第五章

我们究竟需要消耗多少计算资源（包括时间和空间资源）完成 ALC KB consistency 验证呢？依然是三种不同的情况的分析

（1） without TBox
（2） with acyclic TBox
（3） with general TBox

我们对 ALC KB consistency 问题进行归类，通过寻找他们所属的计算复杂度的类（complexity class）来确定这类问题的计算难度，（在这一章节中）这些类包括 PTIME、NP、PSPACE、EXPTIME、NEXPTIME。其中，N 开头的指的是使用 non-deterministic Turing Machine 为计算模型得到的，而其它是使用 deterministic Turing Machine 为计算模型得到的。在这里，我们有一个非常重要的假设，那就是这些类之间存在真包含关系：

PTIME 真包含于 NP 真包含于 PSPACE 真包含于 EXPTIME 真包含于 NEXPTIME

事实真的是如此么？并不是，比如 PTIME 与 NP 之间是否是真包含关系就不得而知（著名的 P=NP 问题），但相信大概率如此。而 PTIME 与 EXPTIME 之间以及 NP 与 NEXPTIME 之间真的存在真包含关系。有同学会问，time 和 space 都分别对应 TM 中的什么元素？time 指的是一台 TM 要执行多少 steps 才可以完成问题的计算，而 space 指的是一台 TM 需要多少的 tape cell 才可以完成问题的计算。TM 完成一次计算执行的 steps 数量不可能超过所使用的 cell 数量。这里注意，steps 与 TM 中的 states 没有一对一的关系，并不是执行一次 step 就跳转一次 state。到底什么是 step?

step：它对应的是 TM 完成一次完整的 transition 的过程：

（1） TM 首先 reads 当下的 tape cell 中的 symbol；
（2） TM 根据 transition function 的指导（这个 transition function 是基于当下的 state 和在第一步中 reads 的 symbol），TM 向当下的 tape cell 中 write 一个新的 symbol（如果不需要写入，则使用当下的 symbol 本身代替它自己），TM 向左或者向右移动 tape head，TM 将 transition 转移到一个新的 state。

完成上述所有过程后称为一个 step。

我们之所以无法将 step 与 state 进行一对一关联，是因为我们可以假设如下场景：在 step 1 中 TM reads 到了 symbol 0，根据 transition function f 的要求，也就是当下 state 1 和读到的 symbol 0 的要求继续执行下面的操作，但是 f 要求继续停留在 state 1，至此 step 1 结束，TM 来到了 step 2。在 step2 中 TM reads 到了 symbol 1，根据 f 的要求，TM 在完成读写操作之后转移到 state 2，至此 step 2 结束；以此类推，你会发现，step 的数量一直在单调增长，但是 state 完全可以回到之前的 state，tape head 也一直在读写操作中移动到之前到过的 cell 上。所以我们可以得到以下结论；

（1） 即便要求 TM 不可以回访之前到过的 state，TM 移动 n 个 states 所使用的 steps 数量是 n-1。但是实际情况是，TM 可以回访之前到过的 state，所以 TM 执行任意计算任务所使用的 steps 数量绝对不会低于 n-1，其中 n 代表 states 的数量；
（2） 同样的故事发生在 steps 与 tape cells 之间，TM 的 tape head 所遍历 n 个 tape cell 的数量最少是 n-1；
（3） states 与 tape cells 之间有无数量上的联系？答案是没有！

我们可以把 TM 中的 states 对应到现实计算机中处理器（processor）的逻辑单元（control logic），但是记住 state 从来不存储信息，是计算机硬件的一部分，是固定大小的；我们可以把 TM 中的 tape cells 对应到现实计算机中的内存（memory），其中每一个 cell 对应 memory 中每一个 unit。所以所以所以：判断一个计算任务所需要的时间复杂度指的是 TM 所执行的 steps 的数量的上限，而所需要的空间复杂度指的是 TM 执行上述 steps 的过程中所使用的的 tape cell 是的数量的上限。空间复杂度与 state 数量没关系，因为 states 数量在计算机设计的时候就已经固定了。states 多的计算机只是反映了该台计算机处理困难问题的潜力很大。这里我该如何给同学们讲清楚呢？这样吧，我们以下面这个例子来说明这些概念之间的关系：我们现在判断一个计算任务所需要的复杂度相当于判断一件物品的贵重程度。我想问一下，一件物品的贵重与否与购买这件物品的人的富有程度有关系么？一杯奶茶需要 10 元，会因为一个家财万贯的人购买就比 10 元贵，因为一个身无分文的人购买就比 10 元便宜么？答案是不会！states 是计算机本身的硬件功能，相当于购买人本身的财富程度。States 越多说明计算机处理复杂任务的能力越强，相当于一个人购买能力越强，但是与一件商品本身的价值没有任何关系。判断商品本身的价值还是要从需要花费多少金钱或者多少时间去得到它的角度来评判，也就是一次计算所需要执行的 steps 数量（时间复杂度）和消耗的 tape cells 的数量（空间复杂度）来判断，而不是由计算本本身的什么参数来判断。所以，一定会有这个问题本身大小的评判方法，比如 input string 的 length，比如一个 input tree 的 nodes 数量或者 outdegree 数量或者 depth 等等，这里如果用 n 来表示的话，然后出现一个函数，与 n 形成某种关系，比如线性、多项式、指数关系等等。

问题难度的上界（upper bound）和下界（lower bound）。当上述复杂度的类有着包含关系时，很明显这些类都是向下兼容的。如果一个计算问题被证明可以在指数时间内解决，那么它有没有可能在多项式时间内解决呢？当然有可能！可能问题本身并没有那么难，只不过你设计的算法太复杂了（相当于一个明明可以 10 元买下的奶茶你非要坚持用 20 元购买也不是不可以，但是用 5 元购买肯定是不可以）。

这里注意：问题 A 和问题 B 都被证明是 np 类里面的问题代表着二者的计算难度一模一样么？当然不是，只能说两个问题都可以在多项式时间内被 non-deterministic TM 解决，但有可能 A 需要用 n^2 时间解决，而 B 需要用 n^3 时间解决，而 np 类中存在一类最难的问题，其它 np 问题可以在多项式时间内规约为这些“最难的问题”。

我们首先确定一个计算问题在最坏情况下的计算上界在哪里（最难能难到哪里？）。这里，我们需要找到这个计算问题在哪个类里面，是 PTIME？还是 NP？还是其它类？做完这一步，我们还要证明这个问题的计算下界在哪里（最简单能简单到哪里？）。我们为此设计了新的术语，叫做 hardness 和 completeness。其中，如果说某问题 A 是 xx-hard 的，说明 A 的计算难度至少与 xx 这个类中那些最难的问题的计算难度相当。可能更难，但绝不会比它简单。例如，我们说 A 是 np-hard 的，则说明 A 这个问题的计算难度“至少”与 np 类中那些最难的问题的计算难度相当。但是 A 是 np-hard 的，不代表 A 在 np 这个类里面，因为 A 的计算难度只是“至少”相当于 np 类中最难的那些问题，很可能实际上比这些问题还要更难。但如果你证明了问题 A 的计算上界，发现它可以在多项式时间内使用 non-deterministic TM 解决，那么你就可以说 A 既在 np 类中，且属于 np 中最难的那类问题，那么 A 是 np-complete 的（如果在研究生面试中老师问你，解释一下什么是 np-complete 问题？可以回答：np 类里面最难的问题；解释一下什么是 PSPACE-complete 问题？可以回答：PSAPCE 类里面最难的问题）。同样的道理，如果证明问题 A 可以在多项式空间内使用 deterministic TM 解决，那

么 A 就是 PSPACE 类里面的;如果你证明问题 A 可以在多项式空间内使用 non-deterministic TM 解决,那么 A 就是 NPSPACE 类里面的。因为 Savitch's theorem 证明了 PSPACE 与 NPSPACE 是等价的类，则我们不再考虑 NPSPACE 这个类，统一用 PSPACE 代替。

如何证明一个问题 A 的计算上界是 xx? 你需要构建一个对应的 Turing Machine M 并证明问题 A 一定可以使用 M 在 xx 范围内的计算资源解决。

如何证明一个问题 A 的计算下界是 xx-hard? 你需要证明每个 xx 类里面的问题都可以在多项式时间范围内规约为（be reduced to）A 问题。但因为这样的问题是无穷的，我们将证明方法改为找到一个已经被证明是 xx 类里面最难的那类问题 B（已经被证明是 xx-hard 的问题 B），并证明 B 可以在多项式时间内规约为 A 问题。这里注意，你也可以寻找一个已经被证明是 xx-complete 的问题，因为 xx-complete 的问题也一定是 xx-hard 的问题。这里一定会有同学提问，我怎么知道哪些问题是 xx-complete 或者 xx-hard 问题？这确实需要一些计算理论的基础。通常每类问题都有一些“canonical problem”，使得一提到这个类，就能想到这个问题。比如典型的 PSPACE-complete 问题是 TQBF 问题。想证明一个问题 A 是 PSPACE-complete 的时候，需要证明 A 的计算下界是 xx-hard，第一想法就是想办法把 TQBF 问题在多项式时间内规约为 A 问题，此时，A 问题的计算下界就得到了。当然，如果你知道更多更直观的规约方法不一定非要使用所谓的 canonical problem。下面我们使用的就不是。

现在我们证明 ALC ABox consistency without TBox or with acyclic TBox 是 PSPACE-complete 的。很明显，证明 with acyclic TBox 的情况要比 without TBox 更难，我们证明 with acyclic TBox 的情况。之前讲过，验证 ALC ABox consistency with acyclic TBox 本质上就是验证 ALC concept satisfiability with acyclic TBox。这里我们可以将 TBox axioms 作为约束条件加入到验证过程。比如，[A 包含于 B]可以转化为[top 包含于~A or B]，也就是说[~A or B]是全集，等同于整个 domain。则在构建 model I 的过程中出现的任何 element 都应该属于$[\sim A \text{ or } B]^I$。而我们在 Theorem 3.24 里面明确了 ALC has the tree model property，意味着我们一定能够为一个 satisfiable 的 ALC concept 构建一个 tree model I。这个 model I 中的 domain element 都对应 tree 的一个 node。回忆一下，在构建 tree model I 的过程中，我们采用的策略是先构建一个 root node（根节点），然后每次只有在必要时才会创造一个新的 node（element），具体来说，只有每次遇到 exists 这个符号的时候才会创建一个新的 node，那么这个 tree model 的 outdegree 的数量上界是多少？一个 node 的 outdegree 是该 node 的子节点的数量，也就是从这个 node 外延出去的 edges 的数量；一个 tree model 的 outdegree 是所有这些子节点的数量的加和，也就是 tree 中所有 edges 的数量，这个数量的上界应该是输入的 concept C 和 TBox T 中 exists 的数量，也就是 C 和 T 中的 subconcepts 的数量，它的上界我们非常熟悉，是 C 和 T 的 size 之和。那么 tree model I 的 depth 的上界是多少呢？如果是 acyclic TBox 的情况，意味着不可能出现一个无限的定义 chain，对么？也就意味着不可能出现一个连续的 exists 的 chain 对么？那么这个树的 depth 的上界应该也是 C 和 T 中 exists 的数量,仔细想一想是不是？如果换成 general TBox 就不一定了,因为一旦出现了 cycles(环)，一个 exists 符号就可以被多次使用从而创造无限个 nodes。

再回忆一下 Theorem 3.16，tree model I 的 domain element 数量最多可以有 2^n，其中 $n = \text{size}(C) + \text{size}(T)$。

首先，对于给定的任意 acyclic TBox T，我们首先为其找到一个所谓的 normal form T'，使得后续的算法可以直接作用在这个 T'上，且我们得证明任意 acyclic TBox T 都可以在多项

式时间内转化成 normal form T'。这个 normal form 是我们自己设计的，你完全可以设计另一种 normal form 并证明任何 acyclic TBox 都可以在多项式时间内转化为这个 normal form。

算法的第一个要求：只包含 exact definitions（≡），不包含 primitive definitions（⊑）。
对应的步骤一：对于那些[A ⊑ C]的形式，我们将其转化成[A ≡ (A' ⊓ C)]的形式，这里的 A'是一个新引入的 concept name。转化前后的 T 和 T'并不是逻辑等价的，因为 T'中引入了新的符号 A'，但是二者是 conservative extension 的关系，T'是 T 的 conservative extension。具体来说，T'与 T 在[A, C]上语义等价：对于任意[A ⊑ C]的 model I，都存在一个 I 的 extension I'使得 I 与 I'的 domain 一致，且二者对于 A 和 C 的解释一致，只不过 I'会对 A'进行进一步解释；任意[A ≡ (A' ⊓ C)]的 model I'都是[A ⊑ C]的 model。则 C is satisfiable wrt T iff C is satisfiable wrt T'。

算法的第二个要求：算法只验证 concept name 是否针对于 acyclic TBox 是否 satisfiable，而对于 concept description 的情况不奏效。
对应的步骤二：这就意味着我们需要将 concept description 的情况转化为 concept name 的情况，方法是 A concept description C is satisfiable wrt T iff the concept name A is satisfiable wrt T ∪ {A ≡ C}。这个转化应该非常直观。转化后的 T ∪ {A ≡ C}是原始 T 的 conservative extension。

算法的第三个要求：所有的 acyclic TBox 都处于 negation normal form（NNF），这里的 NNF 与我们之前的定义有所不同，之前要求 negation 这个符号仅出现在 concept name 之前即可，这里我们要求 negation 符号仅仅出现在 primitive concept name 之前。那么什么是 primitive concept name 呢？在 acyclic TBox 中，如果一个 concept name 出现在 equivalence 的左边，被称为 defined name，“只”出现在 equivalence 的右边称为 primitive name。很明显，这个 NNF 的定义比之前见到的定义要更加严格。现在我们要证明使用 proposition 5.1 的转化方法，任意 acyclic TBox 都可以在多项式时间内转化为 NNF。转化方式如下：

（1） 对于 T 中出现的任意 defined concept name A 且 A ≡ C 属于 T，我们都引入一个对应的 $\underline{A}$，并且给 $\underline{A}$ 一个定义：$\underline{A}$ ≡~C。也就是让 $\underline{A}$ 与 A 互补。
（2） 将所有 equivalences 右边部分按照之前章节中介绍的方式转化为 NNF。
（3） 对于所有的 equivalences 中出现的~A，都是用 $\underline{A}$ 对其进行替换。

这样的得到的结果就是满足上面 NNF 定义的 acyclic TBox。当然，我们需要证明这样得到的 NNF 的正确性：A concept name A is satisfiable wrt T iff A is satisfiable wrt T'in NNF。

算法的第四个要求：所有已经处于 NNF 的 acyclic TBox T 都必须转化为 simple form，其目的是避免 exists 或 forall 发生嵌套。首先看什么是 simple form：

$$A \equiv P,\ A \equiv \neg P,\ A \equiv B_1 \sqcap B_2,\ A \equiv B_1 \sqcup B_2,\ A \equiv \exists r.B_1,\ \text{or}\ A \equiv \forall r.B_1,$$

where P is a primitive concept and B_1, B_2 are defined concept names.

Equivalences 的左边只能是 defined concept name 没问题，但是右边必须满足上述要求。即要不右边是 primitive concept name P，要不是其 negation 形式~P，要不是 ALC 的其它四个基本语法形式：and、or、exists、forall，但这些符号只能连接 defined concept names。首先观察到，一个 simple TBox 必然是 NNF 形式，意味着从 NNF 转到 simple 不会转回 non-NNF

形式。其次，我们要证明任意 acyclic TBox T 都可以在多项式时间内转化为 simple form（另一种说法是任意处于 NNF 的 acyclic TBox T 都可以在多项式时间内转化为 simple form，这两种说法没区别，因为任意 acyclic TBox T 都可以转化为 NNF）。同学们这门课学习到这里，应该特别了解我们 KR 中的一些基本操作：凡是范式转化，就必须证明多项式时间内的可转化性。转化方法在 Lemma 5.2 中也已经写明（这个转化是建立在 T 已经是 NNF 基础上）：

（1） 如果遇到 $A \equiv C \sqcap D$ 的形式，并且发现 C 或 D 并不是 defined concept name，我们引入 B1 和 B2 两个 concept name 来代替 C 和 D，得到 $A = B1 \sqcap B2$，且让 $B1 \equiv C$，$B2 \equiv D$，这样一来，B1 和 B2 就成了 defined concept names，$A \equiv B1 \sqcap B2$ 满足了 simple form 的形式要求，且依然保证正确性（见 Lemma 5.2 中 such that 后面的话）。对于其它几个不满足 simple 的形式，也是同样的处理方法。

（2） 如果遇到 $A \equiv B$ 且 B 是一个 defined concept name（simple form 要求这种情况下 B 必须是一个 primitive concept name），也就是说一个 defined concept name 被另一个 defined concept name 定义，我们用 B 代替 T 中全部的 A，或者用 A 代替 T 中全部的 B。前者发生在 $A \neq A0$ 时，后者发生在 $A = A0$ 时。关于 A0 是什么，见 Lemma 5.2。

我们接下来要介绍的 algorithm 就是建立在 simple acyclic TBox 基础上的。

【定义 5.3】让 T 表示一个 simple TBox。让 Def(T)表示 T 中所有出现的 defined concept name。我们引入一个称为 type 的术语。T 的一个 type τ 被定义为 Def(T)的子集，且满足：

（1） 如果发现 $A \equiv P$ 且 $B \equiv \neg P$ 同时出现在 T 中，则 A 属于 τ 推导出 B 不属于 τ；

（2） 如果发现 $A \equiv B \sqcap B'$ 出现在 T 中，则 A 属于 τ 推导出 B 属于 τ 且 B'属于 τ；

（3） 如果发现 $A \equiv B \sqcup B'$ 出现在 T 中，则 A 属于 τ 推导出 B 属于 τ 或 B'属于 τ；

接下来我们使用 Fig 5.1 中的算法来构建一个 tree model I，这个 model 的特点是 tree 的深度（depth）的上界为需要被验证 SAT 的那个输入的 concept name A0 的 role depth rd(A)。也就是说 A0 到底被多少 exists 和 forall 嵌套着。关于 A 的 role depth 的定义如下：

（1） 如果 $A \equiv P$ 或 $A \equiv \neg P$ 出现在 T 中，则 A 的 role depth $rd(A) = 0$；

（2） 如果 $A \equiv B$ and B'或者 $A \equiv B$ or B'出现在 T 中，则 $rd(A) = \max(rd(B1), rd(B2))$；

（3） 如果 $A \equiv \exists r.B$ 或者 $A \equiv \forall r.B$ 出现在 T 中，则 $rd(A) = rd(B) + 1$；

根据这个定义，只有∃和∀才可以让 role depth +1。其实，本质上就是在计算 A 的嵌套层级是多少，如果 A 被∃或∀嵌套，则我们继续递归去算下去。我们引入一个新的符号 $Def_i(T)$，表示 T 中 role depth 小于 i 的那些 defined concept name，所以 $Def_i(T)$ 是 Def(T)的子集。

接下来我们看一下 Fig 5.1 中的算法，名称为 ALC-Worlds，参数为 A0 和 T（验证 concept name A0 是否针对于 simple TBox T 为可满足的）。i 被初始化为 A0 的 role depth，我们使用 non-deterministic TM 猜一个 type τ，且这个 type τ 必须是 $Def_i(T)$ 的子集且 A0 必须出现在 τ 中。这句话什么意思？$Def_i(T)$ 让 defined concept names 限制在 role depth 小于等于 A0 的那些 defined concept names 上，role depth 大于 A0 的我们不考虑。为什么不考虑？然后我们调用一个称为 recurse 的函数，其参数为 (τ, i, T)。首先遇到一个跳出该函数的 if 判断，这个判断说当 τ 不是一个 type 时，返回 false。这句话什么意思？要想搞清楚这句话的意思我们首先得搞清楚 type 的内涵是什么？type τ 里面是一群 defined concept names，这些 names 最大的特点是，如果一个 domain element d 属于其中一个 name，则也一定属于其他的 names。换句话说，这些 defined concept names 属于同一个派系。派系怎么形成的？比如因为几个人有共同的爱好，有共同的课表，住同一个的宿舍等等，总之有一些共同的东西

让派系中的成员自觉走到了一起，形成了 type。这里也是一样，如果 element d 出现在了 A 中且 A≡B ⊓ B'出现在了 T 中，则 d 一定也出现在了 B 中和 B'中。那么什么时候不是 type 呢？当 τ 中混进了“外人”的时候或者本属于 τ 的成员没有被纳入进来的时候。比如 A≡P 且 B≡¬P 出现在 T 中，但是 τ 中却同时出现了 A 和 B；再比如 A≡B ⊓ B'出现在 T 中，但是 τ 中却只有 A 没有 B...回到算法本身，伪代码按照顺序读下去，if＝0 这一句读不懂，先不管他，继续读下去。我们发现对于 type τ 中所有的 name A 来说，如果发现 A≡∃r.B 出现在 T 中，则我们创建一个 S 集合，这个集合里面装入 B。请记住，这里的 B 只是一个代指的符号，这样的 B 可能有多个，因为上文说 type τ 中所有的满足条件的 A，所以 A 也是一个代指的符号。除了 B，我们还要放入 B'，B'满足的特点与∀符号有关：当发现 type τ 中存在符号 A'且 A'≡∀r.B'出现在了 T 中，则 B'要被纳入到 S 中。这个 S 是什么？首先它纳入的都是 defined concept names 且他们的 role depth 比当下的 i 小 1。

我们将这个计算过程与 recursion tree 的计算过程对应起来，recursion tree 的作用是对一个包含递归的算法进行结构化表示。每一个 tree 的节点对应一次递归 call，并将该节点 label 上该 call 的参数，分别是（τ，i，T)。则 tree 中连接两个节点 v 和 v' 的 edge 表示 v'这个 call 出现在了 v 这个 call 中。则这个 recursion tree 的 depth 上界是 $rd(A_0)$，这是因为 i 被初始化为 $rd(A_0)$，且每一次的递归 call 都会让 i 减 1，而 $rd(A_0)$的上界是 T 的 size。tree 的 outdegree 的上界，也就是 call 的总数量，应该是 exists 的数量，上界依然是 T 的 size。这就保证了构建树这个过程的可终止性：既不会构建无限深度的树，也不会构建无限广度的树。因为每一个 call 需要存储的数据的大小与其参数的关系是多项式空间关系，即

接下来我们需要证明，ALC-Worlds 算法的正确性和完备性。很多同学一直有个疑问，我们什么时候证明这些性质，怎么总是需要证明算法的正确性和完备性？答案是：如果不证明这个算法的正确性，你怎么知道这个算法给出 true 的结果，A_0 针对于 T 一定是 satisfiable 的？如果不证明这个算法的完备性，你怎么知道这个算法对于每个[A_0 针对于 T 是 satisfiable]的问题都能给出 true 的结果？如果这个不能被证明，你刚刚分析的 ALC-worlds 算法的复杂度与 ALC ABox consistency without TBox or with acyclic TBox 或者 ALC concept satisfiability with acyclic TBox 问题的计算复杂度有什么关系？一个问题 A 的计算复杂度是 xx，需要找个 xx 的算法 B 来解决 A。如果我们都不能证明算法 B 可以解决问题 A，凭什么说 B 的复杂度就是 A 的复杂度呢？

我们先对 Lemma 5.4 证明其 only if 方向，即：如果 ALC Worlds 返回 true，则 A0 针对 T 是 satisfiable 的。让 T=（V, E, l）是 A0 和 T 作为 ALC Worlds 算法参数生成的 recursion tree，其中 v0 是根节点。让 gamma(v)表示所有非根节点 for all 代码段中的产生的 role name。我们让那个被构建的 tree model 用 I 表示，则让 $V = \Delta^I$。

接下来我们证明问题的 lower bound。我们选择的被证明为 PSPACE-hard 同时也是 PSPACE-complete 的问题称为 finite Boolean game，简称 FBG。问题也非常简单，就是有两个玩家，Player 1 和 Player 2，他们分别控制着 proposition logic 表达式的变量 p 且各自控制一半，其中 Player 1 控制奇数系，Player 2 控制偶数系。Player 1 与 Player 轮流对这些变量 p1 p2...进行赋值。问题则可以简单描述为：Player 1 是否有一个赢家策略，使得无论 Player 2 如何赋值，整个表达式都为真？这里的理解最好通过参考书 P114 中的例子。我们的任务是将此问题在多项式时间内转化为 ALC concept satisfiability 的问题：让 G 代表这个游戏，Player 1 有一个 G 的赢家策略，当且仅当，ALC concept C_G 是 satisfiable 的。

显然，我们需要建立两个问题之间的一对一联系，这个直观上是看不出来的。这个联系的建立产生的结果是：每一个 C_G 的 model 都对应一个 Player 1 在 G 中的 winning strategy，反过来，每一个 Player 1 在 G 中的 winning strategy 都对应 C_G 的一个 model。我们用 role name r 指代 strategy tree 中的 relation，我们用 concept names P1...Pn 指代每一个 propositional variable 的真值，这样一来，就构建了 proposition logic 符号与 description logic 符号之间的对应关系。接下来，我们一步一步的构建这个 C_G，我们的目的是让 C_G 确实能 model 这个 G 问题，成为其正确且完整的约束条件。C_G 是一个用 conjunction 连接的 concept，每个 conjunct 是其中一个条件，只有每一个 conjunct 都能被满足，整个 concept 才可以被满足。这个构建的过程是一步一步构建其 conjunct，并对其进行解释：

（1） 第一个 conjunct C1 的构建：对于每一个奇数层的 node 来说，这个 node 本身是由 Player 1 赋值的，接下来将由 Player 2 进行赋值，所以接下来有两种赋值可能性，一种将 p_{i+1} 赋值为 1，另一种赋值为 0。既然刚才说过，P 作为 concept name 来指代 propositional variable 的真值，那么 Pi 可以指代第 i 个 p 的为 1，~Pi 指代第 i 个 p 为 0。则 C1 指代在经历了 i 个 r 关系后（node 停留在树的奇数层），此时 i 为奇数，则我们在经历任意奇数个 r 关系后接下来赋值应该是将树分为二叉树，两种赋值同时进行，所以中间连接的符号是 and，而不是 or。Exists r.1 and exists r.0 表示至少存在两个 r 关系，一个后面的 propositional variable 赋值为 1，另一个赋值为 0。这里同学们可以理解为什么是 and 么？如果改成 or 的话，“至少存在两个 r 关系”的事实就不能被 model 出来了，而是变成了存在一个 r 关系之后的 variable 赋值为 1 或者存在一个 r 关系之后的 variable 赋值为 0；

（2） 第二个 conjunct C2 的构建：对于每一个偶数层的 node 来说，这个 node 本身是由 Player 2 赋值的，接下来将由 Player 1 进行赋值，但是接下来 Player 1 赋值为 1 或者 0 并不是由它自己决定的。因为一旦赋值错误，会导致后面 Player 2 的赋值有机会让整个 formula 为 0。所以其实 Player 1 的赋值是一种“必然的选择”或者叫“无奈的选择”。

Lemma 5.6 是 FBG 与 concept satisfiability 问题相互转化正确性的证据，即我们需要证明（if 方向）当 C_G 可满足的时候，Player 1 一定对于 Game G 有一个获胜策略；反过来（only if 方向）当 Player 1 在 Game G 中有一个获胜策略的时候，C_G 一定可满足。

第六章

我们在之前看到了，即便是在 without TBox 或 with acyclic TBox 的情况下，推理的复杂度也达到了 PSPACE-complete，而 with general TBox，推理复杂度达到了 EXPTIME-complete。这一章我们介绍一种新的 DL 语言，称为 EL，它的推理计算复杂度只是 PTIME-complete，但是可以预见，表达力比起 ALC 肯定会有损失。确切地说，比起 ALC，EL 只允许⊓和∃作为逻辑构造符出现在语言中，也只允许 top concept 不允许 bottom concept 出现在语言中。以前的时候，要验证 C in D 成立，我们需要做一个假设，假设存在一个 domain element d 属于 C⊓¬D，如果这件事情成立，则说明 C in D 不成立，否则成立。我们把这种反驳性的算法称为 Tableau。因为⊔的存在，算法不得不对其每个 branch 都进行验证，所以这里我们需要一个 non-deterministic 的 TM 来帮助我们进行验证。现在我们换一种思路，可不可以不再使用反驳性质的验证算法，而是根据已有的 axioms，从中推导出新的 axioms，也就是从显性的 axioms（物理上包含在 TBox 中的）推导出隐性的 axioms（物理上不包含在 TBox 中的

但能够被 TBox 语义蕴涵的)，并且保证所有的隐性 axioms 都被推导出来。这样一来，所有的 axioms，也就是所有的 logical consequences 都存在在 TBox 中（称为饱和，saturated)，想要验证 C in D 是否成立，只需要查看它是否存在在 TBox 中即可。我们把这种类型的推理方式称为 consequence-based reasoning，把 Tableau 的推理方式称为 refutational reasoning。既然我们在之前说过 EL 的验证方法是 PTIME 的，说明这样的隐性 axioms 最多产生多项式多个，否则空间复杂度超过多项式，时间复杂度只会更加糟糕。

说到这里，有同学可能会生出疑问：Tableau 算法可以在多个推理任务中通过互相转化，比如 subsumption 问题转化为 consistency 问题，转化为 concept satisfiability 问题来得到结果，EL 的这个算法怎么验证 TBox 或者 KB 的 consistency，或者 EL concept 的 satisfiability 呢？consequence-based reasoning 好像只可以作 subsumption 验证，也确实如此。但是令人惊讶的事，EL TBox 或者 EL KB 都是 consistent 的，EL concept 都是 satisfiable 的，这一结论可以通过构造通用的 model，或者叫 canonical model 来实现。比如，让 domain 包含元素 a，让所有的 concept name 都被解释为{a}，让所有的 role name 都被解释为{(a,a)}。则所有的 EL TBox/KB 都有 model，所有的 EL concept 都可以不为空。所以 EL 算法只需要负责验证 subsumption 即可。

这一章节介绍的 EL 算法只负责验证两个 concept names 之间是否存在 subsumption 关系，看起来算法失去了完备性，其实不然。任意两个 concept descriptions 之间的 subsumption 验证问题也可以再多项式时间内规约为两个 concept names 之间的 subsumption 验证问题。对于两个 concept descriptions C 和 D 之间的验证，以前我们的规约方法是引入两个新的 concept names A 和 B，并且让 A≡C，让 B≡D，然后我们验证 A 与 B 之间是否存在 subsumption 的关系。这样一来我们可以让得到的 T′与 T 形成在 sig(T)上的语义等价关系，即除了 A 和 B，T′与 T 对于任意 sig(T)中的 names 的解释都一致。所以如果 T 语义蕴涵 C in D，则 T′也一定语义蕴涵 C in D，反之也是一样。T′是 T 的 model conservative extension。但是如果仅仅是验证 C 和 D 之间是否存在 subsumption 关系，我们无需依赖上面的 model conservative extension 这么强的关系，即对于除了 A 和 B 之外的任意 names 的 models 都一致。我们只需要保证 C in D 这个逻辑结果一致即可。注意，models 一致一定会让 subsumption 关系一致，但是 subsumption 关系一致不一定保证 models 一致。这里我们介绍一种 weaker 的转换形式，它能保证如果 T 语义蕴涵 C in D，则 T′也一定语义蕴涵 C in D，反之也是一样，但不保证 T′和 T 在 sig(T)上的 models 一致，我们称其为 deductive conservative extension 或者 consequence-based conservative extension。这种转化方法也同样引入两个新的 concept names A 和 B，不同的是这次我们想 T 中添加的是 A in C 和 D in B 两个 inclusions，而不是之前的 equivalences。尽管很明显可以看到 T 与 T′之间在 sig(T)上并不语义等价，但是在保留 C in D subsumption 关系的等价性上完全一致，这就够了。请阅读 Lemma 6.1 来理解该转化方法在保证 C in D 关系一致的正确性。通过这种方式，任意两个 concepts 之间的验证都可以被转化为两个 concept names 之间的 subsumption 验证。

接下来，算法要求给定的 TBox 要处于 normal form 的状态，并因此介绍了一种新的 normal form，这个 form 要求要不 inclusion 左右两边都是 concept names，要不左边是 and 连接符，要不左右两边可以出现 exists 符号。这样的 normal form 涵盖了所有 EL 可能出现在的基本形式，比如我们为什么不在 inclusion 右边出现 and 连接符，这是因为它可以继续化简为其它形式：A in (C and D)可以等价转化为 A in C 和 A in D。看到这里，可否想一下为什么我们每次介绍一个算法，这个算法都要求它的输入先处于某种 normal form，才能继续工作呢？这是因为算法基于的规则，我们设计的规则不可能适配所有的表达式形式，因为表达式

是无穷的，比如可以有 A≡∃r.B，可以有 A≡∃r.∃r.B，可以有 A≡∃r.∃r.∃r.B，我们的算法不可能为每一种不同的语法形式都设计一组规则。但我们可以做到找到一种通用的形式，所谓 canonical form，甚至这种 canonical form 都可以不唯一。一旦确定了 normal form，接下来需要做三件事：（1）设计一组规则用于将原始的输入转化为 normal form，并证明对于任意的输入都可以使用这组规则在多项式时间内将其转化为 normal form；（2）证明这个转化过程是正确的。这里的正确性很模糊，取决于最终任务是什么。例如，如果任务是验证 C in D 是否存在 subsumption 关系，那么正确性就是证明转化之前与转化之后在 C in D 这件事上的结果一致；如果任务是验证 C 和 a 之间是否存在 membership 的关系，那么正确性就是证明转化之前与转化之后在 C(a)这件事上的结果一致；（3）为这种 normal form 设计对应的算法，并证明算法的可终止性、正确性、和完备性。

这一次，上面（1）中的转化规则在参考书上已经给出在 Figure 6.1 里面，这些 rules 都是针对 normal form 设计，且可以按照任意顺序使用。Lemma 6.2 首先证明了使用这些规则进行 normalization 的计算复杂度：可以使用 linear number of applications of the rules，说明只需要线性次数的 steps 就可以完成这个任务，得到的新的 T′大小针对于原始 T 的大小也只是线性增长。这个 Lemma 的证明相对比较简单，不过多赘述。最重要的是下面证明上述（2）：整个转化过程的正确性。可能有同学想证明转化前后两个 TBoxes 是语义等价的，这个我们说过了，不可行。因为很明显新的 TBox 中引入了新的 names，这导致满足原始 TBox T 的 models 很多都不再满足新的 TBox T′，因为原始的 models 很可能并没有对新引入的 names 进行解释，但他们依然可以满足 T，因为 T 中并没有这些新的 names。我们得需要引入一种新的等价概念，这种等价概念不要求转化前后的两个 TBoxes 语义等价，只要求他们在“某些方面”等价，可以理解为“参数化（parameterized）”的等价关系。我们这里引入一种叫做 conservative extension 的等价关系，其实在前面已经见到过了，确切地说，这里引入的叫 model conservative extension。它的内涵是对于转化前后的两个 TBoxes，要求他们在某些 names 上语义等价。这里的某些 names 是他们的 common names，也就是原始 TBox T 中的 names，因为 T′是 T 的 names 上的直接扩展。所谓“某些”不仅仅可以是 common names，还可以是任意其他参数，比如 common names 的子集都可以，总是就是让 T 和 T′在某些参数上保持等价关系。关于 model conservative extension 的内涵有三个性质需要满足，也就是 Definition 6.3 中的三条性质。前两条非常直观，第三条要求对于任意 T 的 model I，都存在一个“对应的”的 model J 使得 I 与 J 在 sig(T)上的符号解释完全一致，在 T′中额外那些 names 的解释可以不一致（但是你得找到一种通用的解释方式，使得这些额外的 names 可以被解释）。仅仅证明 T′是 T 的 model conservative extension 就可以了么？当然不是，这里 model conservative extension 仅仅是工具人，是我们证明 normalization 正确性的中间过程。我们最终要证明的是对于任意使用 sig(T)中的 names 构建的 concepts C 和 D，如果 T 语义蕴涵 C in D，则 normalization 之后得到的 T′也语义蕴涵 C in D，反过来也是如此（其实这里不一定是 iff 关系，只需要证明单方向即可，想一想是不是？我们只需要证明如果 T 语义蕴涵 C in D，则 normalization 之后的 T′也语义蕴涵 C in D 即可）。Lemma 6.4 给出了上述 iff 的证明，将[model conservative extension]与[normalization 的正确性]巧妙地联系到了一起，即：只要保证 T′是 T 的 model conservative extension，则任意使用 sig(T)中的 names 构建的 concepts C 和 D，如果 T 语义蕴涵 C in D，则 normalization 之后的得到的 T′也语义蕴涵 C in D，反过来也是如此。很明显这里面暗示着一件事：model conservative extension 一定能保证 subsumption 一致，也就是 logical consequences 一致。是保证 normalization 正确性的充分条件，但不一定是必要条件。或许存在一个比 model conservative extension 更弱的条件即可保证 subsumption 的一致性（也就是作业里的 deductive conservative extension）。

Lemma 6.4 的证明也相对简单，它的思想就是如果想证明 p implies q，则我证明~q implies ~p。这里是想证明 T1 entails C in D if T2 entails C in D，则可以证明 T2 does not entail C in D if T1 does not entail C in D。反过来也是一样：想证明 T1 entails C in D only if T2 entails C in D，则可以证明 T1 does not entail C in D if T2 does not entail C in D。

证明到目前为止，同学们有没有觉得少了点什么？我们证明了 model conservative extension 可以保证 subsumption 的一致性（normalization 的正确性），但是却从未证明使用这些规则得到的结果 T'是 T 的 model conservative extension。如果这一步不证明，那我们上面的证明不就全都白忙活了么？所以 Proposition 6.5 给出了这个证明：使用 Figure 6.1 中规则得到的 TBox T2 一定是原始 T1 的 model conservative extension，则 T2 自动继承 Lemma 6.4 的性质，也就是 subsumption 一致性得证，normalization 的正确性自此有了保证。这里的证明有点难度，其中的难点在于构建一个通用的 T2 的 model，所谓通用，就是对 T2 中新引入的 names 有一个通用的解释方法，我们需要找到这个解释方法。最后，Corollary 6.6 给出了 normalization 规则的正确性，其实已经不必要了，因为 Corollary 6.6 就是前面 Lemma 6.4 和 Proposition 6.5 的直接结果。当然写出来看着更加直观。

一切准备就绪之后，我们就可以进入到真正的 classification 环节，也就是对处于 normal form 的 TBox T 进行 classification。所谓 classification 就是找到其中 concept names 之间的上下位关系。Classification 规则提供在了 Figure 6.2 里面。接下来什么操作同学们应该已经猜到一二：我们要证明 classification 过程的可终止性（甚至更确切的计算复杂度）、正确性、完备性。Lemma 6.9 首先给出了计算可终止性以及计算上界。Lemma 6.10 给出了正确性证明。这里的问题是，正确性的内涵是什么？这里指的是每个规则产生的结果都应该是原始前提的逻辑结果，换句话说，原始的前提都能语义蕴涵规则产生的结果。否则，那不是相当于规则产生的错误的结果，其实它并不是蕴含在 T 中的隐性知识，但却被错误的放入了现在的 T，那我们最终得到的 T'就不是 T 的逻辑结果了。Definition 6.11 给出了从 T 中通过规则得到的 T*的一个 canonical interpretation 的构建方法。Lemma 6.12 则进一步证明了这个 canonical interpretation 是 T*的 model。最后我们证明 classification 算法的完备性：对于任意通过 classification 规则的使用得到的 T*，如果 T entails A in B，则 A in B 一定出现在 T*中。最后 Theorem 6.14 给出了整个 subsumption in EL 的计算复杂度，是上述过程的分步总结。这里同学们最大的难点我觉得不在于证明本身，而是对于方法正确性和完备性的理解。我们至今已经讲了很多算法，Tableau 算法、ALC-worlds 算法、EL subsumption checking 算法等等，每个都有正确性和完备性证明，但却每个内涵都不相同。你能通过这么久的学习理解到底什么是正确性？什么是完备性么？有的时候你会发现完备性指的是：这个方法可以处理这种问题。但是刚刚你发现完备性指的是：所有能被语义蕴涵的 subsumption 都一定显性地存在在了 T*之中。是不是发现完备性好像不太一样？我们把前者称为外部完备性，称后者为内部完备性。