Theory and Practice of the Tableau WorkBench

Pietro Abate
Laboratoire PPS
Université Paris Diderot - Paris 7
France
Pietro.Abate@anu.edu.au

Rajeev Goré
Computer Sciences Laboratory
The Australian National University
Australia
Rajeev.Gore@anu.edu.au

Overview

Examples: Tableau and Sequent Calculi for Classical Propositional Logic

Part 1 Theory: 45 minutes + 15 minute break with questions

Part 2: Practice: 45 minutes then 15 minutes break with questions

Part 3: Advanced Topics 45 minutes then 15 minutes break with questions

tabpc.ml: CPL Using Negation Normal Form

```
CONNECTIVES [ "~" ; "&" ; "v" ; "->" ; "<->" ]
GRAMMAR formula :=  ATOM | Verum | Falsum
                  | formula & formula  | formula v formula
                  | formula -> formula | formula <-> formula
                  | ~ formula ;;
        expr    := formula ;;
END
open Pclib
TABLEAU  RULE Id     { a } ; { ~ a } === Close END
         RULE False Falsum           === Close END
         RULE And    { A & B }       === A ; B END
         RULE Or     { A v B }       === A | B END
END
STRATEGY := tactic ( (False ! Id ! And ! Or)* )
PP := List.map nnf      (* PreProcess input formula into nnf *)
NEG := List.map neg     (* Negate input formula by default   *)
MAIN
```

seqpc.ml: Sequent Calculus for CPL

```
CONNECTIVES [ "~" ; "&" ; "v" ; "->" ; "<->" ; "=>"]
GRAMMAR
formula :=
     ATOM | Verum | Falsum
    | formula & formula
    | formula v formula
    | formula -> formula
    | formula <-> formula
    | ~ formula
;;

expr := formula ;;
node := set => set ;;
END
```

```
SEQUENT
  RULE Id    Close === { A } => { A }                              END
  RULE False Close === { Falsum } =>                               END
  RULE True  Close ===  => { Verum }                               END
  RULE AndL  A ; B => === { A & B } =>                             END
  RULE AndR  => A | => B === => { A & B }                          END
  RULE OrL   A => | B => === { A v B } =>                          END
  RULE OrR   => A ;  B ===  => { A v B }                           END
  RULE NegR  A =>  ===  => { ~ A }                                 END
  RULE NegL  => A  === { ~ A }  =>                                 END
  RULE ImpL  => A | B => === { A -> B } =>                         END
  RULE ImpR  A => B === => { A -> B }                              END
  RULE IffL  (A -> B) & (B -> A) => === { A <-> B } =>             END
  RULE IffR  => (A -> B) & (B -> A) === => { A <-> B }             END
END
```

```
let exit = function
    |"Open" -> "Not Derivable"
    |"Close" -> "Derivable"
    |s -> assert(false)

EXIT := exit(status@1)

STRATEGY := tactic ( ( Id
                       ! True ! False
                       ! NegL ! NegR
                       ! AndL ! AndR
                       ! ImpL ! ImpR
                       ! OrL  ! OrR
                       ! IffL ! IffR )* )

MAIN
```

Basic Theory of Tableau Calculi

Tableau: finite single-rooted tree of nodes

Nodes: typically sets (or multisets or lists) of formulae

Rules: $(\rho)\ \dfrac{N}{d_1 \cdots d_n}$ (rulename) $\dfrac{\text{Numerator}}{\text{denominators}}$ $\dfrac{\text{pattern}}{\text{constructors}}$

Rule Applicable: if numerator pattern matches contents of the current node

Rule Action: extends tableau by creating children of current node according to the pattern specified by the denominators

Examples: (And) $\dfrac{A \wedge B; X}{A; B; X}$ (Or) $\dfrac{A \vee B; X}{A; X \mid B; X}$

Examples: (And) rule extends the tableau by adding a single child while (Or) rule extends the tableau by adding two children causing or-branching

Basic Theory of Tableau Calculi Continued

Closed Branch: if the current node contains $A; \neg A$ or $\perp$

Closed Tableau: if every or-branch is closed

Open Branch: if no rule applicable to the current node

Open Tableau: if some or-branch is open

Example: $(Id)\ \dfrac{A; \neg A; X}{\times}$ $\qquad \dfrac{\text{pattern}}{\text{special mark to indicate closure}}$

Backtracking: if current node is closed or no rule is applicable then backtrack to last choice-point

Examples: (Or) $\dfrac{A \vee B; X}{A; X \mid B; X}$ so (Or) rule creates a choice-point

Different Forms of Non-Determinism (Choice-points)

Node Choice: which leaf of the current tableau is the current node?

Rule Choice: which of the applicable rules to apply to the current node?

Formula Choice: which formula is principal for the chosen rule application?

And-Branching: results from all three forms of non-determinism

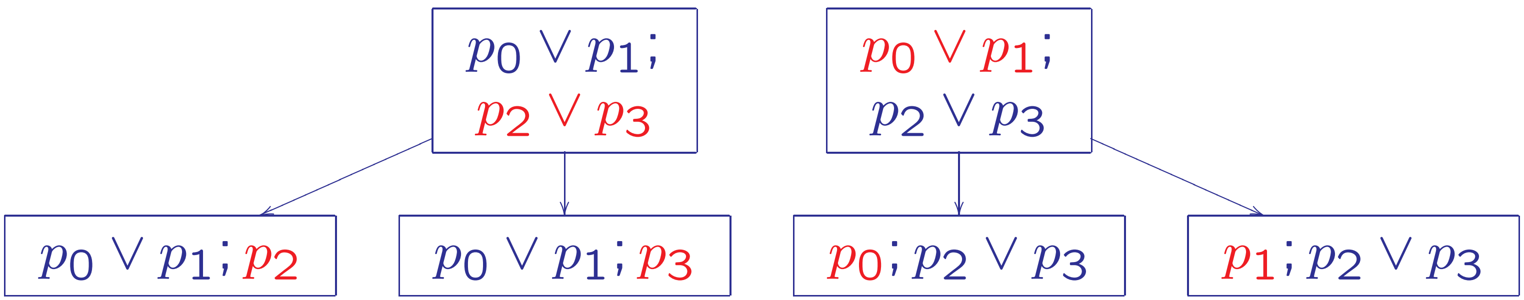
Traditional Tableaux: hide And-branching but show Or-branching explicitly

And-Branching from Formula Choice

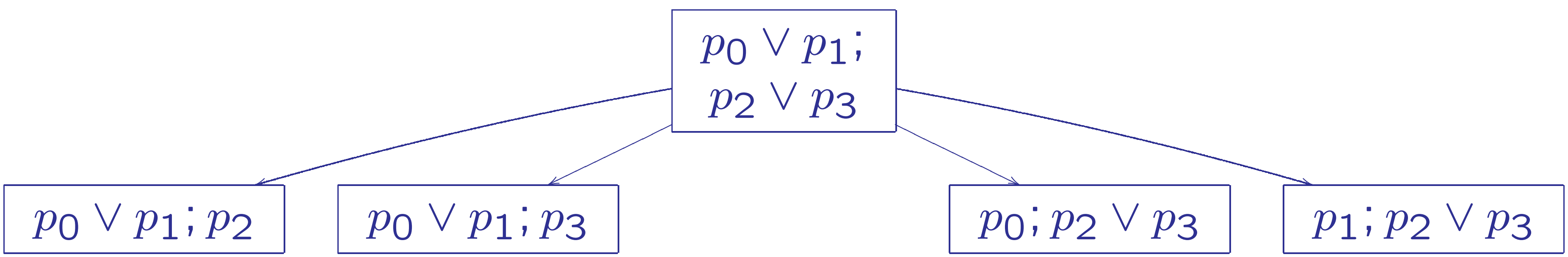
Formula Choice: which formula is principal for the chosen rule application?

Traditional Tableaux: hide And-branching but show Or-branching explicitly

Two Different Or-Trees: of which only one needs to close



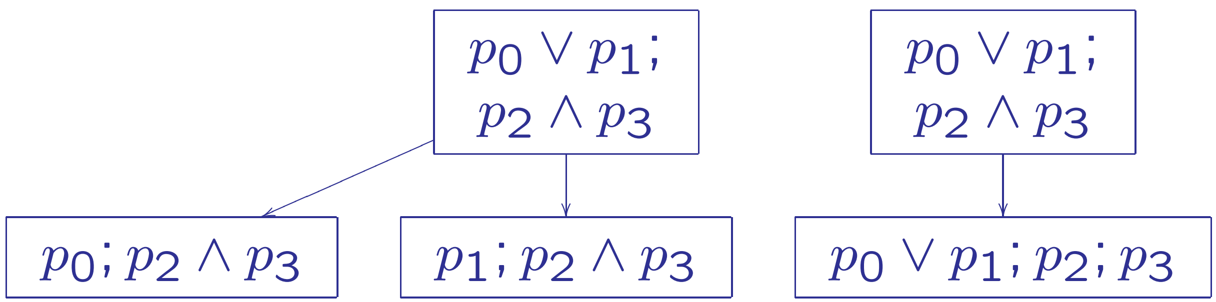
One And-Or-Tree: of which only one And-choice has to close



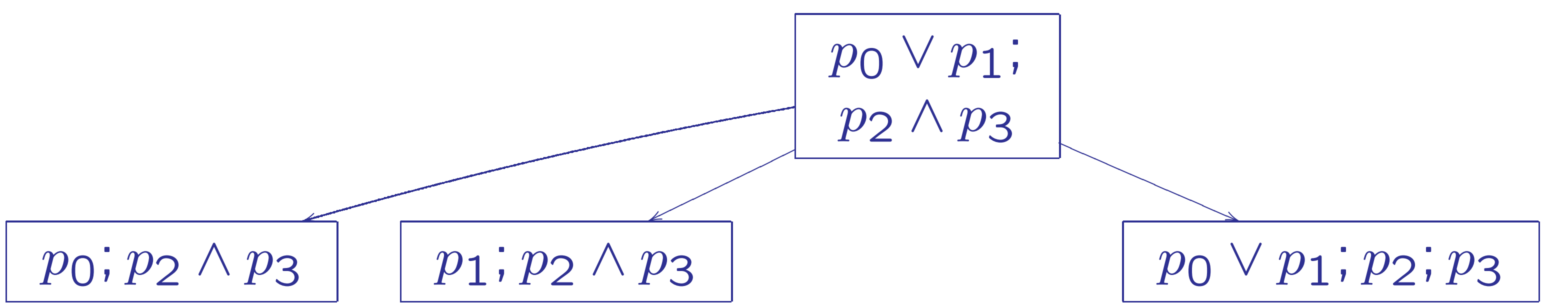
And-Branching From Rule Choice

Rule Choice: which of the applicable rules to apply to the current node?

Two Traditional Tableaux: of which only one needs to close



One And-Or Tableau: of which only one And-choice has to close



Controlling the Various Forms of Non-Determinism

Node Choice: depth-first search, breadth-first search, iterative deepening, heuristic choice (assume depth-first search)

Rule Choice: strategy for applying rules by sorting into invertible (static) and non-invertible rules (transitional)

Example: apply invertible rules in any order until none applicable,
then apply a non-invertible rule
and backtrack over the non-invertible rule choices if the chosen non-invertible rule does not close current branch

Formula Choice: rarely specified but can be done using more advanced techniques

A Tableau Calculus for CPL Using Negation Normal Form

NNF: implication-free formulae with negations appearing only in front of atoms

Theorem: every formula has a CPL-equivalent formula in nnf

Proof: use the distribution laws to push negations inwards

$(\varphi \to \psi) \leftrightarrow (\neg\varphi) \vee \psi \qquad \neg\neg\varphi \leftrightarrow \varphi \qquad \neg(\varphi \wedge \psi) \leftrightarrow (\neg\varphi) \vee (\neg\psi)$ etc.

Rules: (Id) $\dfrac{p; \neg p; X}{\times}$ $\quad (\wedge)$ $\dfrac{A \wedge B; X}{A; B; X}$ $\quad (\vee)$ $\dfrac{A \vee B; X}{A; X \mid B; X}$

Node Choice: depth first search (say)

Rule Choice: prefer (Id) over $(\wedge)$ over $(\vee)$to reduce branching and don't backtrack over rule choices since both $(\wedge)$ and $(\vee)$ are invertible

Formula Choice: don't backtrack over formula-choice points

End of Part 1

Defining Connectives

`CONNECTIVES` [semicolon separated list of double-quoted ASCII strings]

Keyword: `CONNECTIVES` must be in upper-case

Double-quotes: `" . "`

Order: is irrelevant

Example: `CONNECTIVES [ "~" ; "&" ; "v" ; "->" ; "<->" ]`

Exceptions: cannot use `\` `@` `|` `ATOM` `{` `}` `;;` as connectives

Defining Formulae and `expr`

`GRAMMAR` double-semi-colon-separated list of BNF productions `END`

Keyword: `GRAMMAR` and `END` must be in upper-case

Keyword: `ATOM` represents atoms and must be in upper-case

Constants: must start with upper-case (see below)

Example:

```
GRAMMAR
  formula :=  ATOM | Verum | Falsum
            | formula & formula  | formula v formula
            | formula -> formula | formula <-> formula
            | ~ formula ;;
  expr    := formula ;;
END
```

Note: the final `;;` before `END` is necessary!

Note: unary connectives bind tighter than binary ones

Defining Tableau Rules

`RULE` rule-name numerator-pattern horizontal-separator
list-of-denominator-patterns ... `END`

Numerator Pattern: specifies how to instantiate meta-variables in numerator pattern to partition the contents of current node

Denominator Patterns: specifies how to construct the children of the current node using the instantiated meta-variables from the numerator

Notation: lower-case patterns are atoms and upper-case patterns are not

Unresricted Patterns: like `X` are instantiated arbitrarily and maximally

Restricted Patterns: like `A & B` are instantiated as specified and maximally

Principal Formulae: are specified by enclosing them in braces `{` and `}`

Examples of Rule Definitions: The And Rule

```
RULE And
    {A & B} ; X
  ===============
     A ; B ; X
END
```

RuleName: And

Numerator: choose a conjunction from the current node, thereby instantiating A and B, and instantiate X with (the list of) all other formulae in current node

Horizontal Separator ======== means don't backtrack over formula choice

Denominator: construct one and-child of the current node containing (the flattened list consisting of) the left conjunct (A), the right conjunct (B), and all other formulae (X) but exclude the principal formula A & B itself

Examples of Rule Definitions: The `Or` Rule

```
RULE Or
    {A v B} ; X
  ==================
     A ; X | B ; X
END
```

RuleName: `Or`

Numerator: choose a disjunction from current node, thereby instantiating both A and B, and instantiate `X` with (list of) all other formulae in current node

Horizontal Separator `=========` means don't backtrack over formula choice

Denominator: construct two or-children of the current node. A left-child containing (the flattened list consisting of) the left conjunct (A) and all other formulae (X). A right-child containing (the flattened list of) the right conjunct (B), and all other formulae (X), but exclude the principal formula `A v B` itself from both children

Examples of Rule Definitions: The `Id` Rule

```
RULE Id
    {a} ; { ~ a } ; X
  =====================
       Close
END
```

RuleName: `Id`

Numerator: choose an atomic formula a and its negation ~ a from current node, and instantiate `X` with (list of) <u>all other</u> formulae in current node

Horizontal Separator `========` means don't backtrack over formula choice

Denominator: close the current branch (and initiate backtracking)

Synthesising Result `status` Bottom Up

Rules: (Id) $\dfrac{p; \neg p; X}{\times}$ $(\wedge)$ $\dfrac{A \wedge B; X}{A; B; X}$ $(\vee)$ $\dfrac{A \vee B; X}{A; X \mid B; X}$

Status: Every rule manipulates an internal variable called `status`

Example: `Id` rule sets `status := closed`

Example: And rule passes up the `status` of its child unchanged

Example: `Or` rule passes up the `status` of its children according to "if both children have `status = closed` then `closed` else `open`"

Or-branching: if all children are `closed` then `closed` else `open`

And-branching: if all children are `open` then `open` else `closed`

Exploring the Search Space and Synthesising Results

Core: is a procedure to visit a tree generated by repeated application of a finite set of rules to an initial node

Visit: is basically an interpreter for the strategy language

Operation: returns the result of the visit of the tree generated by applying the `STRATEGY` to the current node

Systematic Proof Search Via Strategies

Strategy: a specification of how to apply the rules to the current node

Tactics: a strategy is built out of tactics using a tactic language

$$\mathtt{t} ::= \mathtt{skip} \mid \mathtt{fail} \mid \mathtt{Rule}\ \mathtt{r} \mid \mathtt{Alt}(\mathtt{t}, \mathtt{t}) \mid \mathtt{Seq}(\mathtt{t}, \mathtt{t}) \mid \mathtt{Repeat}(\mathtt{t})$$

Tactic Success: a tactic succeeds according to its form viz:

skip always succeeds

fail always fails

Rule r succeeds if rule r is applicable to the current node else fails

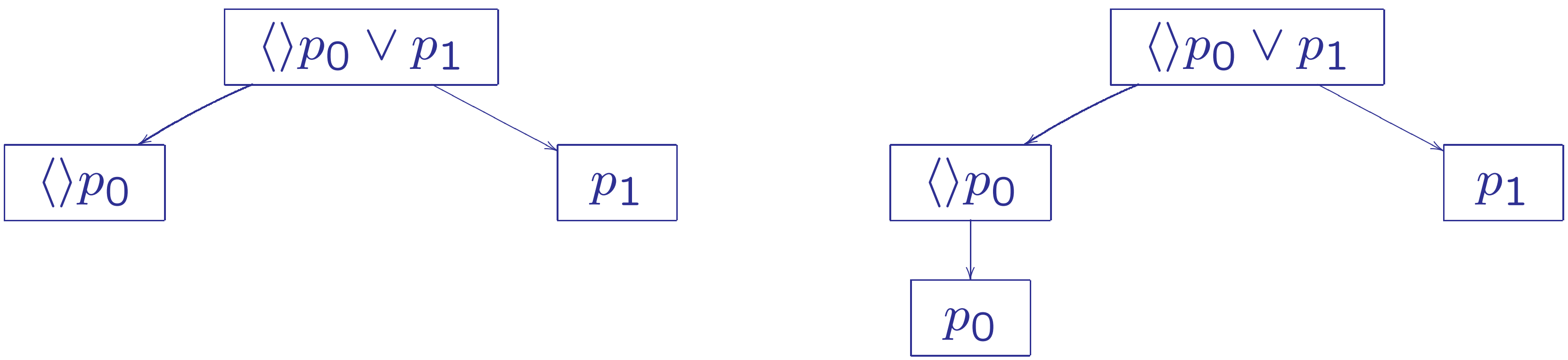
Alt(t1, t2) succeeds if t1 succeeds or else t2 succeeds on the current node

Seq(t1, t2) succeeds if t1 succeeds on the current node, producing a tableau with leaves l_1, l_2, ..., l_n and then t2 succeeds on each leaf l_i

Example: `Seq(t1, t2)`

Rules: (Id) $\dfrac{p; \neg p; X}{\times}$ $(\wedge)$ $\dfrac{A \wedge B; X}{A; B; X}$ $(\vee)$ $\dfrac{A \vee B; X}{A; X \mid B; X}$ (K) $\dfrac{\langle\rangle A; []X; Z}{A; X}$

Example: `STRATEGY = tactic ( Or ; K)`



Or: succeeds on the root node and produces two leaves

K: (K) succeeds on the left leaf but not on the right one

Result: Strategy `Seq(Or, K)` fails

Example: `Alt(t1, t2)`

Rules: (Id) $\dfrac{p; \neg p; X}{\times}$ $(\wedge)$ $\dfrac{A \wedge B; X}{A; B; X}$ $(\vee)$ $\dfrac{A \vee B; X}{A; X \mid B; X}$ (K) $\dfrac{\langle\rangle A; []X; Z}{A; X}$

Example: `STRATEGY = tactic (Id ! And ! Or ! K)`

Example:

$$\begin{tikzcd}[column sep=small] & \boxed{\langle\rangle p_0; []\neg p_1; \langle\rangle p_1} \arrow[dl] \arrow[dr] & \\ \boxed{p_0; \neg p_1} & & \boxed{p_1; \neg p_1} \end{tikzcd}$$

Result: `Id` fails, And fails, `Or` fails, K succeeds and creates And-choice for each $\langle\rangle$-formula $\langle\rangle p_0$ and $\langle\rangle p_1$

What Next? ... nothing, we did not say what to do after one successful application of the tactic

Result: Strategy succeeds but does not find closed tableau since `Id` is never applied to right child

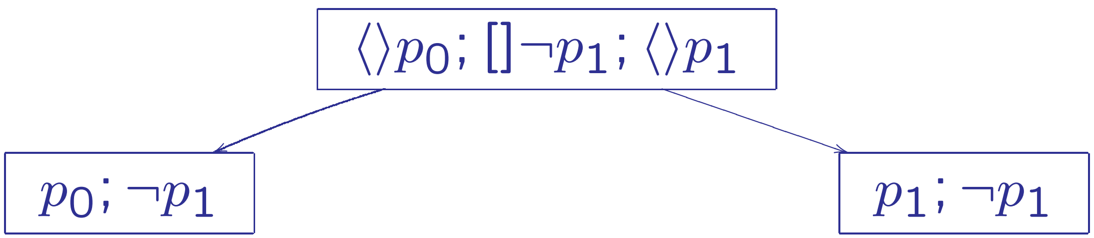
Example: `Repeat(t)`

Rules: (Id) $\dfrac{p; \neg p; X}{\times}$ $(\wedge)$ $\dfrac{A \wedge B; X}{A; B; X}$ $(\vee)$ $\dfrac{A \vee B; X}{A; X \mid B; X}$ (K) $\dfrac{\langle\rangle A; []X; Z}{A; X}$

Example:

```
let t = tactic (Id ! And ! Or )
in STRATEGY = tactic (t ; t)
```

Example:



Result: `Id` fails, And fails, `Or` fails, K succeeds and creates And-choice for each $\langle\rangle$-formula $\langle\rangle p_0$ and $\langle\rangle p_1$

Left leaf: all rules fail so `t` fails

Right leaf: `Id` succeeds and `Closes` right branch

Result: `STRATEGY` fails due to left leaf failure

Committing to Rule Choices

Rule Choice: multiple rules are applicable to current node

Invertible Rules: different choices between `And` and `Or` do not alter the final answer since these rules are invertible

Commit to Rule Choice:

```
let STRATEGY = tactic ( (Id ! And ! Or ! K)* )
```

Example: the modal logic K

Rules: (Id) $\dfrac{p; \neg p; X}{\times}$ $(\wedge)$ $\dfrac{A \wedge B; X}{A; B; X}$ $(\vee)$ $\dfrac{A \vee B; X}{A; X \mid B; X}$ (K) $\dfrac{\langle\rangle A; []X; Z}{A; X}$

Example: `STRATEGY = tactic (Id ! And ! Or ! K)*`

Example:

$$\begin{tikzcd}
 & \boxed{\langle\rangle p_0; []\neg p_1; \langle\rangle p_1} \arrow[dl] \arrow[dr] & \\
\boxed{p_0; \neg p_1} & & \boxed{p_1; \neg p_1}
\end{tikzcd}$$

Result: `Id` fails, And fails, `Or` fails, K succeeds and creates And-choice for each $\langle\rangle$-formula $\langle\rangle p_0$ and $\langle\rangle p_1$

Left leaf: all rules fail so `*` succeeds

Right leaf: `Id` succeeds and `Closes` right branch and no rule is applicable so `*` succeeds

Result: Strategy succeeds and does find closed tableau

Example: k.ml

```
CONNECTIVES [ ... ; "[" ; "]" ; "<" ; ">" ]
GRAMMAR formula :=  ...
                  | [] formula
                  | <> formula ;;
        ...
END
open Klib
TABLEAU  ...
  RULE K    { <>A } ; []X ; Z
            --------------------
                  A ; X
  END (* Dotted lines mean "don't commit" to formula choice *)
END
STRATEGY := tactic ( (False ! Id ! And ! Or ! K)* )
...
```

End of Part 2

Backtracking Over Formula Choices

```
  RULE K    { <>A } ; []X ; Z
          --------------------
                 A ; X
  END (* Dotted lines mean "don't commit" to formula choice *)
END
```

Formula Choices:

$$
\begin{tikzcd}
 & \boxed{\langle\rangle\varphi_1; \cdots; \langle\rangle\varphi_n; []X; Z} \arrow[dl] \arrow[d] \arrow[dr] & \\
\boxed{\varphi_1; X} & \boxed{\varphi_i; X} & \boxed{\varphi_n; X}
\end{tikzcd}
$$

Rule Semantics: apply `K` rule to obtain all children, explore the first child, if its visit returns `status = Close` then backtrack higher with `status = Close` else explore next child ...

Result: `Close` if some child `Close`s else `Open`

Example: kd.ml

```
TABLEAU
  RULE KD    ( <>A ) ; []X ; Z
            --------------------
                  A ; X
  END (* Dotted lines mean "don’t commit" to formula choice *)
END
STRATEGY := tactic ( (False ! Id ! And ! Or ! KD)* )
```

Note: parentheses “(.)” as opposed to braces “.” surrounding the principal formula specifies that it can be missing and rule will still apply

Core Algorithm Components

Rule Condition: check if the rule instance obeys rule side conditions

Rule Action: create the denominators by applying the rule instance

Branch Condition: check result of visit of last child and decide whether to visit the next child

Rule Backtrack: collect results of visited children and synthesise result of rule application to the current node

Defining Histories and Variables

TWB: provides three generic data structures to define histories/variables

`TwbSet.Make (ValType):` set of elements of type `ValType.t`

`TwbMSet.Make (ValType):` multiset of elements of type `ValType.t`

`TwbList.Make (ValType):` list of elements of type `ValType.t`

Signature:
```
module type ValType =
    sig
        type t
        val copy : t -> t
        val to_string : t -> string
    end
```

TWB/OCaml Signature of `ValType`

Example: define a module FormulaSet for "set of formulae" by instantiating the functor TwbSet.Make with type formula as ValType

```
module FormulaSet = TwbSet.Make(
    struct
        type t = formula
        let copy s = s
        let to_string = formula_printer
    end
)

HISTORIES
    (UNBOXES  :  FormulaSet.set := new FormulaSet.set)
END
```

TWB Set API

method `add_filter`: add an element that respects filter condition

method `addlist`: add a list of elements

method `mem`: check if an element is present

method `elements`: return the list of element in the object

method `is_empty`: check if the object is empty

method `empty`: return an empty instance of the object

Other methods: `filter, hd, del, length, cardinal, intersect, union, subset, is_equal ...`

Twblib.ml

```
let add (l, h) = h#addlist l
let notin (l, h) = not(h#mem (List.hd l))
let isin (l, h) = h#mem (List.hd l)
let not_emptyset l = not (l#is_empty)
let clear h = h#empty
```

Histories for Modal Logic KT

KT: extension of Hilbert system for K with reflexivity via $[]\varphi \to \varphi$

Rule: $(T)\ \dfrac{[]A; X}{A; []A; X}$ loops but incomplete without $[]A$ in denominator

History: UNBOXES to keep track of As from (T) applications

```
HISTORIES
    (UNBOXES  : FormulaSet.set := new FormulaSet.set)
END
RULE T     { [] A } ; Z
          ================
                A ; Z
   COND   [ notin(A, UNBOXES) ]
   ACTION [ UNBOXES  := add(A,UNBOXES) ]
END
STRATEGY := tactic ( (False ! Id ! And ! Or ! T ! K)* )
```

Exercise: A Better Version of the K Rule for KT

Old K Rule:

```
RULE K { <>A } ; []X ; Z --- A ; X END
```

Better K Rule for KT:

```
RULE KT { <>A } ; Z --- A ; UNBOXES END
```

Exercise: explain the differences in these rules

Beware: it is tempting to think that we can omit A from the denominator of the T rule (previous slide) on the grounds that it is recoverable from `UNBOXES`

Exercise: why is A essential in the denominator of the T rule?

Modal Logic $\mathrm{K4}$

K4: extension of Hilbert system for K with transitivity via $[]\varphi \to [][]\varphi$

Core: denominator of the K4 rule

Rule: $(K4)\ \dfrac{\langle\rangle A; []X; Z}{A; X; []X}$ loops due to $[]X$ in denominator

Example:

$$\boxed{\langle\rangle p_0; []\langle\rangle p_0}$$
$$\downarrow$$
$$\boxed{p_0; \langle\rangle p_0; []\langle\rangle p_0}$$
$$\downarrow$$
$$\boxed{p_0; \langle\rangle p_0; []\langle\rangle p_0}$$
$$\downarrow$$
$$\ldots$$

Loop Check: stop when we see the same “core”

Loop Checking in K4

```
HISTORIES
 CORES : Set of Formula := new Set.set
END
```

Meaning: declare a history with default value of emptyset

```
   let addcore(a, b, h) = h#addlist(a@b)

   TABLEAU
      RULE K4     { <>A } ; []X ; Z
                 ----------------------
                        A ; X ; []X

           COND   [ loopcheck(<>A, []X, CORES) ]
           ACTION [ CORES := addcore(<>A, []X, CORES) ]
      END
   END
```

Heuerding's Calculus for K4

Heuerding (TAB 96?): boxes accumulate so stop when we see the same diamond formula twice with no new boxes appearing in between

Tracking Boxes: we need a history to track boxes to evaluate whether there are new boxes

Tracking Diamonds: we need a history to track diamonds to evaluate whether we are seeing the same diamond twice

```
HISTORIES
   (DIAMONDS : FormulaSet := new FormulaSet.set);
   (UNBOXES  : FormulaSet := new FormulaSet.set)
END
```

Heuerding’s Calculus for K4

Diamonds: in diamond-history are blocked from being principal in (K4) rule but are added to diamond-history otherwise

```
RULE K4     { <>A } ; []X ; Z
           -----------------------
               A ; X ; []X
   COND   [ notin(A, DIAMONDS) ]
   ACTION [ DIAMONDS := add(A, DIAMONDS) ]
END
```

Heuerding's Calculus for K4

Diamonds: in diamond-history are blocked from being principal in (K4) rule but are added to diamond-history otherwise

```
RULE K4     { <>A } ; []X ; Z
           -----------------------
               A ; X ; []X
   COND   [ notin(A, DIAMONDS) ]
   ACTION [ DIAMONDS := add(A, DIAMONDS) ]
END
RULE NewBox { []A } ; Z ===  []A ; Z
   COND   [ notin(A, UNBOXES) ]
   ACTION [ UNBOXES  := add(A,UNBOXES);
            DIAMONDS := emptyset(DIAMONDS) ]
END
```

New Box: must empty box-history to release blocked diamonds

Unblock: these diamonds by putting `DIAMONDS` to the emptyset

Exercise: do we really need `[]A` in the denominator?

Cross-sibling Blocking

```
RULE K4     { <>A } ; []X ; Z
           ----------------------
                A ; X ; []X
   COND   [ notin(A, DIAMONDS) ]
   ACTION [ DIAMONDS := add(A, DIAMONDS) ]
END
```

Example: given two formula choices $\langle\rangle p_1$ and $\langle\rangle p_2$ in the K4 rule, each sibling should not redo the work of the other

$$\begin{tikzcd}[column sep=small] & \boxed{\langle\rangle p_1; [](\langle\rangle p_1 \wedge \langle\rangle p_2); \langle\rangle p_2} \arrow[dl] \arrow[dr] & \\ \boxed{p_1; \langle\rangle p_1 \wedge \langle\rangle p_2} & & \boxed{p_2; \langle\rangle p_1 \wedge \langle\rangle p_2} \end{tikzcd}$$

Explicit And-branching

And-branching: is created implicitly by rule and formula choices

Disadvantage 1: difficulty to collect the results of the And-siblings to compute a result for the given rule

Disadvantage 2: not possible to pass information from one And-sibling to another

Explicit: And-branching in a rule specified using || as a separator between denominators

Explicit: Or-branching in a rule specified using | as a separator between denominators

Explicit And-branching in Modal Logic K4

```
RULE K4      { <>A } ; []X ; Z
            -----------------------
                A ; X ; []X
   COND      notin(A, DIAMONDS)
   ACTION [ DIAMONDS := add(A, DIAMONDS) ]
END

RULE K4H     { <>A } ; <>Y ; Z
            ===========================
                A ; UNBOXES || <>Y
   COND      notin(A, DIAMONDS)
   ACTION [ [ DIAMONDS := add(A,DIAMONDS);
              DIAMONDS := add(Y,DIAMONDS) ] ;
            [ DIAMONDS := add(A,DIAMONDS) ]
          ]
END
```

Compiling and Executing Provers Using the TWB

Compiling .ml files to produce .twb files: `./twbcompile pc.ml`

Running .twb files: `./pc.twb source/pc/close.twb`

Noneg Option: the TWB negates the input formula by default

```
./pc.twb source/pc/close.twb --noneg
```

Various Flags for Debugging:

```
./pc.twb source/pc/close.twb --trace --verbose
```

End of Part 3

Shorthand Defaults for Rules

Default Minimalism: X below left is not needed since it does not change

$$(\vee)\ \frac{A \vee B; X}{A; X \mid B; X}$$

```
RULE Or { A v B } === A | B END
```

Default Rewriting: `A & B` without braces rewrites all conjunctions at once

$$(\wedge)\ \frac{A \wedge B; X}{A; B; X}$$

```
RULE And A & B === A ; B END
```

Beware: `A v B` without braces rewrites all disjunctions at once but is wrong!

$$(\vee)\ \frac{A \vee B; X}{A; X \mid B; X}$$

```
RULE Or A v B === A | B END
```

Backtracking Over Formula Choices in Or-Rule

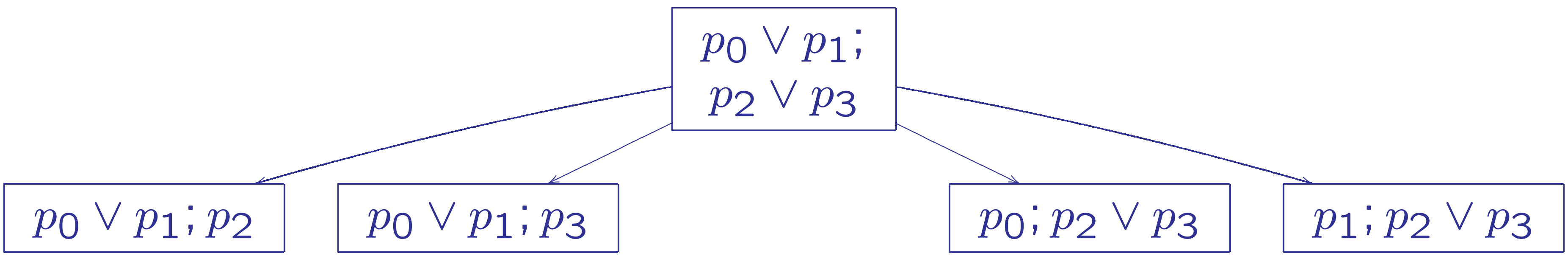
$$(\vee)\ \frac{A \vee B; X}{A; X \mid B; X}$$

```
RULE Or { A v B } === A | B END
```

Forbidden: to leave formula-choice in Or-branching rules

```
RULE Or-Verboten  { A v B } --- A | B END
```

Reason: how to specify which branch of which And-choice ?



Solution:
```
RULE Or   { A v B } --- A # B END
RULE Hash { A # B } === A | B END
STRATEGY = tactic ( (Id ! And ! (Or ; Hash) ! K)* )
```

More Complex Structure of a Node

Keyword: `node` is allowed in the `GRAMMAR` specification

By Default: a `node` is a set

Keywords: `set`, `mset`, `singleton` allowed

Node is Multiset: `node := mset ;;`

Node is Sequent of Sets: `CONNECTIVES [ ... "=>" ]`

```
node := set => set ;;
```

Node is Sequent of MultiSets: `CONNECTIVES [ ... "=>" ]`

```
node := mset => mset ;;
```

Node is Sequent with Singleton on RHS: `CONNECTIVES [ ... "=>" ]`

```
node := mset => singleton ;;
```

Sequent Calculus g4ip

```
CONNECTIVES [ "!";"&";"v";"->";"<->";"=>";"#"]
GRAMMAR
formula :=
     ATOM | Verum | Falsum
    | formula & formula
    | formula v formula
    | formula -> formula
    | formula <-> formula
    | ! formula            (* intuitionistic negation *)
    | # formula            (* marking device explained soon *)
;;

expr := formula ;;
node := mset => singleton ;;
END
```

```
SEQUENT   (* read rules upside down from TABLEAU *)
  RULE Id     Close               (* ie derivation found *)
            ==============
            { a } => { a }
  END

  RULE False  Close
            ================
             { Falsum } =>
  END

  RULE True   Close
            ==============
              => { Verum }
  END
```

```
RULE NegL  A -> Falsum =>
          ================
           { ! A } =>
END

RULE NegR   => A -> Falsum
          =================
            => { ! A }
END

RULE AndL
            A ; B =>
         =============
          { A & B } =>
END
```

```
RULE AndR  (* traditional or-branching *)
          => A | => B
          ============
          => { A & B }
END

RULE OrL   (* traditional or-branching *)
          A => | B =>
         ============
         { A v B } =>
END

RULE OrR   (* note the non-traditional and-branching *)
          => A  || =>  B
         ================
            => { A v B }
END
```

```
RULE ImpR
             A => B
          ============
          => { A -> B }
END

RULE ImpMP (* modus ponens on atomic formulae *)
              a ; B  =>
         =====================
         { a } ; { a -> B } =>
END
RULE ImpVerum
              Verum ; B  =>
         ================================
         { Verum } ; { Verum -> B } =>
END
```

```
RULE ImpNeg
           ( A -> Falsum ) -> B  =>
           ==========================
             { ( ! A ) -> B } =>
END

RULE ImpAnd
              C -> (D -> B) =>
         =====================
           { (C & D) -> B } =>
END
RULE ImpOr
            C -> B ; D -> B =>
         =====================
           { (C v D) -> B } =>
END
```

```
  RULE Hash
                   G ; # (C -> D) -> B => E
              ---------------------------------
                 G ; { (C -> D) -> B } => E
  END

  RULE ImpImp
               G ;  D -> B ; C => D  | B ; G => E
              =====================================
                 G ; { # (C -> D) -> B } => E
  END

END
```

```
let exit = function
    |"Open" -> "Not Derivable"
    |"Close" -> "Derivable"
    |s -> assert(false)

EXIT := exit(status@1)

STRATEGY :=
 let saturate = tactic (
     NegR ! NegL   ! Id        ! False ! True
   ! AndL ! ImpNeg ! ImpVerum ! ImpOr ! ImpAnd
   ! OrL  ! AndR   ! ImpMP    ! ImpR)
 in let hashimp = tactic ( Hash ; ImpImp )
 in tactic ( (saturate ! (OrR || hashimp) )* )

MAIN
```

Memoization

Memoization: is possible for any side-effect-free function f because two different calls to $f(x)$ must give the same result

Remember: the pair $(x, f(x))$ for selected x

Example: keyword `CACHE`

```
RULE K
         { <> A } ; [] X ;  Z
         -----------------------
              A ; X

    CACHE := true
END
```

So x is `(A ; X)` and $f(x)$ is `visit(A ; X)`

Simplification

```
RULE And
                  { A & B } ; X
       =================================================
        simpl(A, B) ; simpl(B, A) ; simpl(simpl(X, A), B)
END

RULE Or
                           { A v B } ; X
      =========================================================
      A ; simpl(X, A) | simpl(B, ˜ A ) ; simpl(simpl(X, B), ˜ A
END
```

simpl.ml

```
let rec simpl phi a =
  let rec aux phi a = match a with
    |formula (~ b) when b = a -> formula(Falsum)
    |formula (~ b)   -> formula ( ~ [aux phi b] )
    |formula (b & c) ->
              formula ( [aux phi b] & [aux phi c] )
    |formula (b v c) ->
              formula ( [aux phi b] v [aux phi c] )
    |_ when phi = a -> formula(Verum)
    |_ when phi = (nnf (formula ( ~ a ))) ->
              formula(Falsum)
    |_ -> a
  in boolean (aux phi a)
```

$\neg\varphi \vee \varphi \longrightarrow \top \qquad \varphi \vee \top \longrightarrow \top \qquad \varphi \vee \bot \longrightarrow \varphi \qquad \varphi \vee \varphi \longrightarrow \varphi$

$\neg\varphi \wedge \varphi \longrightarrow \bot \qquad \varphi \wedge \top \longrightarrow \varphi \qquad \varphi \wedge \bot \longrightarrow \bot \qquad \varphi \wedge \varphi \longrightarrow \varphi$

$\neg\bot \longrightarrow \top \qquad \neg\top \longrightarrow \bot$

Variables

Variables: allow us to pass information from children to parent nodes

Example: `status`

How Do I get the TWB ?

- One The Web:

  `http://twb.rsise.anu.edu.au`

- Via Darcs:

  `http://twb.rsise.anu.edu.au/Repository/twb-dev`

Conclusions

- It is generic and extensible
- Modular framework to build theorem provers
- It is based on a general tableau algorithm
- Front-end for tableau calculi
- The TWB targets both technical and non-technical users;
- It can be used to experiment with new logics, but also to built specialized TPs;
- It is based on a naive algorithms, but it can be easily extended to incorporate well known optimizations or to experiment with new ones.

Our Mantra: Simple things should be <u>simple</u>, difficult things should be <u>possible</u>.

	Universal	Existential	Linear
No Commit	-	-	(K)
Commit	$(\vee)$	$(K - rec)$	$(\wedge)$

Thus, the double bar "$||$" intuitively means: "If the sub-tableau for the first i denominators are all open, then explore the $(i + 1)^{st}$ sub-tableau . This is opposite to the single bar "$|$" traditionally used to specify the $(\vee)$-rule : "if the first branch is closed, then explore the second branch".

Core library by lines of code (loc)

component	loc
core	464
tableau library	818
syntax library	1624
data-type library	664
application (cli)	226
total	3800