**LỜI CAM ĐOAN**

Chúng tôi xin cam đoan rằng ngoại trừ các kết quả tham khảo từ các công trình khác như đã ghi trong luận văn, các kết quả trình bày trong luận văn này là do chính chúng tôi thực hiện và chưa có phần nội dung nào trong luận văn này được nộp để lấy bằng cấp ở bất cứ trường nào.

Nguyễn Ngọc Phước – Nguyễn Duy Nhất.

**LỜI CẢM ƠN**

Chúng tôi xin gửi lời cám ơn đến thầy Th.S. Nguyễn Thanh Sơn đã nhiệt tình hướng dẫn chúng tôi trong quá trình thực hiện luận văn. Thầy đã gợi ý, giúp đỡ cũng như bù đắp những chỗ kiến thức còn thiếu hụt của chúng tôi, tạo cho chúng tôi một tác phong làm việc rất chuyên nghiệp.

Chúng tôi xin gửi lời cảm ơn tới cha mẹ, những người sinh thành và dưỡng dục chúng tôi, những người luôn ủng hộ chúng tôi cả về vật chất lẫn tinh thần, giúp chúng tôi vượt qua những khó khăn trong suốt thời gian qua.

Chúng tôi cũng gửi lời cám ơn tới những người anh, người chị khoá trước, cùng những người bạn thân thiết đã luôn bên chúng tôi động viên, giúp đỡ chúng tôi rất nhiều trong suốt quá trình từ Thực tập đến lúc hoàn thành bản luận văn tốt nghiệp.

**TÓM TẮT LUẬN VĂN**

Chủ để của luận văn là: “ Chứng minh tự động logic vị từ bằng phương pháp suy luận tự nhiên”.

Logic học là khoa học xuất hiện rất sớm trong lịch sử. Nó xuất hiện vào thế kỷ IV trước công nguyên, khi sự phát triển của khoa học nói riêng và tư duy nói chung đã đòi hỏi phải trả lời câu hỏi: làm thế nào để đảm bảo suy ra được kết luận đúng đắn từ các tiền đề chân thực?

Từ “Logic” có nguồn gốc từ Hy Lạp “Logos”, có rất nhiều nghĩa, trong đó hai nghĩa ngày nay được dùng nhiều nhất như sau. Thứ nhất, nó được dùng để chỉ tính quy luật của sự tồn tại và phát triển của thế giới khách quan. Thứ hai, từ “logic” dùng để chỉ những quy luật đặc thù của tư duy. Khi ta nói “Logic của sự vật là như vậy”, ta đã sử dụng nghĩa thứ nhất. Còn khi nói “Anh ấy suy luận hợp logic lắm”, ta dùng nghĩa thứ hai của từ logic.

Theo quan điểm phổ biến nhất hiện nay thì logic học là khoa học về các hình thức, các quy luật của tư duy. Nhưng khác với các khoa học khác cũng nghiên cứu về tư duy tâm lý học, sinh lý học thần kinh…, logic học nghiên cứu các hình thức và quy luật của tư duy để đảm bảo suy ra các kết luận chân thực từ các tiền đề, kiến thức đã có, và đưa ra các phương pháp để có được các suy luận đúng đắn. Để hiểu cặn kẽ hơn về đối tượng của logic học, ta phải tìm hiểu các đặc điểm của giai đoạn nhận thức lý tính và trả lời cho câu hỏi thế nào là hình thức và quy luật của tư duy.

Luận lý toán học hay luận lý là một lý thuyết phân tích những kỹ thuật của lý luận trong đời thường. Lý thuyết hướng tới việc hệ thống hóa và mã hóa các nguyên tắc của lý luận, từ đó rút ra được các qui luật của ngôn ngữ.

Luận lý toán học được hình thành từ việc nghiên cứu cách sử dụng ngôn ngữ tự nhiên trong lý luận. Tuy nhiên, hệ thống này không hàm chứa ý nghĩa của thực tế, nên nó có tính hình thức.

Luận lý toán học từ lâu đã được nghiên cứu và có nhiều công trình. Thêm vào đó, luận lý toán học lại có rất nhiều ngành, nhưng có hai dạng cổ điển, phổ biến là luận lý mệnh đề và luận lý vị từ.

Sự phát minh logic vị từ là một cuộc cách mạng lớn trong ngành triết học. Logic vị từ đủ mạnh để có thể diễn tả hết mọi lập luận của ngôn ngữ tự nhiên (đặc biệt thông dụng là logic vị từ bậc nhất). Lợi ích của logic vị từ là chúng ta có thể dễ dàng chứng minh một lập luận tự nhiên là đúng hay sai bằng cách đưa những lập luận ấy về dạng logic vị từ và chứng minh kết luận có đúng hay không nhờ vào những định lý, tiên đề của logic vị từ.

Có nhiều phương pháp để suy luận một mệnh đề logic vị từ. Phương pháp sơ khai nhất là phương pháp *suy luận tự nhiên*.

Suy luận tự nhiên được phát minh vào những năm 30 của thế kỉ trước bởi Gerhard Gentzen và Stanaslaw Jáskowski. Phương pháp này có đặc điểm là lập luận từ hệ tiên đề đã cho đi đến kết quả của bài toán dựa vào những luật suy luận tự nhiên, định lý đã được chấp nhận bởi logic vị từ.

Hệ thống chứng minh tự động logic vị từ bằng phương pháp suy luận tự nhiên là một hệ thống có khả năng chứng minh các công thức lập luận logic vị từ là đúng hay sai một cách tự động. Việc nghĩ ra những thuật toán thông minh và viết chương trình dựa trên nó có thể giúp chúng ta tổng hợp được các kiến thức đã học ở giảng đường, kỹ năng lập trình, tăng khả năng suy nghĩ thông minh, tạo điều kiện thuận lợi khi rời khỏi ghế nhà trường bước chân vào cuộc sống. Ở đây, đề tài luận văn được thực hiện trong một giai đoạn, do đó chúng tôi tập trung hiện thực để giải quyết vấn đề.

**MỤC LỤC**

**CHƯƠNG I**

**GIỚI THIỆU VỀ ĐỀ TÀI**

1. **Giới thiệu về logic**
   * 1. **Logic**

**Logic** hay **luận lý học**, từ tiếng Hy Lạp cổ điển λόγος (logos), nghĩa nguyên thủy là *từ ngữ*, hoặc *điều đã được nói*, (nhưng trong nhiều ngôn ngữ châu Âu đã trở thành có ý nghĩa là *suy nghĩ* hoặc *lập luận* hay *lý trí*). *Logic* thường được nhắc đến như là một ngành nghiên cứu về tiêu chí đánh giá các luận cứ, mặc dù định nghĩa chính xác của logic vẫn là vấn đề còn đang được bàn cãi giữa các triết gia. Tuy nhiên khi môn học được xác định, nhiệm vụ của nhà logic học vẫn như cũ: làm đẩy mạnh tiến bộ của việc phân tích các suy luận có hiệu lực và suy luận ngụy biện để người ta có thể phân biệt được luận cứ nào là hợp lý và luận cứ nào có chỗ không hợp lý.

Theo truyền thống, logic được nghiên cứu như là một nhánh của triết học. Kể từ giữa thế kỉ 19 logic đã thường được nghiên cứu trong toán học và luật. Gần đây nhất logic được áp dụng vào khoa học máy tính và trí tuệ nhân tạo. Là một ngành khoa học hình thức, logic nghiên cứu và phân loại cấu trúc của các khẳng định và các lý lẽ, cả hai đều thông qua việc nghiên cứu các hệ thống hình thức của việc suy luận và qua sự nghiên cứu lý lẽ trong ngôn ngữ tự nhiên. Tầm bao quát của logic do vậy là rất rộng, đi từ các đề tài cốt lõi như là nghiên cứu các lý lẽ ngụy biện và nghịch lý, đến những phân tích chuyên gia về lập luận, chẳng hạn lập luận có xác suất đúng và các lý lẽ có liên quan đến quan hệ nhân quả. Ngày nay, logic còn được sử dụng phổ biến trong lý thuyết lý luận.

Qua suốt quá trình lịch sử, đã có nhiều sự quan tâm trong việc phân biệt lập luận tốt và lập luận không tốt, và do đó logic đã được nghiên cứu trong một số dạng ít nhiều là quen thuộc đối với chúng ta. Logic Aristotle chủ yếu quan tâm đến việc dạy lý luận thế nào cho tốt, và ngày nay vẫn được dạy với mục đích đó, trong khi trong logic toán học và triết học phân tích (*analytical philosophy*) người ta nhấn mạnh vào logic như là một đối tượng nghiên cứu riêng, và do vậy logic được nghiên cứu ở một mức độ trừu tượng hơn.

Các quan tâm về các loại logic khác nhau giải thích rằng logic không phải là được nghiên cứu trong chân không. Trong khi logic thường có vẻ tự cung cấp sự thúc đẩy chính nó, môn học này phát triển tốt nhất khi lý do mà chúng ta quan tâm đến logic được đặt ra một cách rõ ràng.

* + 1. **Logic vị từ**

Môn Logic như được nghiên cứu ngày nay rất khác với môn học đã được nghiên cứu trước đây, và sự khác biệt chính là sự phát minh của **logic vị từ** (predicated logic or first-order logic). Trong khi logic tam đoạn luận của Aristote định ra những dạng thức cho những phần có liên quan với nhau trong mỗi phán đoán, logic vị từ cho phép các câu được phân tích thành chủ đề và các luận cứ theo nhiều cách khác nhau, do vậy cho phép logic vị từ giải quyết được vấn đề tổng quát hóa nhiều lần - vấn đề đã làm bối rối các nhà logic học thời trung cổ. Với logic vị từ, lần đầu tiên, các nhà logic học đã có khả năng đưa ra các phép lượng hóa (*quantifiers*) đủ tổng quát để diễn tả mọi luận cứ có mặt trong ngôn ngữ tự nhiên.

Sự khám phá ra logic vị từ thường được coi là công của Gottlob Frege, người cũng được xem là một trong những sáng lập viên của ngành triết học phân tích, nhưng dạng phát biểu có hệ thống thông dụng nhất ngày nay của logic vị từ là logic bậc nhất (*first-order logic*) được trình bày trong cuốn sách Các nguyên lý về logic lý thuyết (*Grundzüge der theoretischen Logik*) của David Hilbert và Wilhelm Ackermann vào năm 1928. Tính tổng quát có tính phân tích của logic vị từ cho phép hình thức hóa toán học và đẩy mạnh nghiên cứu về lý thuyết tập hợp, cho phép sự phát triển của cách tiếp cận của Alfred Tarski đối với lý thuyết mô hình; và không quá lời khi nói rằng nó là nền tảng của logic toán học hiện đại.

Hệ thống nguyên thủy của Frege về logic vị từ không phải là bậc nhất mà là bậc hai. Logic bậc hai được bảo vệ mạnh mẽ nhất bởi George Boolos và Stewart Shapiro (trước các phê phán của Willard Van Orman Quine và những người khác).

1. **Giới thiệu về suy luận tự nhiên**

Trong triết học logic, *suy luận tự nhiên* (natural deduction)là một cách tiếp cận lý thuyết chứng minh gần giống với cách suy luận của con người nhất so với những cách suy luận khác. Suy luận tự nhiên cố gắng cung cấp một hệ thống luận lý là mô hình hình thức của logic luận lý làm cho những lý luận này diễn ra theo một cách *tự nhiên.* Suy luận tự nhiên giữ lại cấu trúc của các công thức theo dạng tự nhiên.

Suy luận tự nhiên là nền tảng cơ bản nhất để tìm hiểu và giảng dạy logic. Suy luận tự nhiên dường như không được sử dụng để áp dụng vào lĩnh vực chứng minh tự động. Thay vào đó, người ta dùng các phương pháp như phân giải (resolution), tableau, dpll…

Mô hình của suy luận tự nhiên là:

A ⊢ B

* Trong đó hệ thống phải đi tìm những lập luận tự nhiên để dẫn xuất ra công thức B từ tập công thức A
* Tập công thức A có thể trống. Khi đó hệ thống sẽ là:

⊢ B

và lúc này suy luận B đồng nghĩa với suy luận định lý B.

Dẫn xuất:

Một dẫn xuất F của hệ thống A ⊢B là một công thức có được từ giả thiết hoặc từ các công thức bằng cách áp dụng các luật suy diễn tự nhiên.

Suy luận B từ A là tạo ra một dãy dẫn xuất A0, A1,…,An. Trong đó An chính là B.

**CHƯƠNG II**

**PHÂN TÍCH VÀ THIẾT KẾ HỆ THỐNG**

1. **Phân tích Input và Ouput**
   1. **Input**

Như chúng ta đã biết, logic vị từ là một ngôn ngữ hình thức. Do đó, những câu trong ngôn ngữ tự nhiên sẽ được biểu diễn thành những câu trong ngôn ngữ logic với một hình thức khác. Có những kí hiệu đặc biệt trong ngôn ngữ logic. Cụ thể là các toán tử kết nối như: ∧ , ∨ , ¬ , → , … Chính vì vậy ta phải thiết kế một bộ kí tự tương đương với những kí tự đặc biệt này.

Bộ kí tự tương đương như sau:

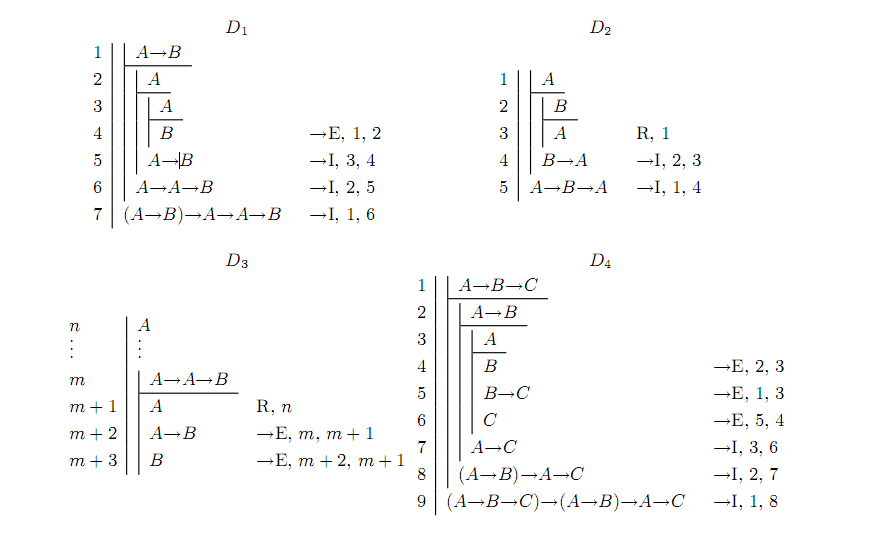
|  |  |
| --- | --- |
| Kí tự đặc biệt trong logic | Kí tự thay thế |
| ∧ | & |
| ∨ | ∨ |
| ¬ | ! |
| → | → |
| ├─ | ∨- |
| ∀ | \- |
| ∃ | ∃ |
| ⊥ |  |

Input sẽ là một chuỗi (string) gồm nhiều công thức logic vị từ. Các công thức này được ngăn cách nhau bởi dấu phẩy “,” . Tiếp theo sau đó là một toán tử kết luận (├─) và cuối cùng sẽ là một công thức logic (đây là công thức sẽ được dẫn xuất bởi hệ thống).

* 1. **Ouput**

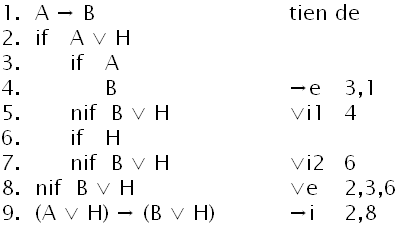
Suy luận tự nhiên có hai cách biểu diễn kết quả cổ điển: G-style (do Gerhard Gentzen đề xuất) và J-style (do Stanaslaw Jáskowski đề xuất).

Đối với yêu cầu của đề tài, Ouput của chương trình sẽ theo định dạng J-style, chính xác hơn là F-style (được cải tiến từ J-style bởi nhà logic học người Mỹ Frederic Brenton Fitch).



Định dạng F-style:

Ví dụ: A → B ┠ (A ∨ H ) → (B ∨ H)



1. **Phân tích thuật giải**
   1. **Nhóm luật**

Suy luận tự nhiên gồm 2 nhóm luật:

* Eliminate rules: gồm những luật dùng để phân rã công thức thành những công thức mới
* Introduction rules: gồm những luật dùng để tạo ra công thức mới từ những công thức thành phần.

Danh sách tập luật

|  |  |
| --- | --- |
| Eliminate rules | Introduction rules |
| ∧e1 | ∧i |
| ∧e2 | ∨i1 |
| →e | ∨i2 |
| ∨e | →i |
| ¬¬ | ¬i |
| ¬e | ∀i |
| ∀e | ∃i |
| ∃e |  |

Chú thích:

1. α : là giá trị bất kỳ
2. β : là giá trị có ràng buộc
3. [A] : giả thiết được thêm vào
   1. **Giải thuật chính**

Từ bài toán

A ⊢ B

Khởi tạo: *list\_proof* = {A}

*list\_goal* = {B}

Giải thuật được hiện thực theo GOAL DRIVEN STRAGEDY tức là mỗi hành động đều phải dựa vào GOAL.

**Contradiction:**

|  |  |
| --- | --- |
| Công thức | hoping\_goal |
| A → B | A |
| A ∨ B | ¬ A |
| A ∨ B | ¬ B |

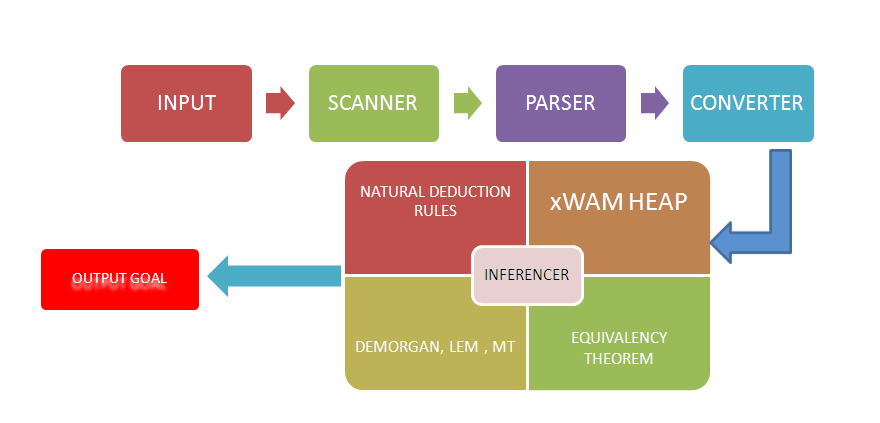
Đây là một procedure được thực thi để sinh ra các hoping\_goal theo bảng trên khi không thể áp dụng eliminate hoặc introduction.

**Giải thuật:**

* 1. Khởi tạo ***list\_proof*** và ***list\_goal***
  2. Tìm goal trong ***list\_proof***
     1. Nếu có: goto 3
     2. Nếu không thì áp dụng **eliminate** vào ***list\_proof***
        1. Nếu **eliminate** thành công: goto 2
        2. Nếu thất bại: goto c
     3. Áp dụng **introduction** vào goal hiện tại
        1. Nếu **introduction** thành công: goto 2
        2. Nếu thất bại: goto d
     4. Thực hiện **contradiction** vào ***list\_proof***
        1. Nếu thành công: goto 2
        2. Nếu thất bại: Failed
  3. Lấy goal hiện tại ra khỏi ***list\_goal***
     1. Nếu ***list\_goal*** trống: Success
     2. Nếu không trống: goto 2

**CHƯƠNG III**

**HIỆN THỰC HỆ THỐNG**

****

1. **Input**

Input của chương trình là dữ liệu ở dạng text theo định dạng của cấu trúc logic vị từ.

* 1. **Cấu trúc của logic vị từ**
* **Bảng ký tự :**Tập hợp hữu hạn các ký tự.

Thí dụ :

a, b, c, d, …, z

* **Ký hiệu :** Chuỗi hữu hạn ký tự được dùng để đặt tên cho các khái niệm trong FOL.

Thí dụ :

tên biến: x, y, …

tên hàm: cong, nhan, chia, …

* **Miền đối tượng D** : là một tập hợp.
* **Tập hợp các ký hiệu biến** : các biến lấy giá trị trên D.
* **Lượng từ có 2 loại :**
* Phổ dụng ∀ (universal quantifier)
* Hiện hữu ∃ (existential quantifier).

Hình thức sử dụng :

(∀x), (∃x) : với x là biến.

* **Hàm** là ánh xạ từ Dn → D, n ∈ N N.

Thí dụ :

nhân, cộng : D × D → D

* **Ảnh** của hàm được gọi là biểu thức hàm.

Thí dụ :

nhân(2, 3), cộng(x, 6)

cộng(nhân(y, z), 5)

* Trường hợp đặc biệt : Hàm từ D0 → D được gọi là hằng.
* Trường hợp đặc biệt : Hàm được gọi là vị từ nếu thỏa thêm các điều kiện :

− Có miền ảnh là tập {1, 0}.

− Chỉ kết hợp với nhau qua các toán tử logic : ¬, ∧, ∨, →

− Khi sử dụng không được làm thông số của hàm khác.

* **Vị từ**

Thí dụ :

cha, mẹ, bạn : D × D → {1, 0}

* **Ảnh của vị từ** được gọi là biểu thức vị từ.

Thí dụ :

cha(Minh, Vũ)

mẹ(x, y)

bạn(y, z)

* 1. **BNF Logic vị từ**

Sentence → AtomicSentence ∨ Sentence Connective Sentence

∨ Quantifier Variable, ... Sentence ∨ ¬ Sentence ∨ (Sentence)

AtomicSentence → Predicate(Term, …) ∨ Term = Term

Term → Function(Term, …) ∨ Constant ∨ Variable

Connective → ∧ ∨ ∨ ∨ 🡪

Quantifier → ∀ ∨ ∃

Constant → A, B, C, X1 , X2, Jim, Jack

Variable → a, b, c, x1 , x2, counter, position

Predicate → Adjacent-To, Younger-Than,

Function → Father-Of, Square-Position, Sqrt, Cosine

* 1. **Input của hệ thống**

Input của chương trình là một đề toán logic vị từ gồm có 2 phần: CONDITION và GOAL. Theo cấu trúc:

CONDITION ⊢ GOAL

* CONDITION: gồm một hoặc nhiều công thức trong đó một công thức là một câu theo cấu trúc của BNF đã cho ở phần trên. Các công thức này được ngăn cách nhau bởi dấu phẩy ‘,’ . CONDITION có thể rỗng.
* GOAL: gồm một công thức trong đó một công thức là một câu theo cấu trúc của BNF đã cho ở phần trên

1. **SCANNER**

Nhận dạng chuỗi nhập và chuyển chuỗi nhập thành dạng Token, đồng thời cho biết vị trí của Token đó. Các Token và các lexeme tương ứng được qui ước như sau:

LGC\_VAR = ‘x’, ‘y’, ‘z’, …(chuỗi bắt đầu bằng kí tự viết thường)

LGC\_INTERSECTION\_OP = ‘and’, ‘AND’, ‘ & ’

LGC\_UNION\_OP = ‘or’, ‘OR’, ‘ ∨ ’

LGC\_NEGATION\_OP = ‘not’, ‘NOT’, ‘!’, ‘~’

LGC\_MAPPING\_OP = ‘map’, ‘MAP’, ‘=>’, ‘→’

LGC\_EQUIVALENT\_OP = ‘<=>’ , ‘<→’

LGC\_LEFTPAR = ‘ (‘

LGC\_RIGHTPAR = ‘)’

LGC\_BOOLEANLITERAL = ‘true’, ‘false’, ‘TRUE’, ‘FALSE’

LGC\_CONTRADITION\_OP = ‘<>’

LGC\_ALL\_OP = ‘\-‘ , ‘all’ , ‘ALL’

LGC\_EXIST\_OP = ‘ ∃’ , ‘∃’ , ‘∃’

LGC\_RESULT\_OP = ‘=∨’ , ‘∨-‘

LGC\_COMMA = ‘,’

LGC\_CON = ‘P’, ‘Q’, …( chuỗi bắt đầu bằng kí tự viết hoa)

LGC\_NIL = $ (end of file)

LGC\_ERROR = others

Ví dụ: Chuỗi nhập là:

**p(t) , all x p(x)→ q(x)** ∨**- q(t)**

Các Token tương ứng là:

Token.LGC\_VAR Lexeme: p CharStart:2 CharFinish:2

Token.LGC\_LEFTPAR Lexeme: ( CharStart:3 CharFinish:3

Token.LGC\_VAR Lexeme: t CharStart:4 CharFinish:4

Token.LGC\_RIGHTPAR Lexeme: ) CharStart:5 CharFinish:5

Token.COMMA Lexeme: , CharStart:7 CharFinish:7

Token.LGC\_ALL\_OP Lexeme: all CharStart:9 CharFinish:11

Token.LGC\_VAR Lexeme: x CharStart:13 CharFinish:13

Token.LGC\_VAR Lexeme: p CharStart:15 CharFinish:15

Token.LGC\_LEFTPAR Lexeme: ( CharStart:16 CharFinish:16

Token.LGC\_VAR Lexeme: x CharStart:17 CharFinish:17

Token.LGC\_RIGHTPAR Lexeme: ) CharStart:18 CharFinish:18

Token.LGC\_MAPPING\_OP Lexeme: → CharStart:20 CharFinish:21

Token.LGC\_VAR Lexeme: q CharStart:23 CharFinish:23

Token.LGC\_LEFTPAR Lexeme: ( CharStart:24 CharFinish:24

Token.LGC\_VAR Lexeme: x CharStart:25 CharFinish:25

Token.LGC\_RIGHTPAR Lexeme: ) CharStart:26 CharFinish:26

Token.LGC\_RESULT\_OP Lexeme: ∨- CharStart:28 CharFinish:29

Token.LGC\_VAR Lexeme: q CharStart:31 CharFinish:31

Token.LGC\_LEFTPAR Lexeme: ( CharStart:32 CharFinish:32

Token.LGC\_VAR Lexeme: t CharStart:33 CharFinish:33

Token.LGC\_RIGHTPAR Lexeme: ) CharStart:34 CharFinish:34

Token.LGC\_NIL Lexeme: $ CharStart:35 CharFinish:35

1. **PARSER**

* Nhận kết quả từ bộ SCANNER
* Kiểm tra cú pháp chuỗi nhập có đúng với văn phạm của logic vị từ hay không
  + Nếu đúng cú pháp thì sẽ đưa vào bảng cấu trúc dữ liệu (bảng xWAM HEAP).
  + Nếu sai thì sẽ ngừng chương trình, đồng thời báo lỗi sai và vị trí sai đầu tiên để người dùng có thể dễ dàng sửa chữa.

Các qui ước như sau

* CONDITION
  + Các tiên đề ngăn cách nhau bởi dấu phẩy và được gọi là một câu (sentence).
  + Một câu sẽ có cú pháp theo cấu trúc văn phạm BNF logic vị từ đã nêu ở trên.
  + CONDITION có thể trống.
* GOAL
  + Là một câu.

Ví dụ 1: Chuỗi nhập là:

**p(t) , all x (p(x)→ q(x))** ∨**- q(t)**

Kết quả:

Sẽ gồm 3 câu như sau:

BeginSentence

p(t)

all x p(x)→ q(x)

q(t)

EndSentence

BeginSentence

EndSentence

BeginSentence

EndSentence

Có nghĩa là 3 câu trên sẽ được đưa vào cấu trúc dữ liệu

Ví dụ 2: Chuỗi nhập là:

p(t) , all x p(x **→** q(x) ∨- q(t) // (thiếu dấu đóng ngoặc tại biến x)

Kết quả:

Unexpected Token.LGC\_MAPPING\_OP Lexeme: **→** CharStart:19 CharFinish:20

Chương trình sẽ báo lỗi sai tại Token. LGC\_MAPPING\_OP tức là dấu : → (vì thiếu đóng ngoặc)

1. **CONVERTER**

* Chuyển các toán tử logic thành dạng hàm

Qui ước:

* + A ∨ B thành or(A,B)
  + A∧ B thành and(A,B)
  + A→B thành map(A,B)
  + ¬ A thành not(A)
* Độ ưu tiên của các toán tử
  + Theo tứ tự: ( ) ¬ ∧ ∨ →

Ví dụ:

A → ¬B ∧ (¬C ∨ D) thành map ( A, and( not(B), or( not(C),D)))

1. **xWAM HEAP**
   1. **Mô hình Warren Abstract Machine**
      1. **Lịch sử**

Vào năm 1983, David H. D. Warren đã thiết kế ra mô hình Abstract Machine để thực thi các chương trình Prolog được biết đến là WAM. Cấu trúc của WAM gồm hai phần:

* *Kiến trúc bộ nhớ*
* *Tập lệnh*

Ở đây chúng tôi chỉ quan tâm đến phần kiến trúc bộ nhớ. Kiến trúc bộ nhớ trong WAM là WAM HEAP, nó đã trở thành một chuẩn cấu trúc dữ liệu trong chương trình dịch của Prolog trong thực tế.

* + 1. **Kiến trúc bộ nhớ trong mô hình WAM**

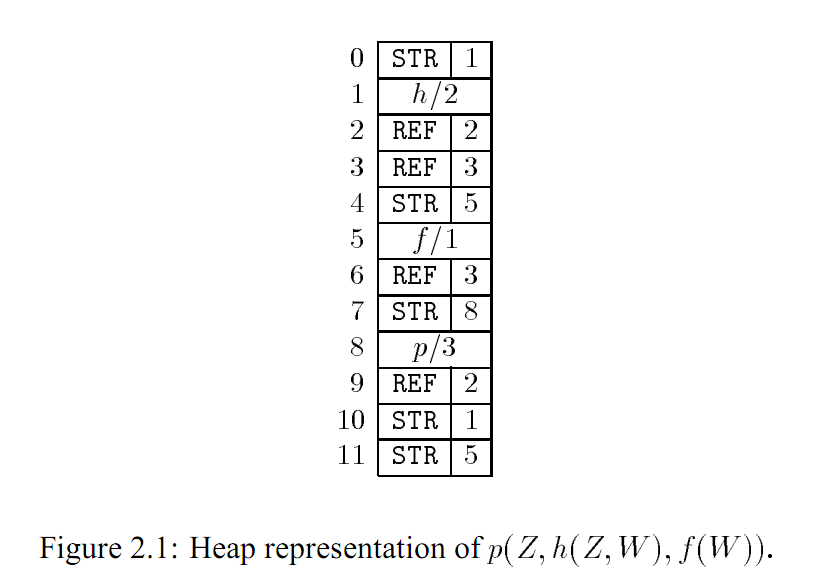
- WAM HEAP sử dụng một array chứa các data cell. Địa chỉ của cell chính là chỉ số index trong array. Chỉ có 2 loại dữ liệu được lưu trong data cell. Đó là variable và structure

- Variable : sử dụng 1 phần tử của WAM heap để lưu trữ tham chiếu đến bảng VAR, được gọi là Variable cell.

- Structure: f(x1,x2,…,xn) : dùng n+2 phần tử của WAM heap để lưu trữ

Trong đó phần tử thứ nhất dùng để lưu kiểu structure và địa chỉ chứa nội dung của nó. Phần tử thứ 2 không nhất thiết phải liên tiếp với phần tử thứ nhất. Phần tử thứ hai dùng để chứa bậc structure (cụ thể ở đây là n). N phần tử tiếp theo ( liên tiếp với phần tử thứ hai) dùng để lưu trữ các thành phần con của structure.

Ví dụ: p (Z, h(X,W), f(W))



* 1. **xWAM HEAP**
* WAM HEAP được thiết kế cho PROLOG mà trong ngôn ngữ PROLOG không có chứa lượng từ hay chính xác hơn mặc định của PROLOG là universal ( ∀ )
* Logic vị từ có chứa lượng từ và chương trình có khả năng thể hiện ý nghĩa của lượng từ. Do đó, chúng tôi mở rộng WAM HEAP thành một cấu trúc dữ liệu mới có khả năng giải quyết vấn đề trên. Được đặt tên là xWAM HEAP.
* Đặc điểm của xWAM HEAP
  + Chứa được lượng từ: ∀ ∃
  + VAR cell giống WAM Heap
  + Structure cell thêm phần tử q tham chiếu đến bảng QUANT
  + Bảng QUANT: Chứa các lượng từ

Ví dụ: ∀x ∃y (p(x,y) → ¬q(x,y))

1. **INFERENCE**

Ý tưởng chính của giải thuật là việc nắm giữ 2 danh sách công thức *list\_proof* và *list\_goal*. Trong đó *list\_proof* là các công thức đã được dẫn xuất, còn *list\_goal* nắm giữ các công thức cần phải suy luận. *list\_goal* ban đầu chỉ là công thức cần chứng minh, nhưng được biến đổi trong quá trình chứng minh. *list\_proof* ban đầu chỉ bao gồm các công thức lấy được từ giả thiết. Trong quá chứng minh ở một thời điểm bất kỳ *list\_proof* và *list\_goal* có dạng:

{P0,P1,P2..Pn} ⊢ {G0,G1,G2..Gn}

* Dẫn xuất:

*Một công thức được dẫn xuất nếu nó có được từ giả thiết hoặc từ các công thức khác đã được dẫn xuất bằng cách áp dụng các luật suy luận tự nhiên.*

Ở đây goal có thể là một công thức bất kỳ hoặc mâu thuẫn (⊥) . Mọi hoạt động của giải thuật sẽ tác động lên *list\_proof* và *list\_goal*, đồng thời các hoạt động đó sẽ bị ảnh hưởng bởi current\_goal ( goal được chọn trong *list\_goal* ). Tại mỗi bước của giải thuật , current\_goal sẽ được kiểm tra nó có mặt trong *list\_proof* hay chưa?

* Sự có mặt :

Current\_goal được xem như có mặt khi và chỉ khi:

* Current\_goal khác ⊥ và chúng ta tìm được một công thức trong *list\_proof* mà tồn tại một phép đồng nhất giữa current\_goal và Pn
* Current\_goal là ⊥ và chúng ta tìm được một cặp công thức Pn , ¬Pk mà tồn tại một phép đồng nhất giữa Pn và Pk
* Đồng nhất:

A,B được gọi là có khả năng đồng nhất khi và chỉ khi tồn tại một thay thế σ mà σ(A) = σ(B).

Một cách tổng quát, khi chúng ta xây dựng một dãy dẫn xuất, chúng ta kiểm tra xem current\_goal đã có mặt hay chưa? Nếu current\_goal đã có mặt, chúng ta sẽ áp dụng những luật introduction để tạo ra công thức mới (previous goal). Chỉ duy nhất trong trường hợp này, chúng ta mới áp dụng các luật introduction. Nếu công thức hiện tại không có mặt, chúng ta sẽ tiến hành thay đổi *list\_proof* và *list\_goal* dựa vào current\_goal trong *list\_goal*.

Nhắc lại hệ thống suy luận tự nhiên ND

A ⊢ B

Hệ thống được chứng minh dựa vào *list\_proof* và *list\_goal*

{P0,P1,P2..Pn} ⊢ {G0,G1,G2..Gn}

Chúng ta sẽ trình bày rõ ràng các module được thực hiện trong quá trình chứng minh:

* 1. **Initiation**

Module này có chức năng dùng để khởi tạo *list\_proof* và *list\_goal* sau khi xWAM đã được tạo ra.

Module này chỉ được gọi 1 lần vào lúc bắt đầu chứng minh.

* 1. **Elimination**

Module này có chức năng tạo ra những những dẫn xuất mới từ những dẫn xuất có trong *list\_proof* . Module này cố gắng áp dụng các luật eliminate vào 1 hay 1 tập công thức trong *list\_proof*. Các dẫn xuất mới này sẽ được thêm vào *list\_proof*. Việc tìm kiếm các công thức hay cặp công thức sẽ được hoạt động theo breadth-first-search. *list\_goal* trong trường hợp này không thay đổi trừ các trường hợp đặc biệt sẽ được trình bày sau.

Thí dụ : Xét một bài toán

{A ∧ B , A→C, B→ ¬ ¬ E} ⊢ {Gn}

Áp dụng elimination lên tập *list\_proof* của bài toán sẽ được bài toán mới :

|  |  |
| --- | --- |
| list\_proof | list\_goal |
| * + - 1. A ∧ B       2. A→C       3. B→ ¬ ¬ E       4. A ( ∧e A ∧ B)       5. B ( ∧e A ∧ B) | Gn |

|  |  |
| --- | --- |
| list\_proof | list\_goal |
| 1. A ∧ B 2. A→C 3. B→ ¬ ¬ E 4. A ( ∧e A ∧ B) 5. B ( ∧e A ∧ B) 6. C ( →e A → C, A) | Gn |

|  |  |
| --- | --- |
| list\_proof | list\_goal |
| 1. A ∧ B 2. A→C 3. B→ ¬ ¬ E 4. A ( ∧e A ∧ B) 5. B ( ∧e A ∧ B) 6. C ( →e A → C, A) 7. ¬ ¬ E (→e B→ ¬ ¬ E , B) | Gn |

|  |  |
| --- | --- |
| list\_proof | list\_goal |
| 1. A ∧ B 2. A→C 3. B→ ¬ ¬ E 4. A ( ∧e A ∧ B) 5. B ( ∧e A ∧ B) 6. C ( →e A → C, A) 7. ¬ ¬ E (→e B→ ¬ ¬ E , B) 8. E ( ¬ ¬ ¬ ¬ E) | Gn |

Sau khi áp dụng elimination lên tập *list\_proof*  ta có được bài toán mới

{A ∧ B , A→C, B→ ¬ ¬ E , A , B, C , ¬ ¬ E , E} ⊢ {Gn}

* Lưu ý : các phần tử trong *list\_proof* phải được đánh dấu sao cho việc áp dụng luật eliminate không bị lặp lại

Trường hợp đặc biệt: OR, ∃ sẽ được giải thích kỹ ở…

* 1. **Introduction**

Khi Elimination không thể áp dụng được nữa và current\_goal chưa có mặt trong *list\_proof*. Chúng ta phải “rã” current\_goal ra thành nhiều goal mới, việc rã này có thể ảnh hưởng đến *list\_proof*. Giả sử bài toán hiện tại là:

{P0,P1,P2…Pn} ⊢ {G0,G1,G2…Gn}

Gn là current\_goal và Gn có một trong các dạng sau:

* **Gn = A ∧ B**
  + Để chứng minh A ∧ B ,một cách “tự nhiên”, chúng ta phải chứng minh được công thức A và công thức B. Do đó thêm vào *list\_goal* 2 goal nữa là A, B đồng thời thiết lập thuộc tính *pending* cho Gn bằng 2 , tức là Gn sẽ xem như được chứng minh nếu 2 goal sau nó Gn+1, Gn+2 được chứng minh. Bài toán thành

{P0,P1,P2…Pn} ⊢ {G0,G1,G2…A ∧ B (2) , B, A}

* **Gn = A → B**
  + Để chứng minh A→B , chúng ta phải chứng minh B với giả thiết A . Do đó việc cập nhật *list\_proof* và *list\_goal* được thực hiện như sau:
    1. Thêm vào *list\_proof* 1 công thức là A , đồng thời đánh dấu đây là giả sử , tức là sau khi chứng minh được A→B, giả sử này phải được loại bỏ đi, đồng thời các dẫn xuất có liên quan đến nó cũng được bỏ đi.
    2. Thêm vào *list\_goal* một công thức mới là B và thiết lập thuộc tính *pending* cho A→B là 1.
  + Bài toán thành

{P0,P1,P2…Pn, A(\*)}⊢{G0,G1,G2…,A→B(1) ,B}

* **Gn = A ∨ B**
  + Đây là trường hợp phức tạp hơn so với 2 trường hợp đầu . Để chứng minh được A ∨ B , chúng ta có thể chỉ cần chứng minh được A hoặc B .
    1. Nếu A ∨ B rã lần đầu tiên : Để có A ∨ B , chúng ta cần chứng minh A
       - Thêm công thức A vào *list\_goal* , thiết lập thuộc tính *pending* cho A ∨ B là 1 , đồng thời đánh dấu công thức A ∨ B để có thể quay về “rã” A ∨ B lần thứ 2
  + Bài toán thành

{P0,P1,P2…Pn}⊢{G0,G1,G2…,A ∨ B(1)(1) , A}

1. Nếu A ∨ B rã lần thứ hai:
   * + - Thêm công thức B vào *list\_goal* , sau đó thiết lập thuộc tính *pending* cho A ∨ B là 1, đồng thời đánh dấu công thức A ∨ B để A ∨ B không thể phân rã thêm nữa.
       - Bài toán thành

{P0,P1,P2…Pn}⊢{G0,G1,G2…,A ∨ B(1)(2) , B}

* **Gn = ¬ A**
  + Để chứng minh được ¬ A , chỉ có thể hoặc ¬ A có mặt trong *list\_proof* hoặc chứng minh ⊥ với điều kiện giả thiết A. Đây là lối chứng minh phản chứng. Do đó *list\_goal* và *list\_proof* được cập nhật như sau:
    1. Thêm ⊥ vào *list\_goal*, thiết lập thuộc tính pending của ¬ A là 1
    2. Thêm công thức A vào *list\_proof*, đánh dấu A là giả sử

Bài toán thành

{P0,P1,P2…Pn,A} ⊢ {G0,G1,G2…,¬ A, ⊥}

* **Gn = ∀x f(x)**
  + Để chứng minh ∀x f(x), chúng ta luôn phải tìm một f(x0) với giả thiết x0 bất kỳ. Do đó *list\_proof* sẽ được thêm vào 1 công thức x0 , đánh dấu x0 là 1 biến bất kỳ không có ràng buộc, đồng thời x0 là giả sử . *list\_goal* sẽ được thêm vào công thức f(x0), và đồng thời thiết lập thuộc tính *pending* của ∀ x f(x) là 1. Sau khi ∀x f(x) được chứng minh, x0 hoặc các công thức có chứa x0 không được xuất hiện trong dãy dẫn xuất nữa.

{P0,P1,P2…Pn, x0}⊢{G0,G1,G2…,∀ xf(x),f(x0)}

* **Gn = ∃x g(x)**
  + Để chứng minh ∃x f(x) , chúng ta chỉ cần tìm một công thức f(α) với α là một giá trị bất kỳ. Do đó chỉ có *list\_goal* được cập nhật. Thêm f(α) vào *list\_goal* với α được thiết lập thuộc tính ANY\_VALUE , cập nhật thuộc tính *pending* của ∃x f(x) là 1.

{P0,P1,P2…Pn, x0}⊢{G0,G1,G2…,∃x f(x),f(α)}

* **Gn = F**
* Khi Gn không phải là một trong các dạng trên hoặc Gn ở dạng A ∨ B nhưng đã phân rã 2 lần . Chúng ta phải đi chứng minh mâu thuẫn , tức là thêm vào *list\_proof*  ¬ F , đánh dấu ¬ F là giả sử đồng thời thêm vào *list\_goal* ⊥. Khi đó thuộc tính *pending* của F là 1.

{P0,P1,P2…Pn, ¬F} ⊢ {G0,G1,G2…, F, ⊥}

* 1. **Contradiction**

Xét bài toán:

{P0,P1,P2,.. **Pk** … Pn} ⊢{G0,G1,G2..Gn-1 , ⊥}

Khi không thể áp dụng **elimination** nữa , và vì ⊥ không thể “rã” , module này sẽ thực thi, nó sẽ tuần tự tìm kiếm các công thức phức tạp trong *list\_proof* để tạo hoping\_goal. Hoping\_goal là goal được mong đợi với sự xuất hiện của nó trong *list\_proof* có thể áp dụng được **elimination**. Giả sử Pk là công thức phức tạp chưa bị tác động của contradiction. Việc cập nhật *list\_goal* sẽ phụ thuộc vào dạng của Pk .

* Pk = A→B
* Nếu chứng minh được công thức A, khi đó A, A→B có thể áp dụng → rule.
* Do đó , hoping goal trong trường hợp này là A . Thêm A vào *list\_goal* , thiết lập A là hoping\_goal, đồng thời đánh dấu A→B đã được **contradiction** tác động lên.
* Bài toán trở thành

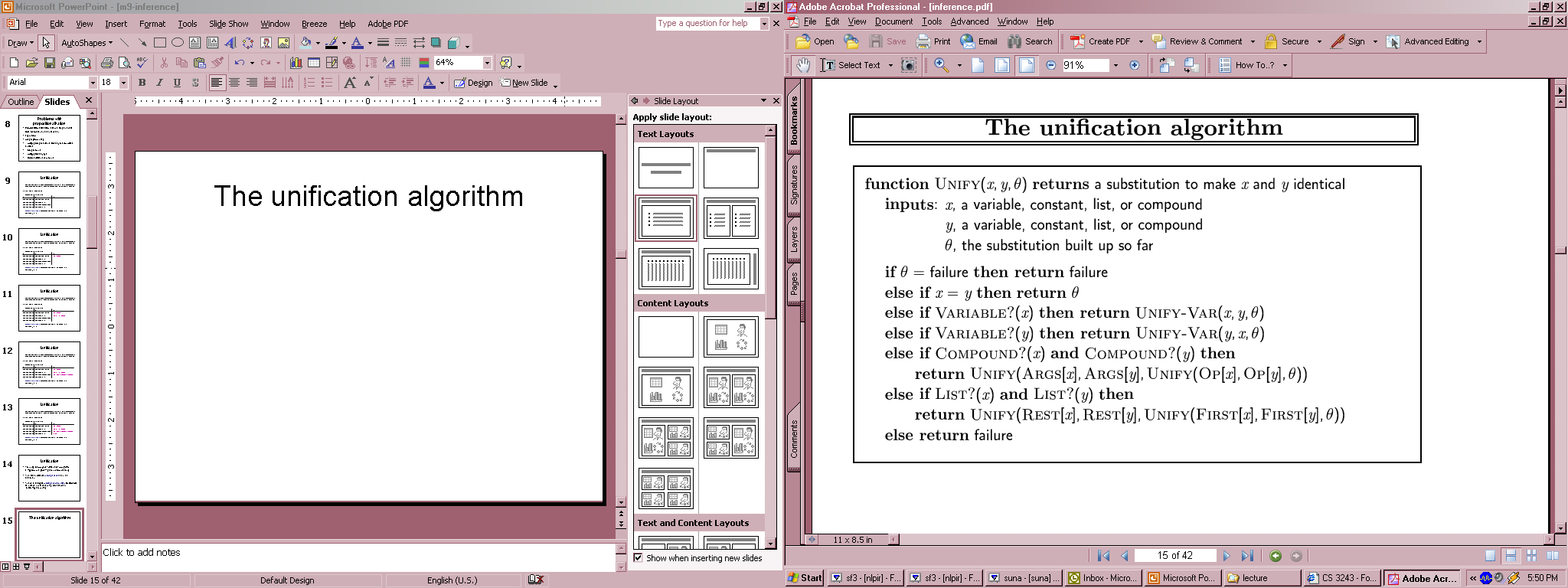
{P0,P1,P2,.., **A→B** ,… Pn} ⊢{G0,G1,G2..Gn-1 , ⊥ , A}

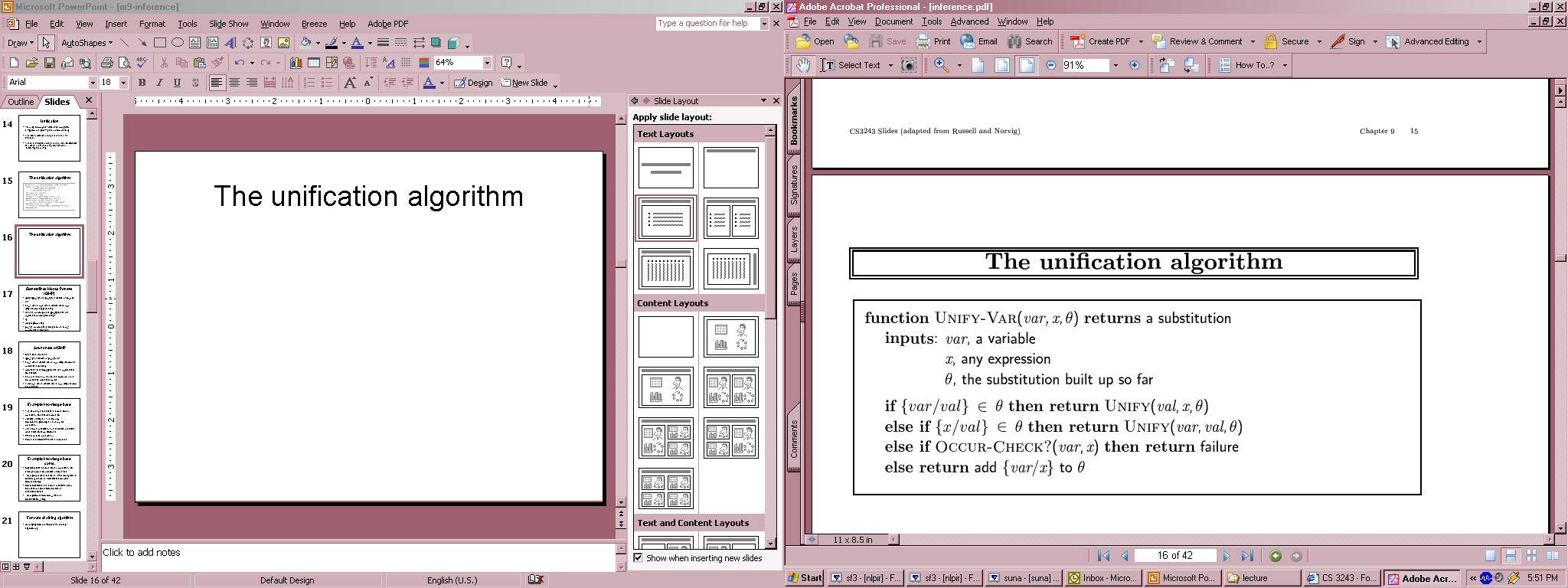
* Pk = ¬ A
* Nếu chứng minh được công thức A, khi đó ¬ A, A có thể áp dụng ⊥ rule.
* Do đó , hoping \_goal trong trường hợp này là A . Thêm A vào *list\_goal* , thiết lập A là hoping\_goal, đồng thời đánh dấu ¬ A đã được **contradiction** tác động lên.
* Bài toán trở thành

{P0,P1,P2,.., **¬A** ,… Pn} ⊢{G0,G1,G2..Gn-1 , ⊥ , A}

* 1. **Unification**

Chúng ta sử dụng **unification** để tìm tập thay thế của 2 công thức x, y. Việc tìm kiếm tập thay thế σ sẽ được thực hiện theo giải thuật sau

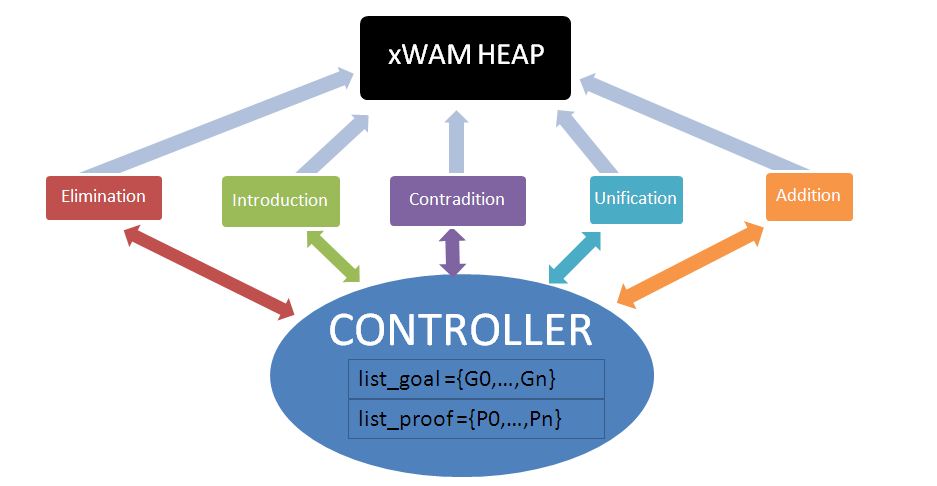




* 1. **Updating**

Khi current goal có *pending* lớn hơn 0 hay là current goal là một goal có được bằng cách áp dụng các luật *introduction* vào các dẫn xuất vừa có. Module này được thực thi để cập nhật rules được dùng để sinh ra current goal này.

* 1. **Controller**

****

Mô hình Inferencer

Controller được thực hiện theo giải thuật này

**Giải thuật:**

1. Gọi module **initiation**
2. Current\_goal có mặt trong ***list\_proof ?***
   * 1. Nếu có: goto 3
     2. Nếu không thì áp dụng **elimination** vào ***list\_proof***
        1. Nếu **elimination** thành công: goto 2
        2. Nếu thất bại: goto c
     3. Áp dụng **introduction** vào goal hiện tại
        1. Nếu **introduction** thành công: goto 2
        2. Nếu thất bại: goto d
     4. Thực hiện **contradiction** vào ***list\_proof***
        1. Nếu thành công: goto 2
        2. Nếu thất bại: Failed
3. Lấy goal hiện tại ra khỏi ***list\_goal***
   * 1. Nếu ***list\_goal*** trống: Success
     2. Nếu không trống: goto 2

Ban đầu chúng ta khởi tạo list\_proof và list\_goal, list\_proof có thể trống và có Gn = Go . Tiếp theo chúng ta kiểm tra xem G0 có mặt hay chưa? Giả sử G0 chưa có mặt, chúng ta tiến hành đi tìm các công thức mới bằng cách cố gắng áp dụng eliminate vào list\_proof. Nếu công việc này thành công, chúng ta sẽ quay lại kiểm G0 có mặt trong list\_proof hay chưa? Nếu list\_proof vẫn chưa có mặt thì chúng ta sẽ tiến hành phân rã G0 ra . Tùy thuộc vào cấu trúc của G0 , mà việc phân rã khác nhau. Lúc đó, thay vì đi chứng minh G0 chúng ta đi chứng minh Gn nào đó…

* 1. **Trường hợp đặc biệt**
     1. **OR Elimination**

Một cách tự nhiên, nếu có A →C, B → C thì có được (A ∨ B)→ C. Công thức này do chính chúng tôi đưa ra . Do đó để áp dụng OR elimination vào A ∨ B thì chúng ta tìm A→C, B→C trong *list\_proof*. Khả năng tìm được A→C, B→C trong *list\_proof* là rất thấp .Vì thế , thay chỉ tác động lên *list\_proof* , chúng tôi đi chứng minh công thức A→C, B→C với C là current goal.

Xét bài toán

{P0,P1,P2,.., **A ∨ B** ,… Pn} ⊢{G0,G1,G2..Gn-1 , Gn}

Để áp dụng OR elimination lên A∨B với mục tiêu đi chứng minh Gn. Chúng ta thêm A→Gn, B→Gn vào list\_proof. Khi đó ,sau khi có A→Gn và B→Gn, chúng ta có áp dụng luật OR Eliminate vào A∨B, Gn[A].Gn[B] để có được Gn.

{P0,P1,P2,.., **A ∨ B(\*)** ,… Pn} ⊢{G0,G1,G2..Gn, Gn[A] , Gn[B] }

Dạng trình bày của OR Elimination là:

* dòng m : F ∨ G
* dòng n : if F
* dòng n+p : nif H
* dòng k : if G
* dòng k+q : nif H
* dòng k+q+1 : H

Nếu F sinh ra H và G cũng sinh ra H thì F ∨ G cũng sinh ra H.

Xét bài toán :

{A∨B, C, C→ D} ⊢ {D}

Dễ dàng tìm ra lời giải cho bài toán bằng cách áp dụng → e vào C, C→D. Nhưng nếu chúng ta đã áp dụng ∨e vào A∨B, thì rõ ràng D chịu sự ảnh hưởng của A∨B.

Tức là :

1. C → D
2. C
3. A∨B
4. If A
5. Nif D (→e 1,2)
6. If B
7. Nif D (→e 1,2)
8. D

Chúng ta cần loại bỏ sự ảnh hưởng vô nghĩa của A∨B. Để giải quyết vấn đề này chúng ta cần phải lưu lại nguồn gốc của dẫn xuất. Nếu trong trường hợp này D không hề có được dẫn xuất từ A∨B thì sự ảnh hưởng sẽ bị loại bỏ.

Xét bài toán khác:

{F→H} ⊢ { F∨G → H∨G∨R}

Bài giải được xem là hoàn hảo:

1. F→H
2. If F ∨ H
3. If F
4. H (→e 1,3)
5. Nif H ∨ G
6. If G
7. Nif H ∨ G
8. H ∨ G
9. Nif H ∨ G ∨ R
10. F ∨ G → H ∨ G ∨ R

Thế nhưng, với giải thuật trên bài toán sẽ được giải là

1. F→H
2. If F ∨ H
3. If F
4. H (→e 1,3)
5. H ∨ G
6. Nif H ∨ G ∨ R
7. If G
8. H ∨ G
9. Nif H ∨ G ∨ R
10. Nif H ∨ G ∨ R
11. F ∨ G → H ∨ G ∨ R

Rõ ràng sự ảnh hưởng của F∨H chỉ tác động trực tiếp lên H∨G còn H∨G∨R chi chịu ảnh hưởng từ H∨G. Do đó chúng ta cần xác định lại phạm vi ảnh hưởng của F∨H lên các dẫn xuất

F ∨ H ảnh hưởng lên H ∨ G ∨ R

F ∨ G ảnh hưởng lên H ∨ G

Để loại bỏ sự ảnh hưởng vô nghĩa của F ∨ G lên các dẫn xuất của bài toán dễ dàng được thực hiện dựa vào 2 hình trên.

* + 1. **∃ Eliminate:**

Nếu f(x0) với x0 là giá trị bất kỳ thỏa mãn f(x0) là một công thức và với f(x0) , chúng ta dẫn xuất được công thức G. Như thế chúng ta sẽ có ∃x f(x) → G bằng cách áp dụng luật ∃ eliminate.

Mô hình

If x 0 f(x0)

…

Nif G

G

Xét bài toán

…, ∃x f(x) ⊢ Gn

Để áp dụng ∃e vào ∃x f(x), chúng ta phải tìm một giá trị x0 chưa từng xuất hiện trong các dẫn xuất trước đó. Thêm vào công thức f(x0) vào *list\_proof* đồng thời xóa sự có mặt của ∃x f(x). Có thể ∃ x f(x) và f(x0) không hoàn toàn tương đương nhau nhưng trong trường hợp này là tương đương vì từ ∃xf(x) chúng ta có được f(x0) và ngược lại.

f(x0) sẽ tồn tại xuyên suốt quá trình chứng minh, vấn đề đặt ra là dẫn xuất nào chịu ảnh hưởng, dẫn xuất nào không bị ảnh hưởng.

Xét bài toán :

{∃x f(x), ∀ x f(x) → G } ⊢ {G∨H}

Bài chứng minh hoàn hảo là:

1. ∀ x f(x) → G
2. {∃x f(x)
3. If x0 , f(x0)
4. f(x0) → G (∀e 1)
5. Nif G (→e 3,4)
6. G (∃e 2)
7. G ∨ H

Thế nhưng theo giải thuật trên bài toán sẽ được giải như sau:

1. ∀ x f(x) → G
2. {∃x f(x)
3. If x0 , f(x0)
4. f(x0) → G (∀e 1)
5. G (→e 3,4)
6. Nif G ∨ H
7. G ∨ H (∃e 2)

Nif G chính là điểm dừng sự ảnh hưởng của f(x0) lên tập dẫn xuất.

Rõ ràng với giải thuật trên chưa , vấn đề này giải quyết được.

Để giải quyết vấn đề này chúng ta bắt đầu thì G0, đi dọc theo các goal con nó để chọn điểm cắt theo hình sau

* 1. **Áp dụng của giải thuật tìm kiếm chứng minh**

Ở phần này chúng tôi sẽ trình chi tiết từng bước trong giải thuật trên.

Thí dụ

F →H ⊢ F∨G → H∨G

Ở bước khởi tạo chúng ta sẽ được 2 danh sách list\_proof và list\_goal như sau

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| * + - 1. F→H | Go = {F∨G → H∨G} |

Tiếp theo chúng ta thực hiện elimination cho tập *list\_proof*. Việc eliminate thất bại vì có F→H nhưng không tìm thấy F.

Do đó trạng thái hiện tại của *list\_proof* và *list\_goal* vẫn không đổi

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H | Go = {F∨G → H∨G} |

Theo giải thuật nếu việc **eliminate** thất bại, chúng ta tiến hành thực thi **introduction** lên current goal. Current goal chính là G0 và G0 có dạng A→B. Dựa theo module **Introduction**, giả thiết F∨G sẽ được thêm vào *list\_proof* và H∨G sẽ được thêm vào *list\_goal*. G0 sẽ được thiết lập thuộc tính *pending* là 1.

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F∨G | Go = {F∨G → H∨G}  {F∨G→H∨G, H∨G} |

Sau khi **introduction** thành công , current goal sẽ được thiết lập lại bằng G1 = {H∨G}. Theo giải thuật, chúng ta sẽ quay lại bước 1 tức là sẽ đi kiểm tra H∨G có mặt trong tập *list\_proof* hay chưa? Kết quả H∨G chưa có mặt trong *list\_proof*.

Chúng ta tiến hành thực thi module **elimination** lên tập *list\_proof*. Lúc này *list\_proof* có 2 phần tử F→H và F∨G . Module **Elimination** thực hiện theo breadth first search nên sẽ duyệt các phần tử từ trái sang phải . F→H không thể **eliminate** được vì không tìm được F. Công thức còn lại F∨G có thể **eliminate** theo trường hợp đặc biệt OR. Theo (1) việc tiến hành **eliminate** OR sẽ thay đổi *list\_goal* và *list\_proof* như sau: *list\_goal* sẽ được thêm vào 2 công thức H∨G [F] và H∨G [G] còn Go = {F∨G → H∨G} được thiết lập thuộc tính *pending* là 2.

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F∨ G assumption | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G , H∨G , H∨G[G] , H∨G[F] |

Lúc này chúng ta lại quay về bước 1. Current goal được thiết lập là H∨G[F]. Tức là tìm H∨G với điều kiện F. Thêm giả thiết F vào tập *list\_proof*, đánh dấu F là giả sử của H∨G.

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. F assumption | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G |

Lúc này tiến hành tìm kiếm sự có mặt của H∨G. Dễ dàng nhận ra H∨G không có mặt trong tập *list\_proof*. Tiến hành áp dụng **elimination** lên *list\_proof*. F→H áp dụng thành công **elimination** nhờ có F. Do đó công thức H sẽ được thêm vào *list\_proof*

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. F assumption 4. H | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G ,H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G |

Việc **elimination** thành công, chương trình quay về bước 1, tiến hành xác định H∨G có mặt hay chưa? Kết quả là H∨G vẫn chưa có mặt. Tiến hành **elimination** tiếp tục và thất bại. Do không thể áp dụng **eliminate** vào *list\_proof* nên phải “rã” current goal. Việc rã sẽ được thực hiện theo **OR Introduction**. Do H∨G rã lần đầu tiên nên *list\_goal* sẽ thêm vào công thức thứ nhất là H

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. F assumption 4. H →e 1,3 | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G,H |

Quay về bước 1, tiến hành tìm kiếm sự có mặt của H trong *list\_proof*. H có mặt. Như vậy goal H đã được chứng minh. Lấy goal H ra khỏi *list\_goal*. Thiết lập current goal là H∨G. Quay về bước 1.

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. F assumption 4. H →e 1,3 | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G |

Current goal là H∨G và H∨G có *pending* bằng H∨G tức là 1 được sinh ra từ 1 goal kế trước nó. Gọi module **Updating**, ta có được H ∨ G sinh ra từ H theo ∨i rules. Đưa H ∨G vào *list\_proof.*

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. F assumption 4. H →e 1,3 5. H ∨ G ∨i 4 | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G] |

Loại bỏ H∨G ra khỏi *list\_goal*. Ở bước này, đồng thời xóa sự có mặt của giả sử F và các dẫn xuất từ F.

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G] |

Thiết lập current goal là H∨G[G], thêm công thức G vào *list\_proof* và đồng thời biến H∨G[G] thành H∨G.

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 6. G assumption | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G]  F∨G→H∨G, H∨G , H∨G |

Ở bước tiếp theo ta đi tìm sự có mặt H ∨ G trong *list\_proof,* dễ dàng nhận ra H ∨ G chưa có mặt trong *list\_proof.* Theo tuần tự của giải thuật, ở bước này module **elimination** sẽ được gọi. Và kết quả **elimination** thất bại. Do đó, công thức H ∨ G sẽ được **introduction** tác động lên. H ∨ G ở dạng OR nên sẽ được “rã” theo **OR** **introduction**. Đây là lần “rã” đầu tiên nên việc “rã” diễn ra như sau:

* + Thêm công thức H vào *list\_goal.*
  + Đánh dấu H ∨ G đã được rã 1 lần

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 6. G assumption | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G]  F∨G→H∨G, H∨G , H∨G  F∨G→H∨G, H∨G , H∨G , H |

Thiết lập current goal là H. Theo giải thuật, sẽ đi tìm sự có mặt của H có trong *list\_proof*  hay chưa? Kết quả H chưa có mặt trong *list\_proof.* Tiến hành áp dụng **elimination** lên tập *list\_proof.*  Kết quả thất bại do đó công thức H sẽ được **introduction**. Do H không phải là các dạng công thức phức tạp nên việc **introduction** sẽ phản chứng. Tức là thêm ⊥ vào *list\_goal* và ¬ H vào *list\_proof.* Thiết lập current goal là ⊥. Bước tiếp theo, đi tìm sự có mặt của ⊥ . Theo định nghĩa sự có mặt, chúng ta phải đi tìm 1 cặp ¬A và A nhưng trong trường hợp này *list\_proof*  không thỏa mãn điều kiện này. Theo giải thuật, **contradiction** sẽ được gọi trong bước này. Dựa vào tập  *list\_proof*  F → H sẽ được **contradiction**. Theo module **contradiction**, F sẽ được thêm vào *list\_goal.*

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 6. G assumption 7. ¬ H assumption | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G]  F∨G→H∨G, H∨G , H∨G  F∨G→H∨G, H∨G , H∨G , H  F∨G→H∨G, H∨G , H∨G , H , ⊥  F∨G→H∨G, H∨G , H∨G , H , ⊥ , F |

Thiết lập current goal là F, bước tiếp theo sẽ là tìm sự có mặt của F trong *list\_proof.*  Kết quả là thất bại. Tiến hành **introduction** F, tức là thêm ⊥ vào *list\_goal,*  thêm giả sử ¬F vào *list\_proof.*

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 6. G assumption 7. ¬ H assumption 8. ¬ F assumption | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G]  F∨G→H∨G, H∨G , H∨G  F∨G→H∨G, H∨G , H∨G , H  F∨G→H∨G, H∨G , H∨G , H , ⊥  F∨G→H∨G, H∨G , H∨G , H , ⊥ , F  F∨G→H∨G, H∨G , H∨G , H , ⊥ , F , ⊥ |

Thiết lập current goal là ⊥. Theo giải thuật, đi tìm sự có mặt của ⊥ trong *list\_proof.*  Dễ dàng nhận ra trong *list\_proof*  không chứa 1 cặp công thức ¬A và A. Bước tiếp theo sẽ là contradiction tập *list\_proof.*  Kết quả việc áp dụng **contradiction** là thất bại nên sẽ quay về bước “rã” OR gần nhất. Theo bài toán, bước “rã” OR gần nhất là bước “rã” H ∨ G. Xóa hết những dẫn xuất hay công thức đã được thêm vào *list\_goal*  hay *list\_proof*  trong quá trình chứng minh của lần “rã” thứ nhất. Áp dụng lần “rã” thứ 2 cho H ∨ G ta được:

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 6. G assumption | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G]  F∨G→H∨G, H∨G , H∨G  F∨G→H∨G, H∨G , H∨G , G |

Thiết lập current goal là G. Dễ dàng nhận ra G có mặt trong *list\_proof.*  Xóa G ra khỏi *list\_goal.* Thiết lập current goal là H ∨ G. H ∨ G có *pending*  bằng 1. Tức là H ∨ G được chứng minh khi có G. Áp dụng module **updating** ta có bảng sau:

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 6. G assumption 7. H ∨ G ∨i 6 | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G]  F∨G→H∨G, H∨G , H∨G  F∨G→H∨G, H∨G , H∨G , G  F∨G→H∨G, H∨G |

Thiết lập current goal vào H ∨ G. H ∨ G có *pending* là 2. Gọi **updating** ta được:

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 6. ~~G~~ assumption 7. ~~H ∨ G~~  ∨i 6 8. H ∨ G ∨e 2,3,6 | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G]  F∨G→H∨G, H∨G , H∨G  F∨G→H∨G, H∨G , H∨G , G  F∨G→H∨G, H∨G  F∨G→H∨G |

Thiết lập current goal là F∨G→H∨G. F∨G→H∨G có *pending*  bằng 2. Gọi **updating** lên F∨G→H∨G ta có được

|  |  |
| --- | --- |
| List\_proof | List\_goal |
| 1. F→H 2. F ∨ G assumption 3. ~~F~~  assumption 4. ~~H~~ →e 1,3 5. ~~H ∨ G~~  ∨i 4 6. ~~G~~ assumption 7. ~~H ∨ G~~  ∨i 6 8. H ∨ G ∨e 2,3,6 9. F∨G→H∨G →i 2,8 | F∨G → H∨G  F∨G→H∨G, H∨G  F∨G→H∨G, H∨G , H∨G[G], H∨G[F]  F∨G→H∨G, H∨G , H∨G[G], H∨G , H  F∨G→H∨G, H∨G , H∨G[G], H∨G  F∨G→H∨G, H∨G , H∨G[G]  F∨G→H∨G, H∨G , H∨G  F∨G→H∨G, H∨G , H∨G , G  F∨G→H∨G, H∨G  F∨G→H∨G |

Vì F∨G→H∨G là goal ban đầu nên giải thuật kết thúc.

**CHƯƠNG IV**

**ĐÁNH GIÁ VÀ KẾT LUẬN**

**PHỤ LỤC**

**TÀI LIỆU THAM KHẢO**