Suy luận tự nhiên (Natural deduction calculi) , được phát triển vào giữa những năm 30 của thế kỷ trước bởi Gentzen và Jaskowski , chúng ta 2 phương pháp suy luận đó tương ứng với G-style và J-style. Trong phạm vi đề tài chúng tôi đề cập đến J-style mà chính xác hơn là F-style do nhà logic học người Mỹ Fitch cải tiến từ J-style.

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…

Ý tưởng chính của giải thuật là việc nắm giữ 2 danh sách công thức lst\_proof và lst\_goal . Trong đó lst\_proof là các công thức đã được dẫn xuất, còn lst\_goal nắm giữ các công thức cần phải suy luận. Lst\_goal ban đầu chỉ là công thức cần chứng minh, nhưng được biến đổi trong quá trình chứng minh. Lst\_proof ban đầu chỉ bao gồm các công thức lấy được từ giả thiết. Trong quá chứng minh ở một thời điểm bất kỳ lst\_proof và lst\_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 lst\_proof và lst\_goal, đồng thời các hoạt động đó sẽ được điều khiển bởi current\_goal ( goal được chọn trong lst\_proof). Tại mỗi bước của giải thuật , current\_goal sẽ được kiểm tra nó có mặt trong lst\_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 Pn mà tồn tại một phép đồng nhất giữa current\_goal và P
* Current\_goal là \_|\_ và chúng ta được 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 nếu 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, chúng ta mới áp dụng các luật introduction. Nếu công thức hiện tại không có mặt , chúng ta sẽ tiến hành thay đổi lst\_proof và lst\_goal dựa vào current\_goal là lst\_proof. Chúng tôi sẽ trình bày rõ ràng hơn phần 3.1

Nhắc lại hệ thống suy luận tự nhiên ND

A |- B

Hệ thống được chứng minh dựa vào lst\_proof và lst\_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:

Initiation:

Module này có chức năng dùng để xây dựng lst\_proof và lst\_goal sau khi xWAM đã được tạo ra.

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

Forward proof

Module này có chức năng tạo ra những những dẫn xuất mới từ những dẫn xuất trong lst\_proof . Module này cố gắng áp dụng các luật eliminate vào 1 hay 1 cặp công thức trong lst\_proof. Các dẫn xuất mới sẽ được thêm vào lst\_proof. Việc tìm kiếm các công thức hay cặp công thức sẽ được hoạt động theo breadth first search. Lst\_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 hiện tại

{A&B, A->C, B->!!E}|- {Gn}

Áp dụng forward proof lên tập lst\_proof của bài toán sẽ được bài toán mới :

1. {A&B, A->C, B->!!E, A, B}|- {Gn}
2. {A&B, A->C, B->!!E, A, B,C}|- {Gn}
3. {A&B, A->C, B->!!E, A, B,C,!!E}|- {Gn}
4. {A&B, A->C, B->!!E, A, B,C,!!E,E}|- {Gn}

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

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

Backward proof

Khi Elimination không thể áp dụng được nữa và current\_goal chưa có mặt trong lst\_proof. Chúng ta phải “rã” current\_goal ra thành nhiều goal mới, việc rã này có thể ảnh hưởng đến lst\_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:

1. Gn = A &B
   1. 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 lst\_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
   2. {P0,P1,P2…Pn}|-{G0,G1,G2…,A&B(2),A,B}

…

A

…

B

A & B

…..

1. Gn = A ->B
   1. Để chứng minh A->B ,chúng ta phải chứng minh B với giả thiết A . Do đó việc cập nhật lst\_proof và lst\_goal được thực hiện như sau: Thêm vào lst\_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. Thêm vào lst\_goal , một công thức mới là B và thiết lập thuộc tính pending cho A->B là 1.
   2. Bài toán thành {P0,P1,P2…Pn, A(\*)}|-{G0,G1,G2…,A->B(1),B}

…

If A

..

..

Nif B

A->B

…

1. Gn = A | B
   1. 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
         1. Thêm công thức A vào lst\_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
         2. Bài toán thành {P0,P1,P2…Pn}|-{G0,G1,G2…,A|B(1)(1),A}

..

A

A|B

….

* + 1. Nếu A|B rã lần thứ hai:
       1. Thêm công thức B vào lst\_goal , 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
       2. Bài toán thành {P0,P1,P2…Pn}|-{G0,G1,G2…,A|B(1)(2),B}

…

B

A|B

…

1. Gn = !A
   1. Để chứng minh được !A , chỉ có thể hoặc !A có mặt hoặc chứng minh \_|\_ với điều kiện giả thiết A. Đây là lối chứng minh phản chứng. Do đó lst\_goal và lst\_proof được cập nhật như sau:
      1. Thêm \_|\_ vào lst\_goal, thiết lập thuộc tính pending của !A là 1
      2. Thêm công thức A vào lst\_proof, đánh dấu A là giả sử
         1. {P0,P1,P2…Pn,A}|-{G0,G1,G2…,!A,\_|\_}

If A

..

..

Nif \_|\_

!A

1. Gn = all x f(x)
   1. Để chứng minh all x f(x), chúng ta luôn phải tìm một f(x0) với giả thiết x0 bất kỳ. Do đó lst\_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 . Lst\_goal sẽ được thêm vào công thức f(x0), và đồng thời thiết lập thuộc tính pending của all f(x) là 1

{P0,P1,P2…Pn, x0}|-{G0,G1,G2…,all xf(x),f(x0) }

If x0

..

..

Nif f(x0)

All x f(x)

1. Gn = exists xg(x)
   1. Để chứng minh exists 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 lst\_goal được cập nhật. Thêm f(α) vào lst\_goal với α được thiết lập thuộc tính ANY\_VALUE , cập nhật thuộc tính pending của exist x f(x) là 1

{P0,P1,P2…Pn, x0}|-{G0,G1,G2…,all xf(x),f(x0) }

..

F(α)

Exists f(α)

..

1. 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 lst\_proof !F , đánh dấu !F là giả sử đồng thời thêm vào lst\_goal \_|\_. Khi đó thuộc tính pending của F là 1.

{P0,P1,P2…Pn, x0}|-{G0,G1,G2…,all xf(x),f(x0) }

If !F

…

…

Nif \_|\_

F

Hoping proof:

Xét bài toán:

Xét bài toán hiện tại {P0,P1,P2 … Pn} |- {G0,G1,G2..Gn-1,\_|\_}

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

1. Pk = A->B
   1. Nếu chứng minh được công thức A, khi đó A, A->B có thể áp dụng -> rule.

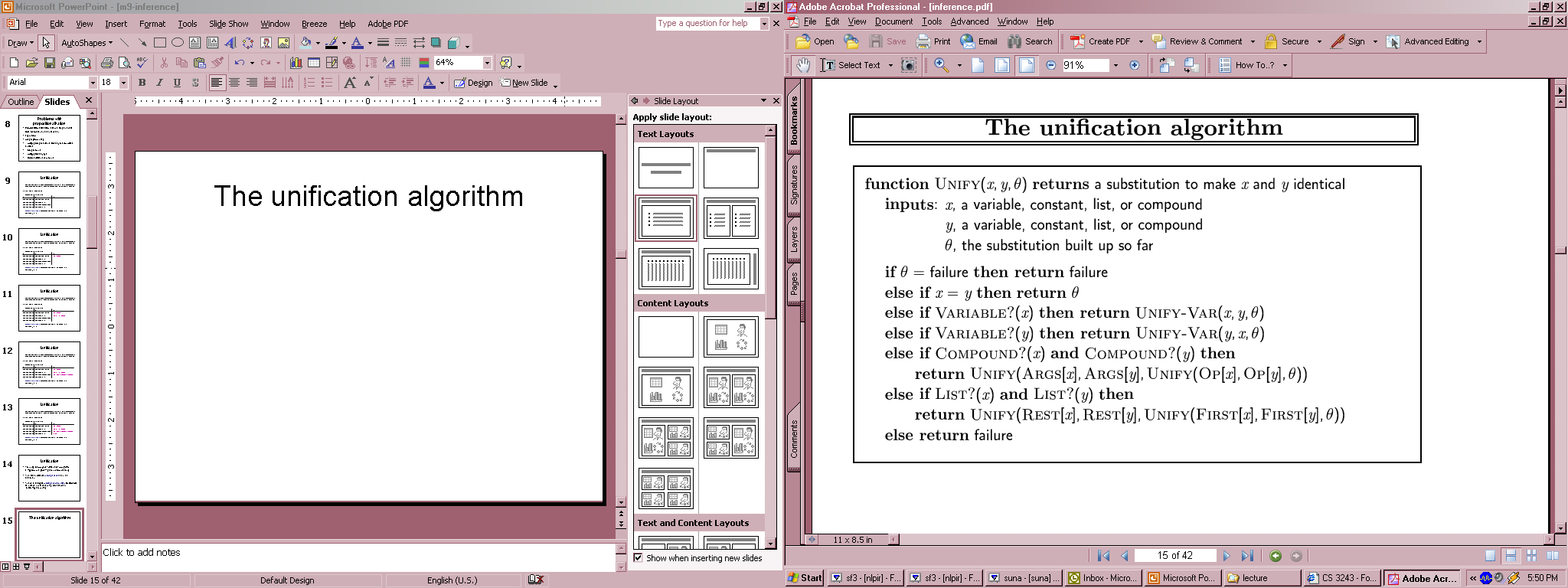
Do đó , hoping goal trong trường hợp này là A . Thêm A vào lst\_goal , thiết lập A là hoping\_goal, đồng thời đánh dấu A->B đã được tác động lên

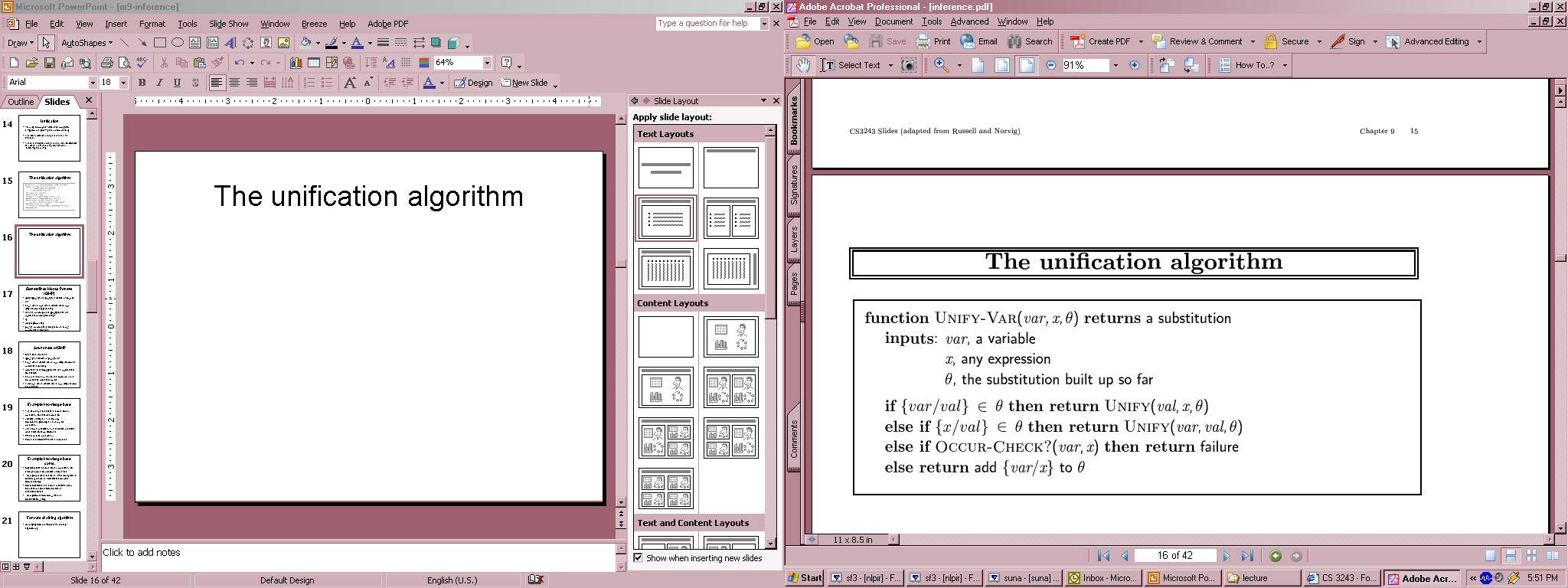
1. Pk = !A
   1. Nếu chứng minh được công thức A, khi đó !A, A có thể áp dụng -> rule.

Do đó , hoping goal trong trường hợp này là A . Thêm A vào lst\_goal , thiết lập A là hoping\_goal, đồng thời đánh dấu !A đã được tác động lên

Unification:

Chúng ta sử dụng unificaiton để tìm tập thay thế của 2 công thức x, y

.



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 .

Giải thuật chính :

Ban đầu chúng ta khởi tạo lst\_proof và lst\_goal, lst\_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 lst\_proof. Nếu công việc này thành công, chúng ta sẽ quay lại kiểm G0 có mặt trong lst\_proof hay chưa? Nếu lst\_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…

Trường hợp đặc biệt:

1. Eliminate OR:

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 đo để áp dụng eliminate OR vào A|B thì chúng ta A->C, B->C. Khả năng tìm được A->C, B->C trong lst\_proof là rất thấp .Vì thế , thay vì chỉ tác động lên lst\_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 Gn}

Để áp dụng eliminate OR lên A|B với mục tiêu đi chứng minh Gn. Chúng ta thêm A->Gn, B->Gn vào lst\_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 Gn, Gn[A].Gn[B] }.

A|B

If A

…

Nif C

If B

…

Nif C

C

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 :

A|B

If A

Nif D

If B

Nif D

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:

F->H

If F|H

If F

Nif H|G

If G

Nif H|G

H|G|R

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

F->H

If F|H

If F

Nif H|G|R

If G

Nif H|G|R

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

1. Eliminate exists:

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 exists x f(x) -> G bằng cách áp dụng luật exists eliminate.

Mô hình

If x 0 f(x0)

Nif G

G

Xét bài toán {} |- {}

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 lst\_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(x) 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 :

{Exists x f(x), all x f(x) -> G } |- {G|H}

Bài chứng minh

If xo f(xo)

F(x0) ->G

Nif G

G|H

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

Áp dụng của giải thuật tìm kiếm chứng minh:

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

1. Thí dụ F ->H |- F|G -> H|G

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

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H | Go = {F|G -> H|G} |

Tiếp theo chúng ta thực hiện elimination cho tập proof. Việc eliminate thất bại vì có F->H nhưng không tìm thấy F.

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

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| 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 thiếtF|G sẽ được thêm vào lst\_proof và H|G sẽ được thêm vào lst\_goal . G0 sẽ được thiết lập thuộc tính pending là 1.

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. 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 lst\_proof hay chưa? Kết quả H|G chưa có mặt trong lst\_proof.

Chúng ta tiến hành thực thi module elimination lên tập lst\_proof. Lúc này lst\_proof có 2 phần tử F->H và F|H . 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 lst\_goal và lst\_proof như sau. Lst\_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.

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. F|G ass | Go = {F|G -> H|G}  {F|G->H|G, H|G}  {F|G->H|G, H|G , H|G[F], H|G[G]} |

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

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. F | G ass 2. F ass | Go = {F|G -> H|G}  {F|G->H|G, H|G}  {F|G->H|G, H|G , H|G[F], H|G[G]}  {F|G->H|G, H|G , H|G[F], 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 lst\_proof. Tiến hành áp dụng elimination lên lst\_proof. F|H áp dụng thành công elimination nhờ có F. Do đó công thức H sẽ được thêm vào lst\_proof

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. F | G ass 2. F ass 3. H | Go = {F|G -> H|G}  {F|G->H|G, H|G}  {F|G->H|G, H|G , H|G[F], H|G[G]}  {F|G->H|G, H|G , H|G[F], 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 lst\_proof nên phải “rã” current goal. Việc rã sẽ được thực hiện theo OR Introduction . Do H|G rã lần đầu tiên nên lst\_goal sẽ thêm vào công thức thứ nhất là H

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. F | G ass 2. F ass 3. H ->e,.. | Go = {F|G -> H|G}  {F|G->H|G, H|G}  {F|G->H|G, H|G , H|G[F], H|G[G]}  {F|G->H|G, H|G , H|G[F], H|G}  {F|G->H|G, H|G , H|G[F], H|G,H} |

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

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. F | G ass 2. F ass 3. H ->e,.. | Go = {F|G -> H|G}  {F|G->H|G, H|G}  {F|G->H|G, H|G , H|G[F], H|G[G]}  {F|G->H|G, H|G , H|G[F], H|G}  {F|G->H|G, H|G , H|G[F], 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. Lúc introduction , tên ND rule đã được thiết lập cho H|G là |I nên dễ dàng có được H|G trong lst\_proof

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. F | G ass 2. F ass 3. H ->e,..   H|G |I .. | Go = {F|G -> H|G}  {F|G->H|G, H|G}  {F|G->H|G, H|G , H|G[F], H|G[G]}  {F|G->H|G, H|G , H|G[F], H|G}  {F|G->H|G, H|G , H|G[F], H|G} |

Loại bỏ H|G ra khỏi lst\_goal. Ở bước này , đồng thời xóa sự có mặt của giả sử H.

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. F | G ass 2. F ass 3. H (-) ->e,..   H|G |I .. | Go = {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[F]} |

Thiết lập current goal là H|G[F], thêm công thức G vào lst\_proof và đồng thời biến H|G[F] thành H|G.

|  |  |
| --- | --- |
| Lst\_proof | Lst\_goal |
| F->H   1. F | G ass 2. F ass 3. H (-) ->e,..   H|G |I ..  H | Go = {F|G -> H|G}  {F|G->H|G, H|G}  {F|G->H|G, H|G , H|G[F], H|G[G]}  {F|G->H|G, H|G , H|G[F], H|G}  {F|G->H|G, H|G , H|G[F]} |