

EXPTIME Tableaux with Global Caching for ALC

© Rajeev Goré
Computer Sciences Laboratory
The Australian National University
Australia

Rajeev.Gore@anu.edu.au

Linh Anh Nguyen
Institute of Informatics
University of Warsaw
Poland

nguyen@mimuw.edu.pl

5 December 2007

Overview

Description Logic ALC (aka multi-modal K) and Complexity

Naive Tableau Search for ALC and their Complexity

Blocking and Anywhere Blocking

Caching and Global Caching

Tableau Rules for Basic ALC

Propagation of Status

Example

Complexity, Soundness and Completeness

ALC + TBoxes = Multi-modal logic K_N + global assumptions

Modal Indices: $i ::= i_0 \mid i_1 \mid \dots \mid i_N$ for some positive fixed integer N

Atomic Formulae: $p ::= p_0 \mid p_1 \mid \dots$

Formulae: $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \supset \varphi \mid [i]\varphi \mid \langle i \rangle \varphi$

Semantics:
 $M, w \models [i]\varphi$ iff $\forall v \in W. R_i(w, v)$ implies $M, v \models \varphi$
 $M, w \models \langle i \rangle \varphi$ iff $\exists v \in W. R_i(w, v)$ and $M, v \models \varphi$

Global Assumptions: finite set Γ of formulae true in all worlds of all models

Global Logical Consequence: $\Gamma \models \varphi$ iff $\forall M. M \Vdash \Gamma \Rightarrow M \Vdash \varphi$

φ is a logical consequence of the assumptions Γ iff every model that makes Γ true **everywhere** also makes φ true **everywhere**

Complexity: general satisfiability problem is EXPTIME-complete

Goal: find an optimal algorithm

Naive Tableaux for Global Logical Consequence in ALC

NNF: assume negations appear only in front of atoms

Root Node and Problem Size: $Z := NNF(\Gamma \cup \neg\varphi)$ $n = |Z|$

Fisher-Ladner Closure: $FL(Z) := Sf(\neg Z)$ is finite and of $O(n)$

Classical Logic: $(\perp) \frac{X; p; \neg p}{\perp}$ $(\sqcap) \frac{X; \varphi \sqcap \psi}{X; \varphi; \psi}$ $(\sqcup) \frac{X; \varphi \sqcup \psi}{X; \varphi \mid X; \psi}$

Modal Rule: $(\exists) \frac{\exists R.\varphi; \forall R.X; Y}{\varphi; X; NNF(\Gamma)}$ i.e. $(\langle i \rangle) \frac{\langle i \rangle \varphi; [i] X; Y}{\varphi; X; NNF(\Gamma)}$

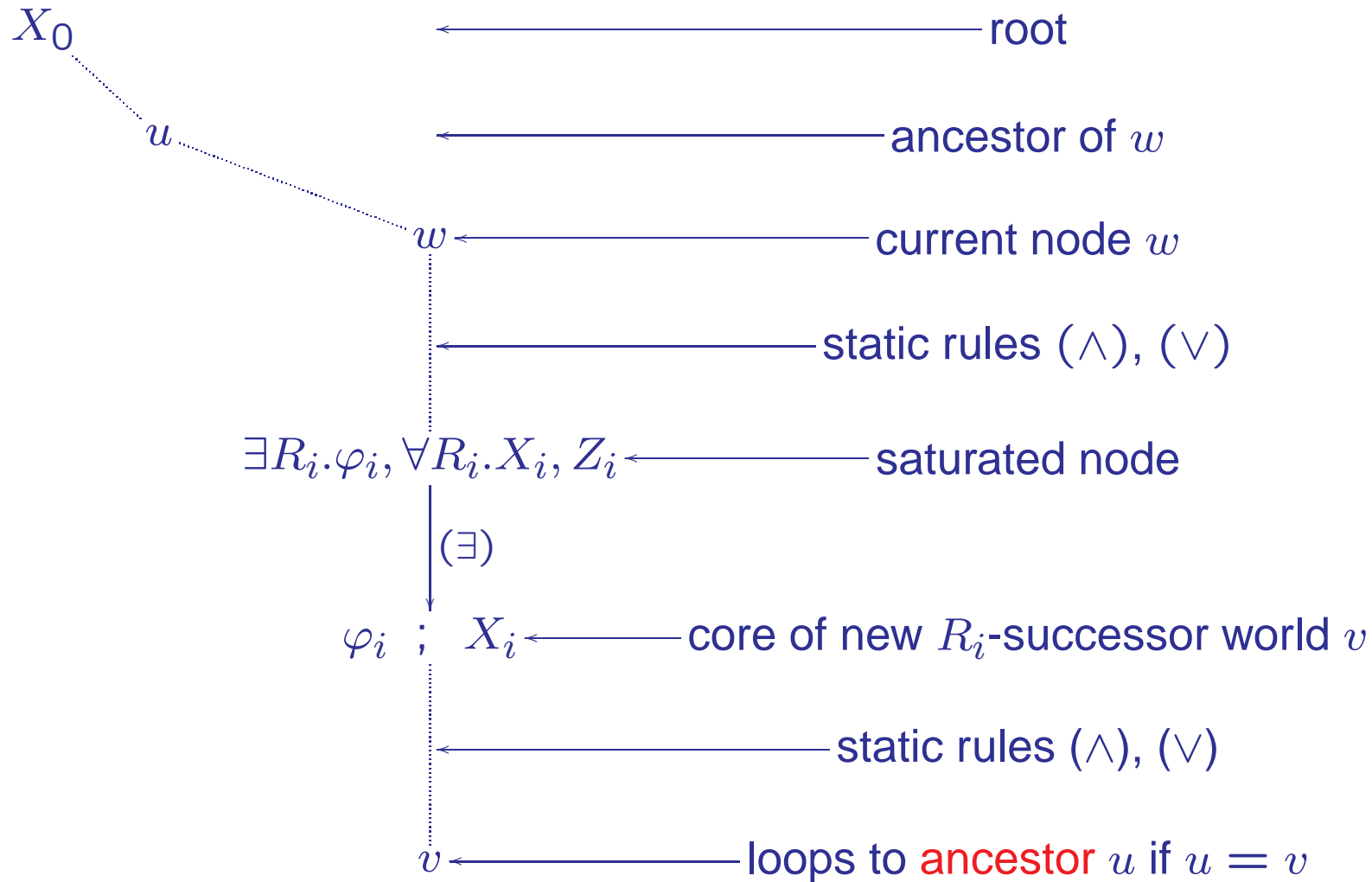
Note: Choose one (\exists) -child and do not keep track of edge-labels

Search Strategy: use depth first search

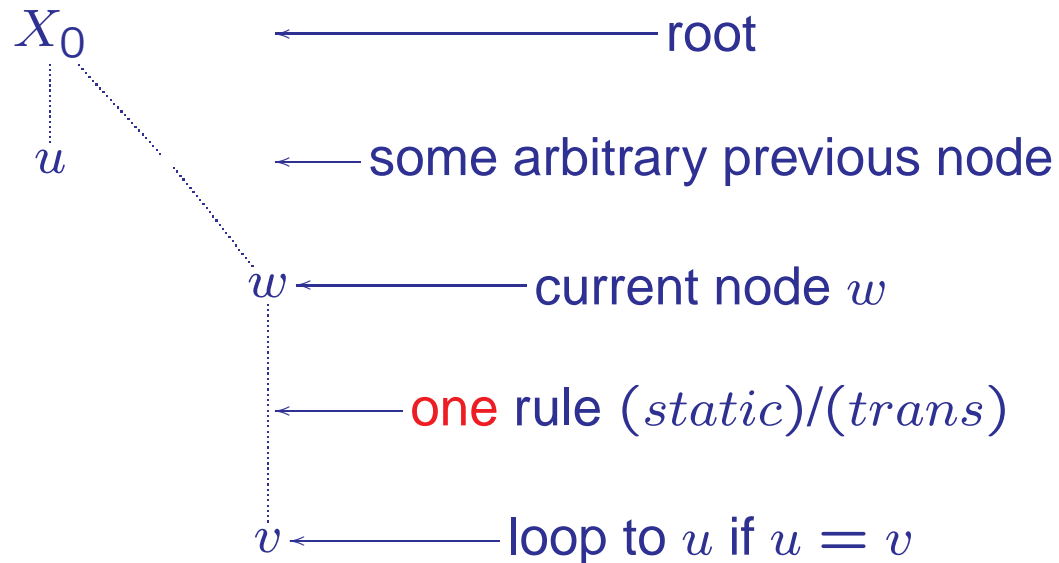
Blocking: termination by stopping when any node re-appears on branch

Complexity: worst-case behaviour is double-exponential i.e. $O(2^{2^n})$ because it evaluates the same nodes on multiple branches

Traditional Algorithm for DFS Proof Search Using Blocking

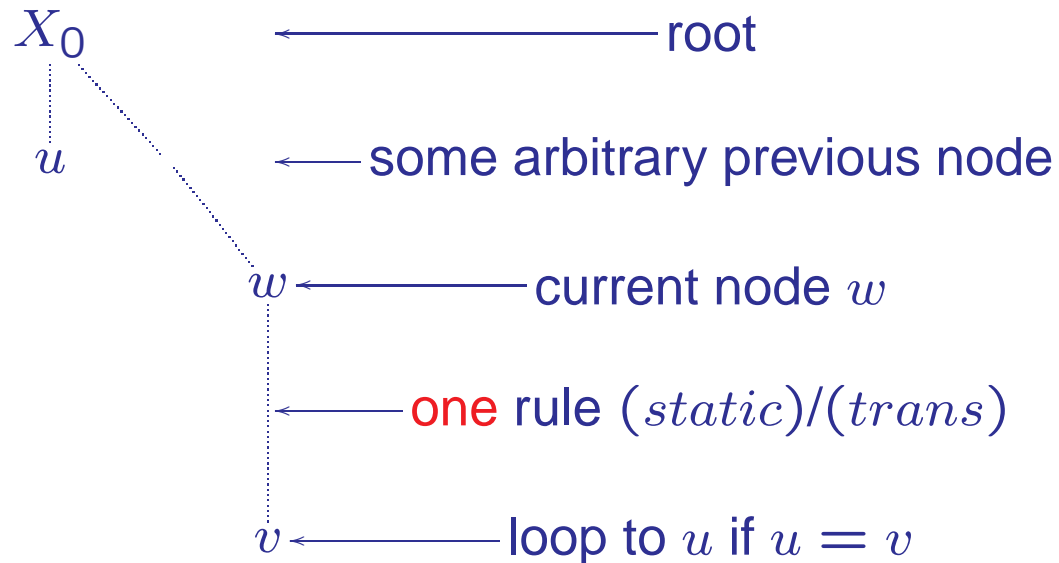


Proof Search Using Global Caching By Building graph G



That is, we allow cross-branch (anywhere) blocking ... but there is more of course :)

Proof Search Using Global Caching By Building graph G



That is, we allow cross-branch (anywhere) blocking

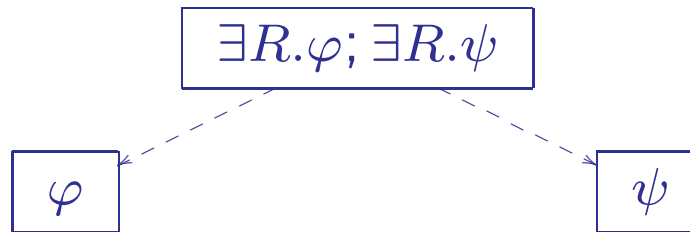
Big question: Is it safe ?

Tableaux, Models and And-Or-Structures

DFS: would like to use other strategies, even heuristic ones

Traditional Tableaux: are or-structures because of the branching arises from Or-rules which require both children to be closed

Models: are And-structures where **And-Branching** arises from **different** ways of applying a non-invertible rule and each Or-node has one open child



Search Space: is one And-Or-structure and a closed tableau is one such structure where each Or-node has all children closed and each And-node has some child closed

Status: of a node during search is “closed” (unsat), “open” (sat) or “unknown”

Caching of Nodes with Known Status is Easy

Cache: the contents X and status (sat or unsat) of nodes

Equality Check: before creating a new node with content X check if the cache contains $(X, \text{sat/unsat})$ and backtrack with $status := \text{sat/unsat}$

Subset Check: before creating a new node with content X check if the cache contains $(Y, status_Y)$ and backtrack with

$$status_X := \begin{cases} \text{sat} & \text{if } X \subseteq Y \text{ and } status_Y = \text{sat} \\ \text{unsat} & \text{if } X \supseteq Y \text{ and } status_Y = \text{unsat} \end{cases}$$

Better Caching: stores each created node X and its status as sat, unsat or “unknown”

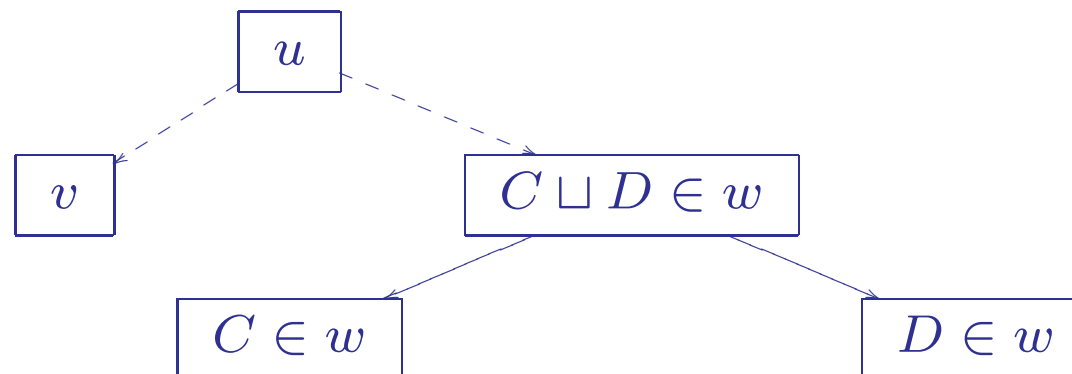
Caching Nodes With Status “unknown”

Basic Method: block expansion of node X if it duplicates an ancestor on the unique path back to the root (blocking)

More Advanced: block expansion of node X if it duplicates any node (including ancestors) in the current And-tree and discard when backtracking to new And-tree i.e. different OR-branch

Why discard? Because the copy of X may live on an OR-branch which we no longer follow so underlying model has changed

Donini and Massacci: “many potentially satisfiable sets of concepts are discarded when passing from a branch (And-tree) to another (And-tree)”



Global Caching

Global Caching: for each possible set X of formulae, at most one node with content X is created in the **search space** and this node is expanded **at most once** by the algorithm

Donini and Massacci: global caching “prunes heavily the search space but its unrestricted usage may lead to unsoundness ... It is conjectured that (global) caching leads to EXPTIME bounds but this has not been formally proved so far, nor the correctness of (global) caching has been shown ... because the caching optimisations are left out of the formal descriptions”

Our contribution: EXPTIME tableau procedure for ALC using **provably sound unrestricted global caching**

Practically: our algorithm is very easy to implement

Tableau Rules for ALC

Classical Logic: $(\perp) \frac{X; p; \neg p}{\perp}$ $\sqcap \frac{X; \varphi \sqcap \psi}{X; \varphi; \psi}$ $\sqcup \frac{X; \varphi \sqcup \psi}{X; \varphi \mid X; \psi}$

Modal Rule: $(\exists) \frac{\exists R. \varphi; \forall R. X; Y}{X; \varphi}$ i.e. $(\langle i \rangle) \frac{\langle i \rangle \varphi; [i] X; Y}{\varphi; X; \mathit{NNF}(\Gamma)}$

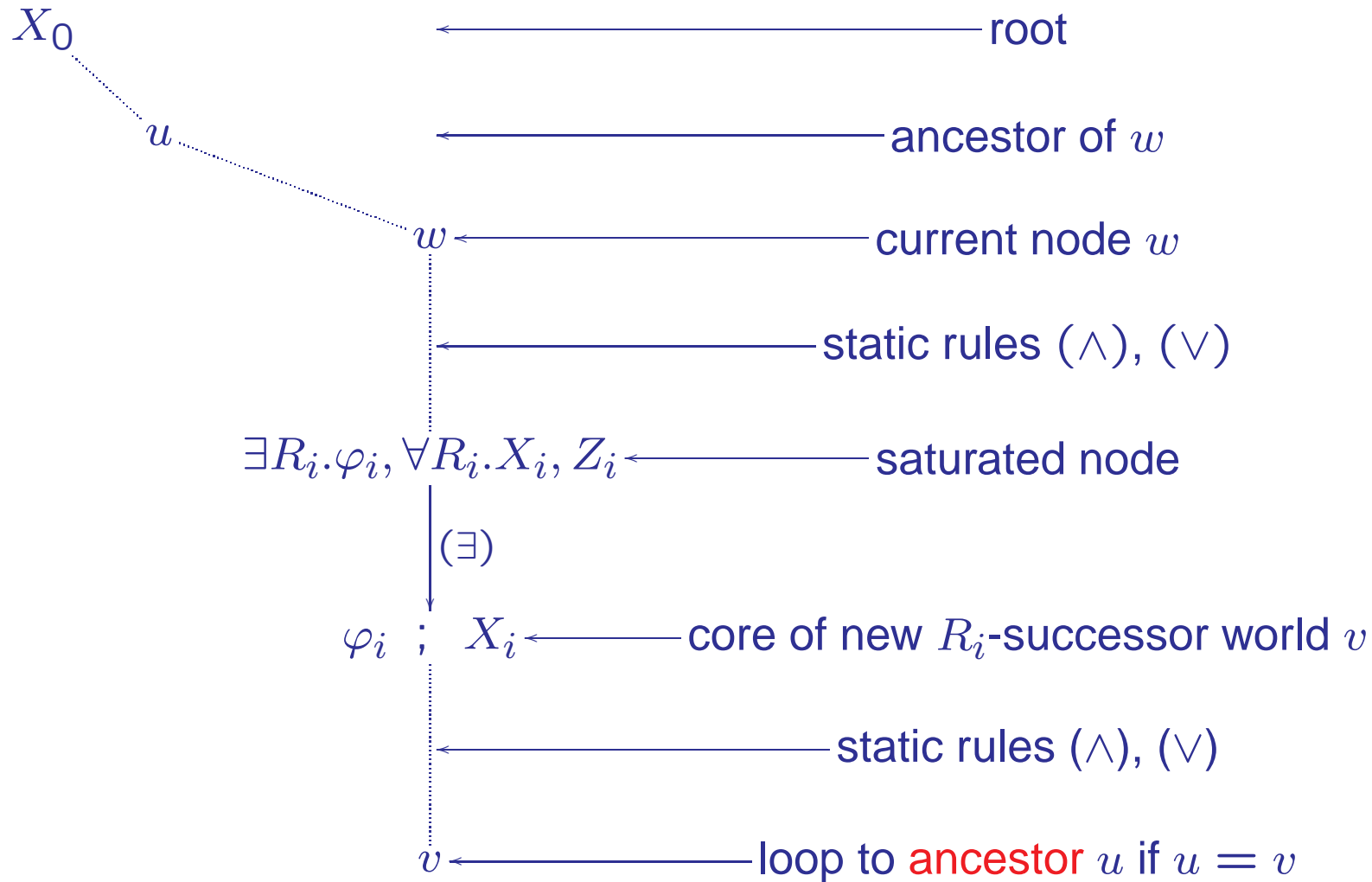
Top line: just captures classical logic in negation normal form

Next line: captures “transitional” modal rule

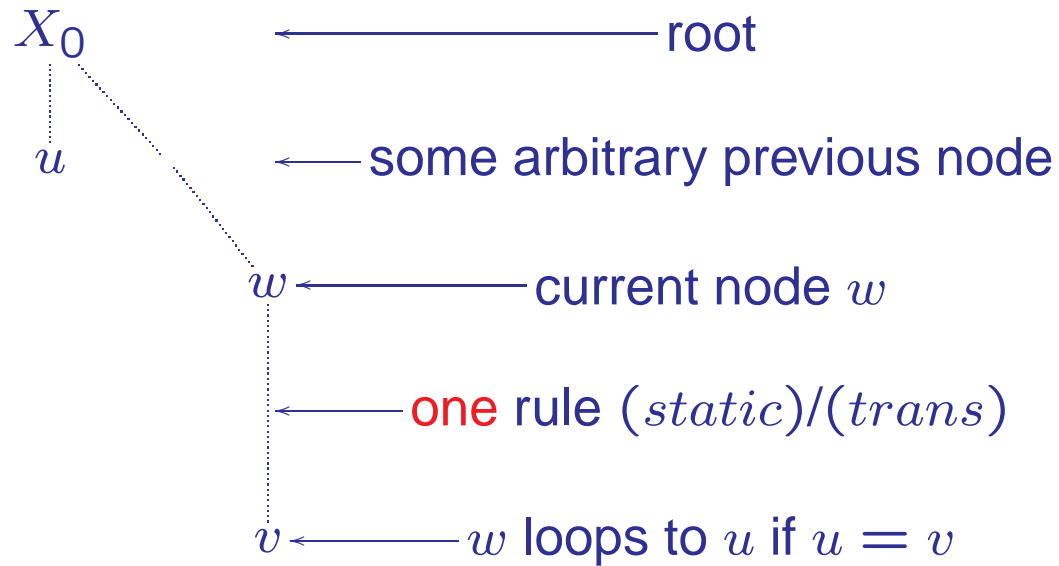
Note: Do not need to keep track of edge-labels e.g. R or i because we care only about node contents

Data Structure: Graph $G = \langle V, E \rangle$ of Vertices and unlabelled Edges with each vertex (node) having a status of unexpanded, expanded, sat, unsat where expanded and unexpanded correspond to “unknown”

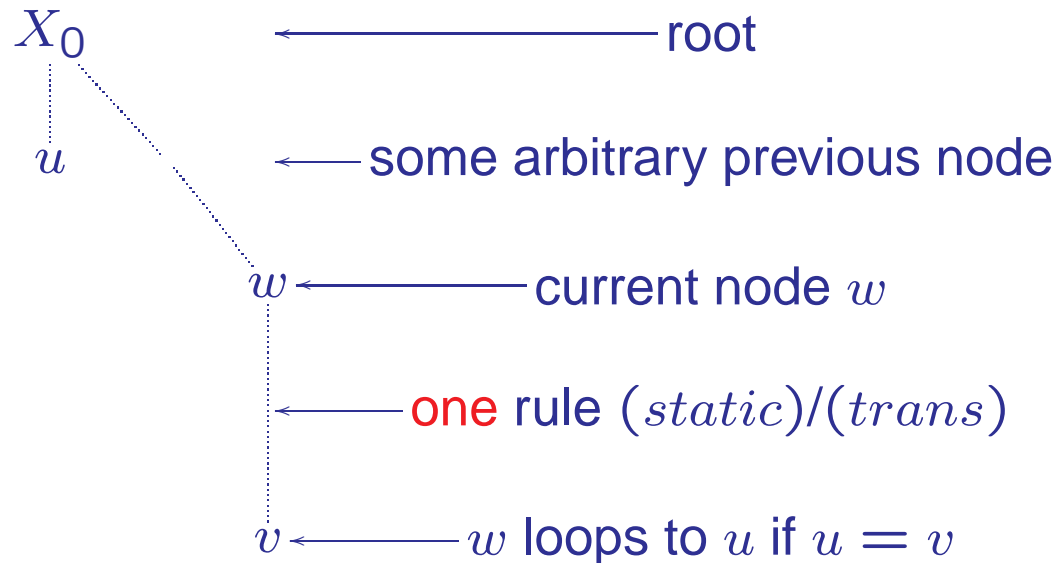
Traditional Algorithm for Proof Search Using Caching



Proof Search Using Global Caching By Building graph G

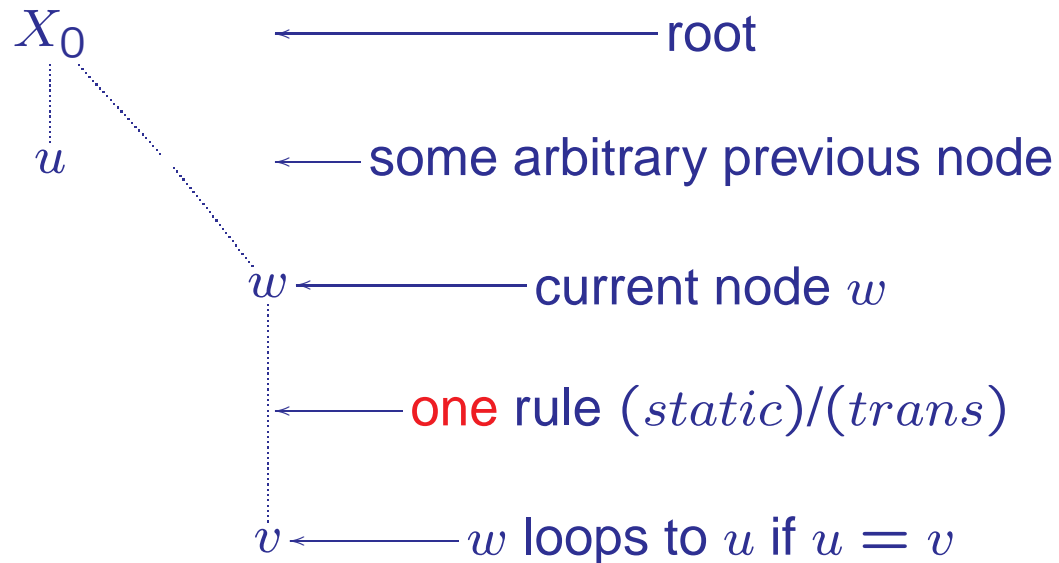


Proof Search Using Global Caching By Building graph G



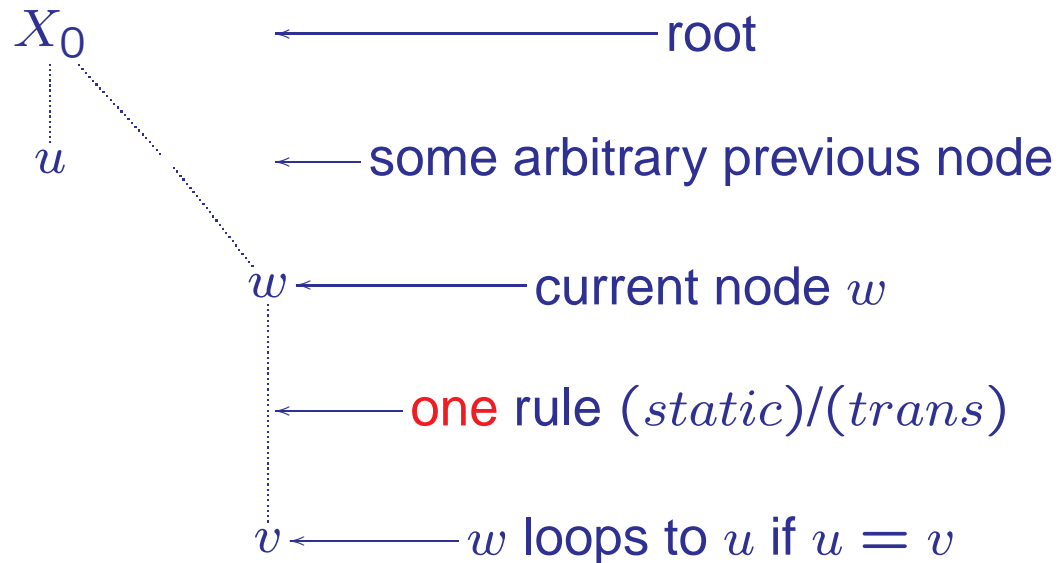
Builds an **and-or** graph (cache) G . The graph G contains each applied (static) rule denominator, not just saturated nodes as in the traditional algorithm.

Proof Search Using Global Caching By Building graph G



Each node appears only once because repetitions are represented by “cross-tree” edges to their first occurrence, so G has at most $2^{O(n^2)}$ nodes.

Proof Search Using Global Caching By Building graph G



$v.status = sat$ if no rule is applicable to v

(known status)

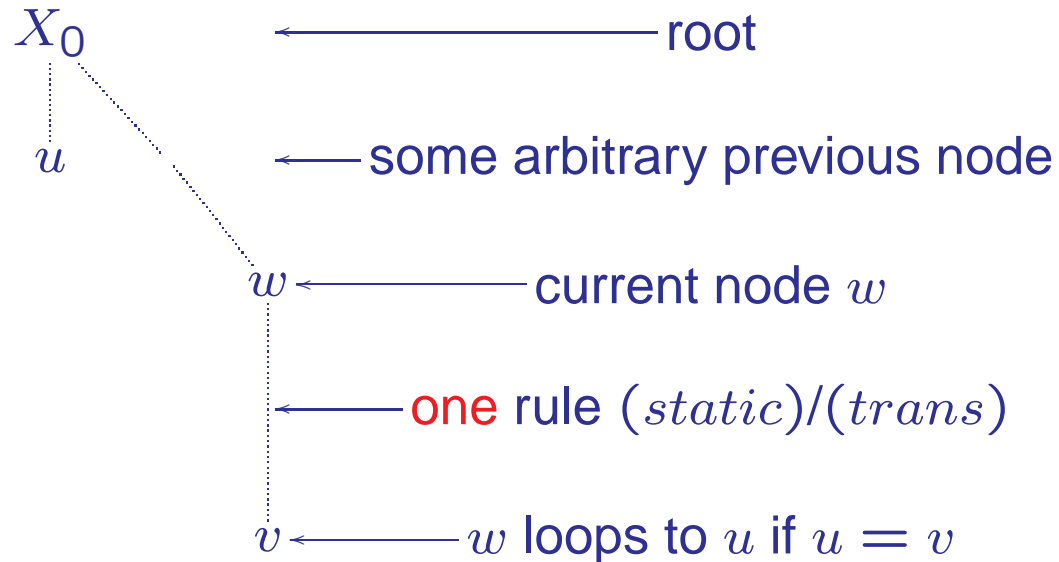
or $v.status = unsat$ if $\{A, \neg A\} \subseteq v$

(known status)

or $v = u$ and $v.status = u.status$

(unknown status)

Proof Search Using Global Caching By Building graph G



- $v.status = sat$ if no rule is applicable to v (known status)
- or $v.status = unsat$ if $\{A, \neg A\} \subseteq v$ (known status)
- or $v = u$ and $v.status = u.status$ (unknown status)

Whenever it determines that $v.status \in \{sat, unsat\}$ it calls $propagate(G, v)$ because $v.status$ is **irrevocable**

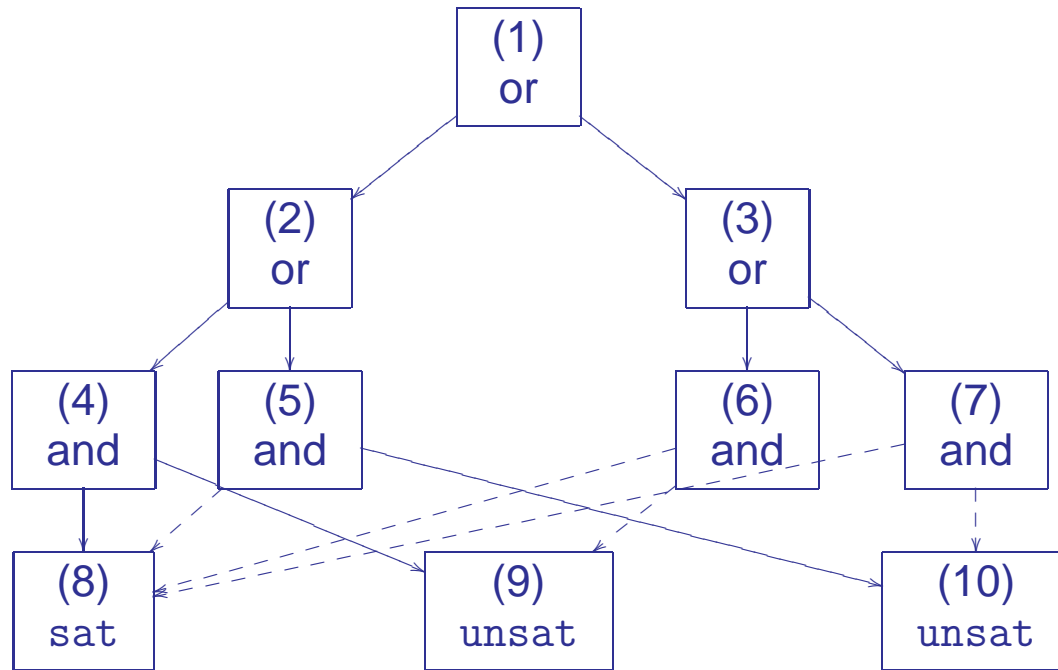
Procedure *propagate*(G, v)

Accepts an and-or graph $G = \langle V, E \rangle$ and $v \in V$ with

$v.status \in \{\text{sat}, \text{unsat}\}$, initialises $queue := \{v\}$ and returns a modified and-or graph $G = \langle V, E \rangle$ by propagating sat/unsat **repeatedly**

while $root.status \notin \{\text{sat}, \text{unsat}\}$ and $queue$ is not empty do

1. extract x from $queue$;
2. for every $u \in V$ with $(u, x) \in E$ and $u.status = \text{expanded}$ do
 - a. if ($u.kind = \text{or-node}$ and some child of u has status sat)
or ($u.kind = \text{and-node}$ and all children of u have status sat)
then $u.status := \text{sat}$, $queue := queue \cup \{u\}$
 - b. else if ($u.kind = \text{or-node}$ and all children of u have status unsat)
or ($u.kind = \text{and-node}$ and some child of u has status unsat) then
 $u.status := \text{unsat}$, $queue := queue \cup \{u\}$;



$\Gamma = \{A \sqsubseteq B \sqcap C\} = \{\neg A \sqcup (B \sqcap C)\}$	
Node	Content
(1)	$\neg A \sqcup (B \sqcap C); (\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$
(2)	$\neg A; (\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$
(3)	$B; C; (\exists R.A \sqcap \exists R.(A \sqcap \neg B)) \sqcup (\exists R.A \sqcap \exists R.(A \sqcap \neg C))$
(4)	$\neg A; \exists R.A; \exists R.(A \sqcap \neg B)$
(5)	$\neg A; \exists R.A; \exists R.(A \sqcap \neg C)$
(6)	$B; C; \exists R.A; \exists R.(A \sqcap \neg B)$
(7)	$B; C; \exists R.A; \exists R.(A \sqcap \neg C)$
(8)	$A; B; C$
(9)	$A; \neg B; B; C$
(10)	$A; \neg C; B; C$

Termination and Complexity

Finite Number of Different Nodes: All nodes are subsets of the finite and fixed set $FL(Z)$

Termination: Hence every branch must eventually loop or terminate

Size of $FL(Z)$: polynomial in $O(n)$

Size of G : There are at most $2^{O(n)}$ nodes in G

Propagate: called at most $2^{O(n)}$ times and might traverse G each time

Complexity: $2^{O(n)} \cdot 2^{O(n)} \subseteq 2^{O(n^2)}$

Tableau Rules for ALC

Classical Logic: $(\perp) \frac{X; p; \neg p}{\perp}$ $\Box \frac{X; \varphi \Box \psi}{X; \varphi; \psi}$ $\Box \frac{X; \varphi \Box \psi}{X; \varphi \mid X; \psi}$

Modal Rule: $\exists \frac{\exists R. \varphi; \forall R. X; Y}{X; \varphi}$

Top line: just captures classical logic in negation normal form

Next line: captures “transitional” modal rule

Note: Do not need to keep track of edge-labels e.g. R or S because we care only about node contents

Data Structures: Graph $G = \langle V, E \rangle$ of V ertices and unlabelled E edges with each vertex (node) having a status of unexpanded, expanded, sat, unsat where expanded and unexpanded correspond to “unknown”

The Algorithm

```
while  $root.status \notin \{sat, unsat\}$  and there is a node with status unexpanded do
  choose a node  $x$  with  $x.status = unexpanded$  using some strategy
  if (id) rule is applicable then put  $x.status := unsat$  and  $D := \emptyset$ 
  else if a static rule is applicable then apply it and obtain
    denominators  $D = \{d_1\}$  or  $D = \{d_1, d_2\}$ 
  else if the transitional rule is applicable then apply it in all possible
    ways to obtain denominators  $D = \{d_1, d_2, \dots, d_n\}$ 
  else (no rule is applicable so) put  $x.status := sat$  and  $D := \emptyset$ 
  if  $x.status$  is either sat or unsat then propagate(G, x)
  else for every denominator  $d \in D$  do
    if there is a (proxy) node  $y \in V$  with  $y.content = d.content$ 
    then insert an edge  $(x, y)$  into  $E$ 
      if  $y.status$  is sat or unsat then propagate(G, y)
    else add  $d$  to  $V$  and insert an edge  $(x, d)$  into  $E$ 
  if  $x.status = unexpanded$  then put  $x.status := expanded$ 
if  $root.status \notin \{sat, unsat\}$  then for all nodes with status expanded put status to sat
```

Is it safe?

Lemma: It is an invariant of our Algorithm that for every $v \in V$:

1. if $v.status = \text{unsat}$ then

- v contains both A and $\neg A$ for some atomic concept A ,
- or $v.kind = \text{and-node}$ and there exists $(v, w) \in E$ such that $w \neq v$ and $w.status = \text{unsat}$,
- or $v.kind = \text{or-node}$ and for every $(v, w) \in E$, $w.status = \text{unsat}$;

Proof Sketch: these are the only ways that a node gets status unsat

Is It safe?

Lemma: It is an invariant of Algorithm that for every $v \in V$:

1. if $v.status = sat$ then

- no ALC-tableau rule is applicable to v ,
- or $v.kind = or\text{-}node$ and there exists $(v, w) \in E$ with $w.status = sat$,
- or $v.kind = and\text{-}node$ and for every $(v, w) \in E$, $w.status = sat$.

Proof Sketch: a node can also get status `sat` by the last step of the algorithm. But if we get there then this node never got status `unsat`, and neither did the root node.

Soundness: `unsat` means closed

Soundness: Suppose $G = \langle V, E \rangle$ is constructed by our Algorithm for $NNF(\Gamma \cup \neg\varphi)$. Each $v \in V$ with status `unsat` is inconsistent w.r.t. Γ .

Proof Sketch: We have to construct a closed (tree) tableau for v .

Previous lemma about the invariant about `unsat` guides us

So start with node v and mimic the algorithm until it hits the cache or hits an (\exists) -node or the (tree) tableau finds a repeated ancestor

If algorithm hits a cached node w then copy w , thereby unwinding the graph into a tree (the rule is sound of course)

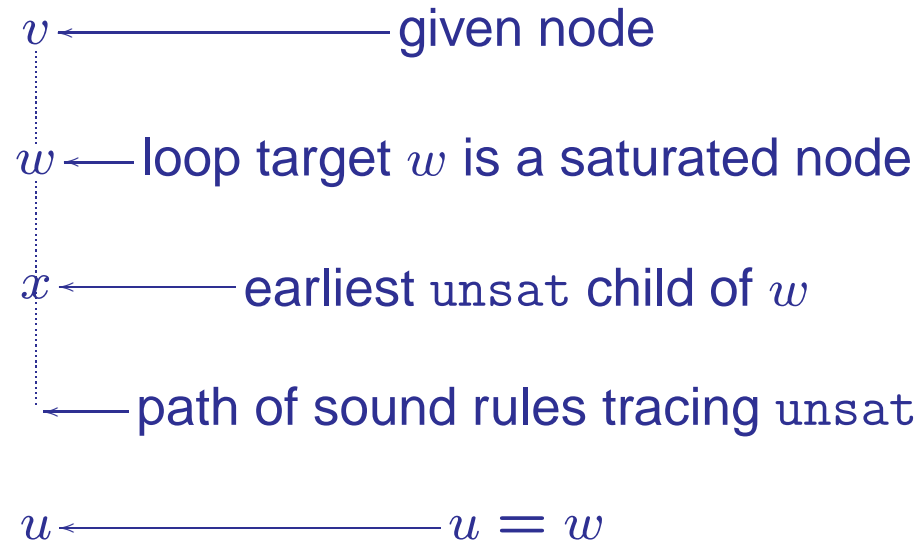
If algorithm hits an (\exists) -node then follow the child whose status became `unsat earliest`, this is a sound application of the original linear (\exists) -rule

If tree-tableau finds a repeated `saturated` node then stop else copy

Sooner or later, we know we will terminate the tree-tableau construction

Why can't we go into an ancestor loop? (time-stamp)

Why Unwinding Cannot Find an Ancestor Loop



Thus u gets its status of unsat from w .

Implies x gets its status of unsat from u .

Implies w gets its status of unsat before x .

But then w must inherit unsat from some child other than x

So x is not the earliest child of w to get unsat

Contradiction.

Completeness: sat means open

Lemma: Let $G = \langle V, E \rangle$ be the graph constructed by our Algorithm for $NNF(\Gamma \cup \neg\varphi)$. Every tree-tableau for $v \in V$ with status sat is open.

Proof Sketch: Choose any $v \in V$ with $v.status = \text{sat}$ and let T be an arbitrary tableau (tree) w.r.t. Γ for $v.fmlset$.

Let G guide us.

At each \forall -node the graph points us to at least one open child

At each (\exists) -node the graph tells us that every child is open

Sooner or later the tree-tableau branch from T will find a saturated ancestor and create a “good” ancestor loop

Yes it is safe!