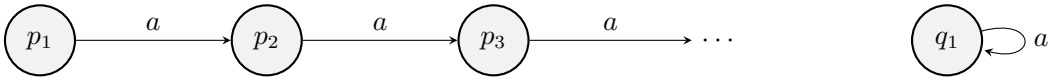
Definition 3.8. (Bisimultation) Given two labelled transition system $(S_1, \Lambda, \to_1)$ and $(S_2, \Lambda, \to_2)$, relation $R \subseteq S_1 \times S_2$ is a bisimulation iff both R and its converse $\overline{R}$ are simulations, for all $(p, q) \in R$ and $\alpha \in \Lambda$ satisfies

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

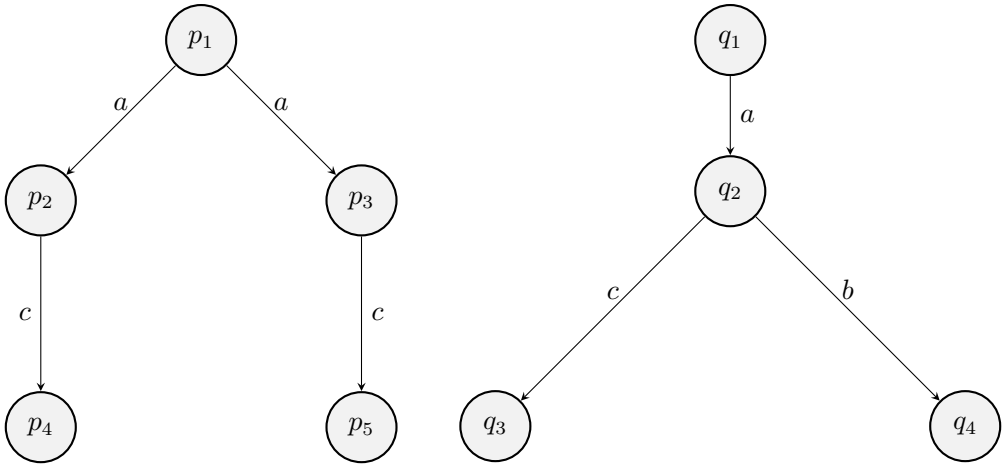
Example 3.9. 一些 bisimulation 的例子



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

Some Reasoning

Example 4.1. 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 4.2. 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 4.3. (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 4.4. 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 4.5. (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 4.6. 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 4.7. 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 4.8. 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 4.9. 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 4.10. 注意上面 variable 和 literal 的描述.

Theorem 4.11. (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 4.12. 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:

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

我们就把下面这个 resolution 往上移动，变成下面这样

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

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

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

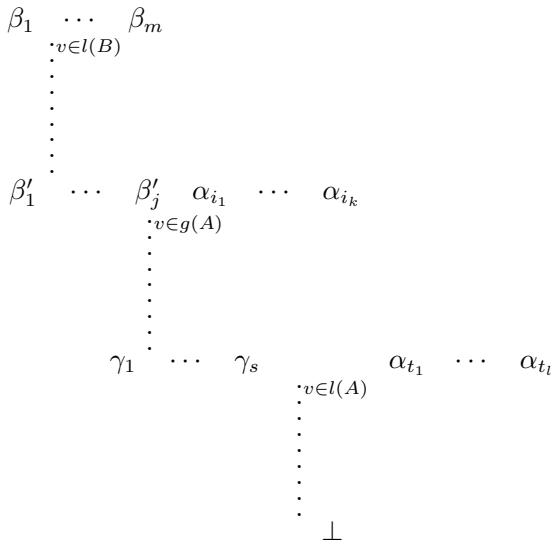
- $c_1 \in c_4$: $p_1'(c_1) = (b_1 \wedge b_3) \vee b_2$.
- $c_1 \in c_5$: $p_2'(c_2) = b_1 \vee (b_3 \vee b_2)$.

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 4.13. 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 4.14. (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.

Annotation 4.15. 路径条件通常形如 canonical form $A_1 \wedge A_2 \wedge \cdots \wedge A_n$. 当通过 SAT-slover 对其求解得到结果是 unsatisfiable, 意味这这样的路径是 unreachable 的. 于是我们可以把这样的路径排除在后续的分析之外, 排除在外的手法有很多:

- 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1中的 φ 为图

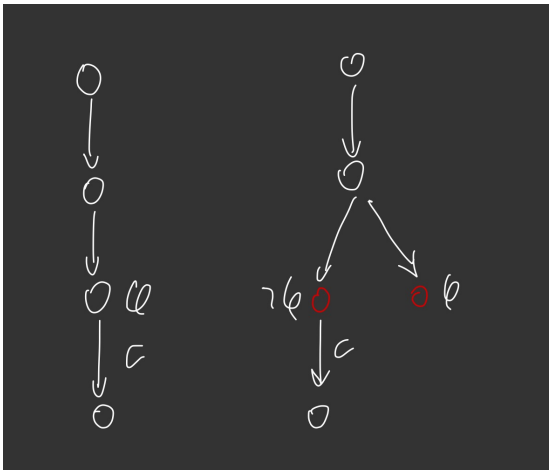


图 1: Refinement by node copying

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2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 $\neg\varphi_1 \wedge (\varphi_1 \vee \neg\varphi_2)$.

  其余过程类似, 直到从开始结点没有一条到 error 结点的路径为止. 这样看起来分析是非常 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需要的分析步骤是冗余的. 那么另外一种手法就是不对 CFG 图进行修改，直接在 CFG 上做 annotation, 这样做有两个好处:(1 很明显 CFG 不会扩张了; (2 annotation 可以 cover 不同路径但是” 相同状态”.

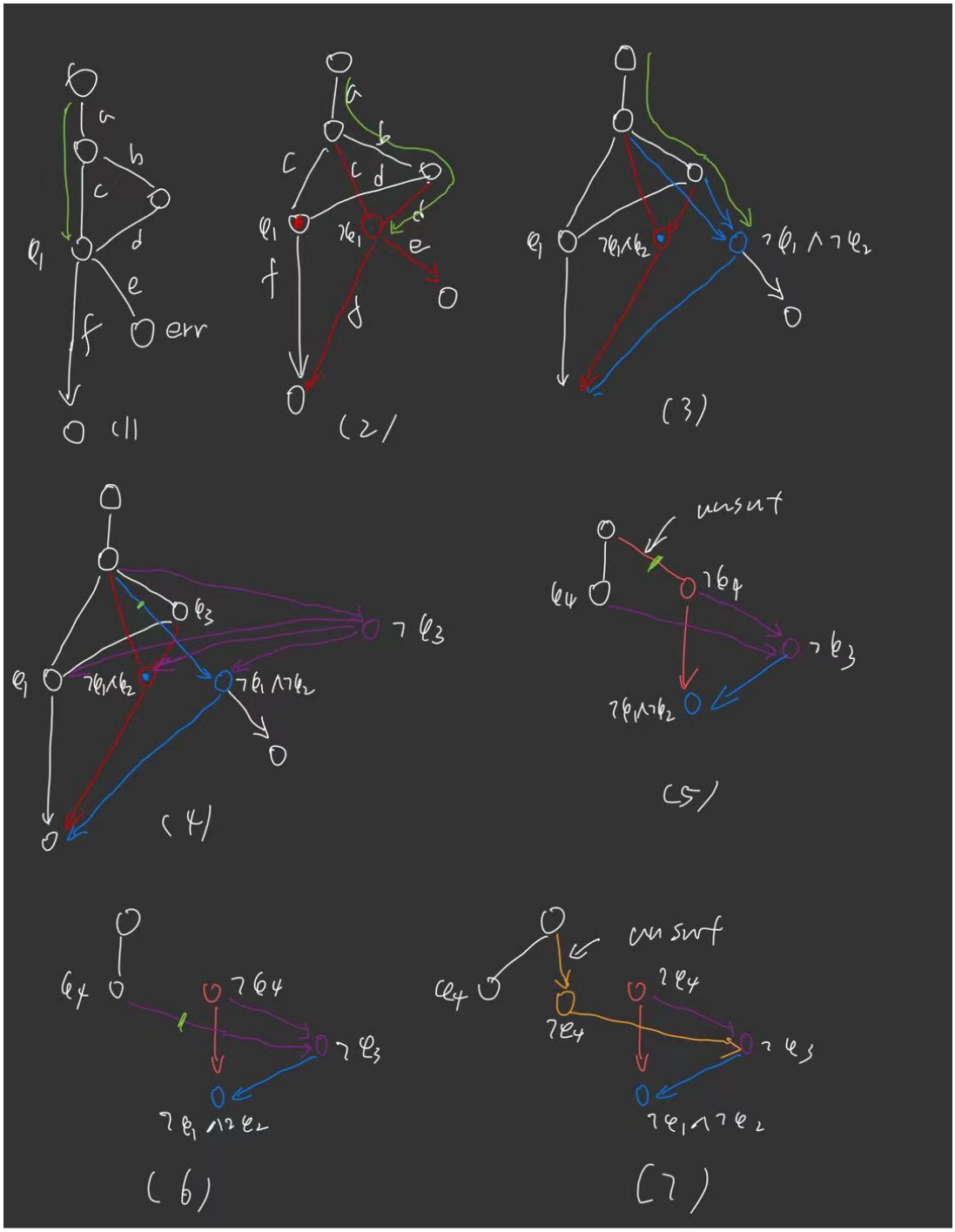


图 2: Refinement by node copying

Homotopy Type Theory

Universe and Families

Definition 5.1. A universe is type whose elements are types.

Annotation 5.2. 为了避免 Russell's paradox 在集合论中尴尬，可以用一种分级定义的 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 5.3. Given type A, the functions $B : A \to \mathcal{U}$ whose codomain is a universe are called families of type A.

Annotation 5.4. 理解 family 的关键是: 当 B 一个 function, 它接受一个类型为 A 的元素，输出是一个新的 type T, 它的类型为 $\mathcal{U}$. 这里最重要的说 B

Example 5.5. 设 $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$, 定义 $\mathrm{Fin}(n)$ 是一个 type, 符合这种 type 的元素是一个包含 n 个元素的集合，那么 Fin 就是关于描述 the types of finite sets 的的 family.

Example 5.6. 一个 constant type family 就是 type $B(x)$ 并不依赖某个 element with type A, 此时可以直接写作 $B : \mathcal{U}$, 因为 $\mathcal{U}$ 也是可以盖住 function types 的.

Dependent Function Types(Π-types)

Definition 5.7. 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 5.8. dependent funtion 和 type family 的区别是什么? 区别就是前者描述函数输出的是一个满足某个 type $B(x)$ 的元素，后者描述的是输出的是一个满足 $\mathcal{U}$ 的类型. 所以这也是这里为什么叫 dependent function type 的原因，它确实是一个描述函数的类型.

Example 5.9. 取5.5中的 family $\mathrm{Fin} : \mathcal{N} \to \mathcal{U}$, 我们可以构造一个 dependent function $\mathrm{fmax} : \Pi_{n:N} \mathrm{Fin}(n+1)$, fmax 返回值为非空集合中最大值. 如果我们用 $\{0_n, 1_n, \cdots, (n-1)_n\}$ 来表示 elements of $\mathrm{Fin}(n)$, 那么 $\mathrm{fmax}(n) :\equiv n_{(n+1)}$[1]

[1] :≡ 表示 function definition

Example 5.10. 当 family B 是 constant 的时候，有 $\Pi_{(x:A)}B(x) \equiv (A \to B)$[2]，即 original function type. 这说明 dependent function type 是可以完美盖住之前的 function type 的.

Example 5.11. 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 5.12. 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

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

Dependent Pair Types

Annotation 5.13. 这里有一个 uniqueness principle，我暂时不知道把它放在哪里，索性先放在这里. 我将它理解为: 要确定某个东西 A, 我们只需要看 A 是如何被使用的. 例如给定一个函数 f, 它可以等价于 $\lambda x.\, f(x)$, 这里其实就是用了一次 η-conversion. 这里就相当于是 f 被它对应的函数值唯一确定了. 这也是所谓的 extensionally (外延).

Annotation 5.14. 另外一个需要说明的是这里我们并没有规定 product type 的 inhabitant 就一定是一个 pair, 也就是说这个东西并没有作为规则写到 type theory 中. 相反我们可以引入一个 elimination rule: 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

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

这个 rule 是什么意思呢? f is well-defined when we specify its values on pairs. 我的理解这个 f 只是一个 partial function, 只给出了关于 pairs 情况下的定义.

Example 5.15. 按照上面的思路可以推出两个关于 product type 有两个 projection functions, 它们对应的 types 为:

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

根据前面的提到的 uniqueness principle, 我们通过确定 π_1 和 π_2 的对应函数值来唯一确定它们, 即

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

这里有一个小问题你在这里定义两个函数时候，你都得用一下 uniqueness principle. 我们试着想想这两个函数是如此的相似，我们能不能只用一次 uniqueness principle 就给出两个的定义呢? 这里可以很自然地提出 recursor.

[2] $\equiv$ 表示 definitional equality

Definition 5.16. 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 5.17. 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.

Definition 5.18. 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 5.19. 关于 recursor 的理解，在这里还是比较模糊.

Annotation 5.20. 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 $f : \Pi_{x:A\times B} C(x)$.

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

Example 5.21. 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，如何进一步说话 x 确实是一个 pair 呢? 还是有点小问题.

Annotation 5.22. *induction* 的由来

Definition 5.23. 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 5.24. 显然 $\mathrm{rec}_{A\times B}$ 只是 $\mathrm{ind}_{A\times B}$ 的一种特殊情况，即当 C 是一个 constant 的时候. induction 也可以被称之为 (dependent) eliminator, 而 recursion 可以被称之为 non-dependent eliminator.

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