Introduction to Formal Proof

Bernard Sufrin

Trinity Term 2018

2: Proofs *about* Propositional Calculus

Road Map

- ▷ We need a definition of the validity of conjectures that is independent of Natural Deduction
- ▷ We will construct this by
  - ∘ Defining the semantics of propositions
    - ⋆ inventing two truth-values – to represent true and false
    - ⋆ defining a valuation as a mapping from the *atomic* propositions to truth-values
    - ⋆ showing how to map *every* proposition to a truth value, given a valuation
- ▷ We will then equip ourselves to discuss soundness and completeness by
  - ∘ Defining the entailment relation: $P_1, P_2, ...P_n \vDash Q$ to mean:
    Q is true in any valuation in which $P_1, ..., P_n$ are all true
  - ∘ Defining *semantic validity* of the conjecture

$$P_1, P_2, ...P_n \vdash Q$$

to mean

$$P_1, P_2, ...P_n \vDash Q$$

- ▷ Next we will give a (Haskell) representation for proof trees, together with the definition of functions that
  - ∘ check that a proof tree is valid according to the rules of Natural Deduction
  - ∘ check that a valid proof tree proves the theorem it purports to prove
- ▷ We will use these definitions to prove the main result of this part of the course, namely that

every valid Natural Deduction proof proves a semantically valid conjecture

- ▷ Finally, we will prove some results *about* Natural Deduction proofs and use these to justify a new form of presentation of the Natural Deduction rules.

Propositional semantics

- ▷ There are two truth values. T represents true and F represents false.
- ▷ We are going to define the (truth-)value of every composite proposition in terms of the (truth-)values of its components.
- ▷ Our first step is to define for each propositional connective c, a truth-function c (*i.e.* a function of truth-valued operand(s) that yields a truth-value).
- ▷ The truth functions ¬, ∧, ∨, →, and ↔ are specified by the tables:

ϕ	$\neg\phi$
F	T
T	F

ϕ	ψ	$\phi \wedge \psi$
F	F	F
F	T	F
T	F	F
T	T	T

ϕ	ψ	$\phi \vee \psi$
F	F	F
F	T	T
T	F	T
T	T	T

ϕ	ψ	$\phi \rightarrow \psi$
F	F	T
F	T	T
T	F	F
T	T	T

ϕ	ψ	$\phi \leftrightarrow \psi$
F	F	T
F	T	F
T	F	F
T	T	T

Propositions are a recursive data type

- ▷ Imagine Haskell with a more liberal notation for defining **data** types
- ▷ We can define a type Prop to represent propositions

```
type AtomName = String
data Prop     = ⊥
              | Atomic AtomName
              | ¬ Prop
              | Prop ∧ Prop
              | Prop ∨ Prop
              | Prop → Prop
              | Prop ↔ Prop deriving (Show, Eq)
```

- ▷ Example: the proposition $P \rightarrow \neg Q \rightarrow R \rightarrow \bot$ would be represented by the Prop

```
(Atomic "P" → ((¬(Atomic "Q") → Atomic "R") → ⊥))
```

Proving things about Propositions

▷ A reminder: structural induction for `Prop`

- ∘ Suppose we want to prove $\mathcal{P}(p)$ for every p `::Prop`
- ∘ Base cases:
  - ⋆ prove $\mathcal{P}(\bot)$
  - ⋆ prove $\mathcal{P}(a)$ for every proposition a of the form `Atom` n
- ∘ Inductive cases:
  - ⋆ Assuming $\mathcal{P}(p)$ prove $\mathcal{P}(\neg(p))$
  - ⋆ Assuming $\mathcal{P}(p_l)$ and $\mathcal{P}(p_r)$ prove

    $\mathcal{P}(p_l \wedge p_r)$

    $\mathcal{P}(p_l \vee p_r)$

    $\mathcal{P}(p_l \rightarrow p_r)$

    $\mathcal{P}(p_l \leftrightarrow p_r)$

▷ This method of proof can be used in proofs about propositions.

Evaluating propositional formulae

▷ Definition: the *atoms* of proposition ϕ are the atomic propositions that appear in it.

▷ Definition: a *valuation for proposition* ϕ is a mapping from its atoms to truth values.

▷ Definition: a *valuation* is a total function from atoms to truth values

▷ Using the notation $[\![\, \phi \,]\!]_v$ to denote the value of ϕ in valuation v we can define, recursively, the value of any formula in any valuation:

$$
\begin{array}{rcll}
[\![\, \bot \,]\!]_v & = & \mathtt{F} & \\
[\![\, a \,]\!]_v & = & v(a) & \text{(for atomic } a\text{)} \\
[\![\, \neg\phi \,]\!]_v & = & \neg\ [\![\, \phi \,]\!]_v & \\
[\![\, \phi \wedge \psi \,]\!]_v & = & [\![\, \phi \,]\!]_v \wedge [\![\, \psi \,]\!]_v & \\
[\![\, \phi \vee \psi \,]\!]_v & = & [\![\, \phi \,]\!]_v \vee [\![\, \psi \,]\!]_v & \\
[\![\, \phi \rightarrow \psi \,]\!]_v & = & [\![\, \phi \,]\!]_v \rightarrow [\![\, \psi \,]\!]_v & \\
[\![\, \phi \leftrightarrow \psi \,]\!]_v & = & [\![\, \phi \,]\!]_v \leftrightarrow [\![\, \psi \,]\!]_v &
\end{array}
$$

▷ Note that *connectives* always appear within [[...]] and the truth functions always appear outside [[...]] (without the colour cue we'd be able to take our cue from the types)

▷ Example: suppose $v(R) = \mathtt{T}, v(H) = \mathtt{T}, v(D) = \mathtt{F}$, then:

$$
\begin{array}{rl}
 & [\![\, H \wedge R \rightarrow D \,]\!]_v \\
= & [\![\, H \wedge R \,]\!]_v \rightarrow [\![\, D \,]\!]_v \\
= & ([\![\, H \,]\!]_v \wedge [\![\, R \,]\!]_v) \rightarrow [\![\, D \,]\!]_v \\
= & (v(H) \wedge v(R)) \rightarrow v(D) \\
= & (\mathtt{T} \wedge \mathtt{T}) \rightarrow \mathtt{F} \\
= & \mathtt{T} \rightarrow \mathtt{F} \\
= & \mathtt{F}
\end{array}
$$

A lemma about irrelevant atoms

▷ Lemma: if the atom a is not an atom of ϕ, then $[\![\, \phi \,]\!]_v$ is independent of $v(a)$

▷ Proof method: induction over the structure of the proposition

▷ Base cases:

- ∘ $[\![\, \bot \,]\!] = \mathtt{F}$, and this is independent of $v(a)$
- ∘ Let P be an atomic proposition distinct from a

  If $v(a) = \mathtt{T}$, then $[\![\, P \,]\!]_v = v(P)$

  If $v(a) = \mathtt{F}$, then $[\![\, P \,]\!]_v = v(P)$

  So $[\![\, P \,]\!]_v$ is independent of $v(a)$

▷ Inductive cases are typified by $\wedge$

Let P_1 and P_2 be propositions not containing a, with $[\![\, P_i \,]\!]_v$ independent of $v(a)$ $(i = 1, 2)$

Then $[\![\, P_1 \,]\!]_v \wedge [\![\, P_2 \,]\!]_v$ is independent of $v(a)$

and $[\![\, P_1 \wedge P_2 \,]\!]_v = [\![\, P_1 \,]\!]_v \wedge [\![\, P_2 \,]\!]_v$ so it is independent of $v(a)$

Definitions: tautology, satisfiability, entailment

▷ Definition: "ϕ is a *tautology*" means $[\![\ \phi\ ]\!]_v = \mathtt{T}$ for *every* valuation v

▷ Definition: "ϕ is *satisfiable*" means $[\![\ \phi\ ]\!]_v = \mathtt{T}$ for *some* valuation v

(in this case we say that v satisfies ϕ)

▷ Definition: "the propositions $\phi_1, \phi_2, \ldots \phi_n$ *entail* ψ" means

$$[\![\ \psi\ ]\!]_v = \mathtt{T} \text{ for } \textit{every} \text{ valuation } v \text{ for which } [\![\ \phi_i\ ]\!]_v = \mathtt{T} \text{ (all } i = 1, \ldots n)$$

▷ We write this as $\phi_1, \phi_2, \ldots \phi_n \vDash \psi$

▷ Notice that $\vDash \phi$ if and only if ϕ is a tautology

Detour: Tautology and Satisfiability Checking

▷ In principle we must evaluate ϕ for all possible valuations

- If its value is *always* T then it's a tautology.
- If its value is *sometimes* T then it's satisfiable.
- If its value is *always* F then it's unsatisfiable.

▷ The lemma suggests that we can just evaluate ϕ for all combinations of values of the atoms that occur in it.

▷ The *truth table* method allows us to do this systematically by hand

▷ Example: $(H \wedge R \to D) \to \neg D \to \neg H$ is satisfiable but not a tautology

H	R	D	$((H\ \wedge$	$R)\ \to$	$D)\ \to$	$(\neg$	$D\ \to$	$\neg\ H)$
F	F	F	F	T	T	T	T	T
F	F	T	F	F	T	F	T	T
F	T	F	F	T	T	T	T	T
F	T	T	F	T	T	F	T	T
T	F	F	F	T	[F]	T	F	F
T	F	T	F	T	T	F	T	F
T	T	F	T	F	T	T	F	F
T	T	T	T	T	T	F	T	F

▷ Each row has

- on the left: a description of the relevant part of a valuation
- on the right: the values of each sub-proposition at that valuation written beneath the main connective of that sub-proposition.

☜13

▷ Is brute-force tautology / satisfiability testing practical?

- Evaluating a proposition is easy to implement:
  e.g. (in Haskell) using `Bool` for truth values

```
eval:: (Prop->Bool) -> Prop -> Bool
eval v prop = case prop of
                  ⊥          -> False
                  Atomic _   -> v(prop)
                  Not p      -> not (eval v p)
                  p∧q        -> eval v p && eval v q
                  p→q        -> if eval v p then eval v q else True
                  ...
```

- But a proposition ϕ with n distinct atoms has a truth table with 2^n rows, and needs evaluating 2^n times for a tautology test or to find all satisfying valuations.
- So tautology / satisfiability by this "brute force" method gets impractical quite quickly.
- There are more sophisticated algorithms (SAT-solvers) that can do these checks on propositions with huge numbers of atomic propositions of the kind that arise when search problems are modelled in propositional logic.

End of Detour

☜14

Soundness 1: Definition

▷ Our natural deduction proof system was intended to allow us to make rigorous arguments in support of conjectures of the form $\phi_1, ..., \phi_n \vdash \psi$

▷ Having a proof of such a conjecture ought to give us complete confidence that the conclusion is true in a situation where all the premisses are true.

▷ We need to convince ourselves that the Natural Deduction rules are **sound as a whole**; in other words, that

if there is a proof of $\phi_1, ..., \phi_n \vdash \psi$, then $\phi_1, ..., \phi_n \vDash \psi$

▷ We will convince ourselves by means of a rigorous argument about proofs: a *meta-proof*

▷ This argument will look like an argument about a (recursively-defined) data structure

- We start by showing how to represent a proof tree as a (Haskell) data structure
- Then we show how to define a (Haskell) function that checks the validity of a proof tree
- Then we show, with a proof by structural induction over valid proof trees, that

Every valid proof tree proves a semantically valid conjecture

Soundness 2: Proofs represented as data structures

▷ Proofs are trees built by putting simpler proofs together using inference rules

▷ The leaves of a proof tree are the premisses of the conjecture being proved (or the hypotheses of hypothetical subproofs)

▷ So we can define a type `ProofTree` (in Haskell) as follows:

```
data ProofTree = InferBy RuleName [ProofTree] Prop
```

▷ Each node in a proof tree represents the use of an inference rule, and is labelled with

- zero or more subproofs
- the name of a proof rule
- the conclusion that is inferred (and that the node purports to prove from the subproofs)

▷ For later use we define:

```
conclusion :: Proof -> Prop
conclusion(InferBy name subproofs conc) = conc
```


▷ In this representation, the proof

$$\dfrac{\dfrac{\dfrac{}{\psi \wedge \phi}\,\text{hyp}}{\phi}\,\wedge\text{-elim-R} \qquad \dfrac{\dfrac{}{\psi \wedge \phi}\,\text{hyp}}{\psi}\,\wedge\text{-elim-L}}{\phi \wedge \psi}\,\wedge\text{-intro}$$

would be represented by the Haskell tree

```
InferBy "∧-intro" [l, r] (phi ∧ psi) where
   l = InferBy "∧-elim-R" [InferBy "hyp" [] (psi ∧ phi)] phi
   r = InferBy "∧-elim-L" [InferBy "hyp" [] (psi ∧ phi)] psi
   phi = Atomic "φ"
   psi = Atomic "ψ"
```

▷ But not every `ProofTree` built by `InferBy` represents a proper proof. For example:

- The *hyp* rule has no subproofs, *and can only infer an actual premiss (or hypothesis)*
- The and-introduction rule requires subproofs that prove the conjuncts of its conclusion
- The and-elimination rules requires a subproof that ends in a conjunction ....

Soundness 3: a proof checker

Next we will build a (Haskell) function that checks whether a tree that purports to be a proof of a conjecture actually represents a valid proof of that conjecture.

```
data Conjecture = [Prop] ⊢ Prop

proves:: Proof  -> Conjecture -> Bool
p ‘proves‘ (ps ⊢ c) = conclusion p == c && valid ps p
```

We need to check that the purported proof’s conclusion is the conclusion of the conjecture; and that the tree as a whole was built according to the proof rules, and that that the leaves of the proof tree are premisses or assumptions, and that the assumptions made for hypothetical subproofs are used in only those subproofs.

For the last two reasons we pass a list representing the collection of currently-in-scope hypotheses (and premisses) to the workhorse validity-checker, `valid`.

The validity of a particular inference depends on the rule used, and requires the validity of its subproofs (if any). First we present a few of the more straightforward cases.

```
valid::  [Prop] -> Proof -> Bool
valid hs proof = case proof of
 InferBy "hyp"    []       c     -> c ∈ hs

 InferBy "∧-intro"   [l, r]   (p∧q) ->
   valid hs l && conclusion l == p &&
   valid hs r && conclusion r == q

 InferBy "∧-elim-L"  [pr’] c  ->
   valid hs pr’ && case conclusion pr’ of p’∧_ -> c==p’; _ -> False

 InferBy "∧-elim-R"  [pr’] c  ->
   valid hs pr’ && case conclusion pr’ of _∧p’ -> c==p’; _ -> False

 InferBy "→-elim"   [l, r]    c  ->
   valid hs l && valid hs r &&
   case conclusion r of
     (p→q) -> conclusion l == p && q == c
      _    -> False
```


The interesting cases are those with hypothetical subproofs. For example:

```
InferBy "→-intro" [pr’] (p→q) ->
  valid (p:hs) pr’ && conclusion pr’ == q

InferBy "∨-elim"  [d, l, r] c   ->
  valid hs d &&
  conclusion l == c &&
  conclusion r == c &&
  case conclusion d of
    p∨q -> valid (p:hs) l &&
           valid (q:hs) r
    _   -> False
...
```

In each case, the assumption is added to the collection of assumptions permitted in the subproof(s) while they are being checked for validity.

This captures the graphically-presented notion of “boxed subproof” – making quite precise what we mean when we say of a rule that the assumption made here cannot be used outside of the subproof(s) used to justify the inference step.

▷ Exercise: complete the proof checker by implementing the other proof rules.

Soundness 4: some observations about subproofs

▷ Observations about hypothetical subproofs of valid proofs

▷ Example:

```
  valid hs (InferBy "→-intro" [pr’] (p→q))
= {by valid definition (the "→-intro" case) }
  valid (p:hs) pr’ && conclusion pr’ == q
= {by definition of proves}
  pr’ ‘proves‘ (p:hs ⊢ q)
```

▷ Similarly, if `conclusion d = p ∨ q`

```
  valid hs (InferBy "∨-elim"  [d, l, r] c
= { ... }
  d ‘proves‘ (hs ⊢ p ∨ q)      &&
  l ‘proves‘ (p:hs ⊢ c)        &&
  r ‘proves‘ (q:hs ⊢ c)
```

▷ Observations about non-hypothetical subproofs of valid proofs

▷ Example:

```
  valid hs (InferBy "∧-intro" [l, r] (p∧q))
= {by valid definition (the "∧-intro" case) }
  valid hs l && conclusion l == p &&
  valid hs r && conclusion r == q
= {by definition of proves}
  l ‘proves‘ (hs ⊢ p) &&
  r ‘proves‘ (hs ⊢ q)
```

▷ Similarly, if `conclusion r = p→q`

```
  valid hs (InferBy "→-elim" [l, r] q
= { ... }
  l ‘proves‘ (hs ⊢ p )      &&
  r ‘proves‘ (hs ⊢ p→q)
```


Soundness 6: proof of soundness

Soundness: Every valid proof tree is sound – *i.e.* proves a semantically valid conjecture

Proof: (for every proof tree pr) if pr `‘proves‘` $[\phi_1, ...\phi_n] \vdash \psi$ then $\phi_1, ...\phi_n \vDash \psi$

Suppose pr `‘proves‘` $[\phi_1, ...\phi_n] \vdash \psi$

▷ We will proceed by induction on the structure of pr to show that $[\![\psi]\!]_v = \mathtt{T}$ for any valuation v satisfying $\phi_1, ...\phi_n$; *i.e.* such that $[\![\phi_1]\!]_v = ... = [\![\phi_n]\!]_v = \mathtt{T}$.

▷ There will be a case for each inference rule.

Base Case: pr is `InferBy "hyp" []` ψ

- then `valid` $[\phi_1, ...\phi_n]$ pr (definition of `proves`)
- so $\psi \in [\phi_1, ...\phi_n]$ (definition of `valid`)
- so $[\![\psi]\!]_v = \mathtt{T}$ for any v satisfying $\phi_1, ...\phi_n$

∧ introduction: suppose pr is `InferBy "∧-intro" [l, r] (p∧q)`

then l `‘proves‘` $[\phi_1, ..., \phi_n] \vdash p$ (by an earlier observation)

and r `‘proves‘` $[\phi_1, ..., \phi_n] \vdash q$ (by an earlier observation)

Suppose (induction hypotheses) that the nested valid proofs l, r are sound,

i.e. $\phi_1, ...\phi_n \vDash p$ and $\phi_1, ...\phi_n \vDash q$

- then $[\![p]\!]_v = [\![q]\!]_v = \mathtt{T}$, for any v satisfying $\phi_1, ...\phi_n$ (by the induction hypotheses)
- so $[\![p \wedge q]\!]_v = \mathtt{T}$, for any v satisfying $\phi_1, ...\phi_n$
- so $[\![\psi]\!]_v = \mathtt{T}$ for any v satisfying $\phi_1, ...\phi_n$
- so $\phi_1, ...\phi_n \vDash \psi$

∧ elimination R: suppose pr is `InferBy "∧-elim-R" [pr’] q`

then pr' `‘proves‘` $[\phi_1, ..., \phi_n] \vdash p \wedge q$ (for some p).

Suppose (induction hypothesis) that pr' is sound, *i.e.* $\phi_1, ...\phi_n \vDash p \wedge q$

- So $[\![p \wedge q]\!]_v = \mathtt{T}$, for any v satisfying $\phi_1, ...\phi_n$ (by the induction hypothesis)
- So $[\![q]\!]_v = \mathtt{T}$, for any v satisfying $\phi_1, ...\phi_n$ (definition of ∧)
- so $[\![\psi]\!]_v = \mathtt{T}$ for any v satisfying $\phi_1, ...\phi_n$
- so $\phi_1, ...\phi_n \vDash \psi$

$\vee$ elimination : suppose pr is `InferBy "∨-elim" [d l r]` ψ

then d `` `proves` `` $[\phi_1, ..., \phi_n] \vdash p \vee q$ (for some p, q) (by an earlier observation)

and l `` `proves` `` $[p, \phi_1, ..., \phi_n] \vdash \psi$ (ditto)

and r `` `proves` `` $[q, \phi_1, ..., \phi_n] \vdash \psi$ (ditto)

Suppose (induction hypothesis) that d, l, r are sound,
i.e. (a) $\phi_1, ...\phi_n \vDash p \vee q$, and (b) $p, \phi_1, ...\phi_n \vDash \psi$, and (c) $q, \phi_1, ...\phi_n \vDash \psi$

- So $[\![p \vee q]\!]_v = \mathtt{T}$, for any v satisfying $\phi_1, ...\phi_n$ (by induction hypothesis a)
- and $[\![\psi]\!]_v = \mathtt{T}$, for any v satisfying $p, \phi_1, ...\phi_n$ (ditto)
- and $[\![\psi]\!]_v = \mathtt{T}$ for any v satisfying $q, \phi_1, ...\phi_n$ (ditto)
  - ∗ Now suppose that v satisfies $\phi_1, ...\phi_n$, then $[\![p \vee q]\!]_v = \mathtt{T}$
  - ∗ then one or both of $[\![p]\!]_v = \mathtt{T}$ or $[\![q]\!]_v = \mathtt{T}$ (definition of $\vee$)
    - ▷ If $[\![p]\!]_v = \mathtt{T}$ then v satisfies $p, \phi_1, ...\phi_n$ so $\phi_1, ...\phi_n \vDash \psi$ (induction hypothesis b)
    - ▷ If $[\![q]\!]_v = \mathtt{T}$ then v satisfies $q, \phi_1, ...\phi_n$ so $\phi_1, ...\phi_n \vDash \psi$ (induction hypothesis c)

▷ Exercise: complete the proof of soundness

▷ Exercise: what happens to soundness if you add the rule $\overline{\psi}$ Placet

Consequence of Soundness

▷ Suppose you cannot find a proof of $\phi_1, ...\phi_n \vdash \psi$.

▷ Then it may be worth checking whether it is *semantically invalid*.

▷ For if it is semantically invalid, then you will *never* find a proof for it.

Q: How can I check for semantic invalidity?

A: Just find a single *counterexample*
— a valuation that satisfies $\phi_1, ...\phi_n$ but doesn't satisfy ψ.

Exercise: find a proof of $R, H \wedge R \to D, D \vdash H$, or give a counterexample.

Exercise: find a proof of $R, H \wedge R \to D, \neg D \vdash \neg H$, or give a counterexample.

Statement of the Completeness Theorem for Natural Deduction

▷ Definition: We say that a proof system is *complete* for a semantics, when

everything that is true in the semantics can be proven in the proof system

▷ Completeness of Natural Deduction:

every semantically valid[1] conjecture can be proven by Natural Deduction

in symbols:

If $\phi_1, ..., \phi_n \vDash \psi$ then there is a pr :: `ProofTree` such that pr `` `proves` `` $[\phi_1, ..., \phi_n] \vdash \psi$

▷ The proof of completeness is intricate, and beyond the scope of these lectures.

[1] *i.e.* semantically valid under the present definition of $\wedge, \vee, ..., \mathtt{T}, \mathtt{F}$.

Reusing proofs and using proofs-about-proofs

▷ There are two kinds of "reusable result" we can use in proofs

- Derived Rules: a derived rule is the obvious generalization of a proven conjecture.
- Admissible Rules: an admissible rule is a rule that can be proven by a meta-proof *about* proofs, that shows that any proof that uses the rule is equivalent to one that doesn't.

Warning: for a while we will use the *unofficial notation:* $\phi_1, ...\phi_n \boxed{\vdash} \psi$
to mean that $\phi_1, ...\phi_n \vdash \psi$ has a (natural deduction) proof.

▷ Example: **weaken** is an admissible rule

$$\frac{\phi_1, ...\phi_n \boxed{\vdash} \psi}{\phi, \phi_1, ...\phi_n \boxed{\vdash} \psi} \text{ weaken}$$

▷ This can be read as a proof rule, or as a conjecture about proofs, whose meta proof goes:

- Let pr be such that pr `` `proves` `` $\phi_1, ...\phi_n \vdash \psi$
- Then $valid\ [\phi_1, ...\phi_n]\ pr$, so $valid\ [\phi, \phi_1, ...\phi_n]\ pr$, so pr `` `proves` `` $\phi, \phi_1, ...\phi_n \vdash \psi$

▷ Another admissible rule is:

$$\frac{\phi, \phi, \Gamma \boxed{\vdash} \psi}{\phi, \Gamma \boxed{\vdash} \psi}\ \text{contract}$$

▷ This suggests that the number of occurences of a formula in the assumptions doesn't really matter when we are doing (ND) proofs; and the admissible rule **weaken** means that we can neglect spurious assumptions.

▷ Another way of looking at these rules is that we can prune irrelevant premisses from a conjecture, and irrelevant subproofs from the proof of a conjecture.

▷ Example: **cut** is an admissible rule

Mathematicians and Computer Scientists habitually use Lemmas to simplify the structure of a proof by proving an intermediate result "on the fly". This is justified by the following result (which has an analogous dual reading to **thin**)

$$\frac{\Gamma \boxed{\vdash} \phi \qquad \phi, \Gamma \boxed{\vdash} \psi}{\Gamma \boxed{\vdash} \psi}\ \text{cut}$$

If there is a proof pl of $\Gamma \vdash \phi$ and a proof pr of $\phi, \Gamma \vdash \psi$, then there is a proof of $\Gamma \vdash \psi$

▷ Meta proof: (sketch)

- Define $paste$`::ProofTree->ProofTree->ProofTree` such that $paste\ pl\ pr$ replaces every occurence in pr of `InferBy "Hyp" []` $(conclusion\ pl)$ by pl.
- Prove (by structural induction) that $paste\ pl\ pr$ `` `proves` `` $\Gamma \vdash \psi$

ASIDE: completeness steps in proofs of admissibility

▷ Example: substitutivity of equivalent propositions is admissible

$$\frac{\phi_1, ...\phi_n, \phi(A) \boxed{\vdash} \psi(A) \qquad \phi_1, ...\phi_n \boxed{\vdash} A \leftrightarrow B}{\phi_1, ...\phi_n, \phi(B) \boxed{\vdash} \psi(B)}\ \text{subst}$$

(Here ϕ, ψ are "proposition schemas"; *i.e.* have type `Prop->Prop`)

▷ Proof:

1: By soundness: $\phi_1, ...\phi_n, \phi(A) \vDash \psi(A)$ and $\phi_1, ...\phi_n \vDash A \leftrightarrow B$

2: A straightforward semantic argument from the definition of $[\![\ ...\ ]\!]$... can be used to show from (1) that: $\phi_1, ...\phi_n, \phi(B) \vDash \psi(B)$

3: By completeness we know that there is a proof of $\phi_1, ...\phi_n, \phi(B) \vdash \psi(B)$ (*but not what the proof is!*)

END ASIDE

Reformulating ND as a single-conclusion sequent calculus

▷ We have used the unofficial notation $\Gamma \boxed{\vdash} \psi$ to mean $\Gamma \vdash \psi$ has a (natural deduction) proof

▷ We will now reformulate natural deduction as an inference system using this notation

▷ The assumption collections implicit in ND will become explicit here

▷ We drop the word "conjecture" and refer to the form $\Gamma \boxed{\vdash} \psi$ as a *single-conclusion sequent*

▷ The hypothesis rule:

"We may conclude ϕ from any collection of hypotheses that contains ϕ"

$$\frac{}{\Gamma, \phi \boxed{\vdash} \phi}\ \text{hyp}$$

▷ We will take the cut, weaken, and contract rules as read

▷ Rules can be straightforwardly transcribed from ND. *eg:*

$$\frac{\Gamma \boxed{\vdash} \phi \qquad \Gamma \boxed{\vdash} \psi}{\Gamma \boxed{\vdash} \phi \land \psi}\ \land\text{-i}$$

$$\frac{\Gamma \boxed{\vdash} \phi}{\Gamma \boxed{\vdash} \phi \lor \psi}\ \lor\text{-i}_L$$

$$\frac{\Gamma \boxed{\vdash} \psi}{\Gamma \boxed{\vdash} \phi \lor \psi}\ \lor\text{-i}_R$$

$$\frac{\Gamma, \phi \boxed{\vdash} \psi}{\Gamma \boxed{\vdash} \phi \to \psi}\ \to\text{-i}$$

▷ They may be seen as the rules of a new inference system *equivalent to ND*....

... or as an inference system whose subject matter is proofs in ND.

▷ Elimination rules are also straightforwardly transcribed:

$$\frac{\Gamma \boxed{\vdash} \phi \land \psi}{\Gamma \boxed{\vdash} \phi}\ \land\text{-e}_L$$

$$\frac{\Gamma \boxed{\vdash} \phi \land \psi}{\Gamma \boxed{\vdash} \psi}\ \land\text{-e}_R$$

$$\frac{\Gamma \boxed{\vdash} \phi \qquad \Gamma \boxed{\vdash} \phi \to \psi}{\Gamma \boxed{\vdash} \psi}\ \to\text{-e}$$

$$\frac{\Gamma \boxed{\vdash} \phi \lor \psi \qquad \Gamma, \phi \boxed{\vdash} \kappa \qquad \Gamma, \psi \boxed{\vdash} \kappa}{\Gamma \boxed{\vdash} \kappa}\ \lor\text{-e}$$

Exercise: transcribe the rules for negation

▷ These derived rules can be proven from the elimination rules (and *vice-versa*) using cut, hyp

$$\frac{\Gamma, \phi, \psi \boxed{\vdash} \kappa}{\Gamma, \phi \land \psi \boxed{\vdash} \kappa}\ \land\vdash$$

$$\frac{\Gamma, \phi \to \psi \boxed{\vdash} \phi \qquad \Gamma, \psi \boxed{\vdash} \kappa}{\Gamma, \phi \to \psi \boxed{\vdash} \kappa}\ \to\vdash$$

$$\frac{\Gamma, \phi \boxed{\vdash} \kappa \qquad \Gamma, \psi \boxed{\vdash} \kappa}{\Gamma, \phi \lor \psi \boxed{\vdash} \kappa}\ \lor\vdash$$

▷ They support *goal-directed* proof search more directly than the elimination rules.

▷ Proofs in this calculus consist of trees of sequents

- Leaves are labelled with `hyp`
- Branches are labelled with the name of a non-hyp rule

▷ For example (reverting to the conventional notation for sequents)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{E \vdash E}\ \text{hyp}}{E \vdash E \lor F}\ \lor\text{-i}_L \quad \dfrac{\dfrac{}{E \vdash E}\ \text{hyp}}{E \vdash E \lor G}\ \lor\text{-i}_L}{E \vdash (E \lor F) \land (E \lor G)}\ \land\text{-i} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F, G \vdash F}\ \text{hyp}}{F, G \vdash E \lor F}\ \lor\text{-i}_R \quad \dfrac{\dfrac{}{F, G \vdash G}\ \text{hyp}}{F, G \vdash E \lor G}\ \lor\text{-i}_R}{F, G \vdash (E \lor F) \land (E \lor G)}\ \land\text{-i}}{F \land G \vdash (E \lor F) \land (E \lor G)}\ \land\vdash}{F \land G \vdash (E \lor F) \land (E \lor G)}\ \land\text{-i}}{E \lor (F \land G) \vdash (E \lor F) \land (E \lor G)}\ \lor\vdash$$

▷ Reading the tree upwards, think of the left hand side of $\vdash$ as "what we have established from the assumptions" and the right hand side as "what we need to establish to close this branch of the proof"

▷ Such proof trees can be linearized

1:	$E \vee (F \wedge G)$	premiss
2:	E	hyp
3:	$E \vee F$	$\vee$-i$_L$ 2
4:	$E \vee G$	$\vee$-i$_L$ 2
5:	$(E \vee F) \wedge (E \vee G)$	$\wedge$-i 3, 4
6:	$F \wedge G$	hyp
7:	F	hyp
8:	G	hyp
9:	$E \vee F$	$\vee$-i$_R$ 7
10:	$E \vee G$	$\vee$-i$_R$ 8
11:	$(E \vee F) \wedge (E \vee G)$	$\wedge$-i 9, 10
12:	$(E \vee F) \wedge (E \vee G)$	$\wedge\vdash$(6) 7-11
13:	$(E \vee F) \wedge (E \vee G)$	$\vee\vdash$(1) 2-5, 6-12

Is it essential to represent proofs in Haskell?

No, provided

- we are prepared to accept the idea of structural induction over a proof tree
- we are quite precise about the meaning of the proof-rule notation: namely that it constructs valid proof trees from valid proof trees, where validity is specified by patterns and (possibly) subproof notation
- we are quite precise about the meaning of the subproof notation
- we make quite explicit the places where hypotheses/premisses can be used: namely at the leaves of (certain) trees

But it helps novices distinguish propositional proofs from proofs about such proofs

> "Logicians were functional programmers *avant la lettre*; but we are now living *après la lettre*, so if you want to study logic then you should really study functional programming first" *Fr. Saul N. Braindrane*

An alternative approach: valid proofs as a data type

▷ We can make the *rules themselves* the constructors of the proof data type

```
data Proof = Hyp   Prop
           | AndI  Proof Proof | AndEL Proof       | AndER Proof
           | ImpI  Proof       | ImpE  Proof Proof | ...
```

define a data type to represent proven conjectures (theorems)

```
data Theorem = [Prop] ⊢ Prop             -- ⊢ written :|- in ‘‘proper’’ Haskell
```

and define a function that extracts the theorem that the proof proves

```
thm :: Proof -> Theorem
thm pr =
 case pr of
   Hyp  h    -> [h] ⊢ h
   AndI l r  -> let hsl ⊢ p = thm l; hsr ⊢ q = thm r in  hsl++hsr ⊢ p ∧ q
   AndEL pr' -> let hs   ⊢ p ∧ q = thm pr'           in  hs ⊢ p
   ImpI  pr' -> let p:hs ⊢ q = thm pr'               in  hs ⊢ p→q
   ImpE l r  -> let hsl ⊢ p = thm l; hsr ⊢ p'→q = thm r
                in  if p==p' then hsl++hsr ⊢ q else error (show [p, p'->q])
```


▷ The function `thm` is *partial* – not all `Proofs` correspond to a theorem ...

... but if a theorem emerges from `thm` it will have *exactly* the right premisses

⟨aside⟩

Avoid accidental forgery of arbitrary theorems by making `Theorem` an abstract type

... in Haskell this is done by hiding its constructor in the module that defines theorems

... and exporting `thm`, but not the `Theorem` constructor

```
module Theorems (Theorem, thm)
  import Prop
  data   Theorem = [Prop] ⊢ Prop
  thm :: Proof -> Theorem
  thm pr = ...
```

⟨/aside⟩

Proof procedures and completeness

▷ Suppose we could build a function `prove:: Conjecture -> Proof` that, *if it terminates without error*, finds a correct proof for its argument conjecture.

▷ Then we could define

```
provable :: Conjecture -> Bool
provable(Γ ⊢ p) =
        let Γ' [⊢] p' = thm(prove(Γ ⊢ p))
        in  Γ' == Γ && p' == p
```

and be able to claim that $provable(\Gamma \vdash p)$ == `True` (if it terminates without error).

▷ The completeness theorem would now take the form of a proof that

$$provable(\Gamma \vdash p) == \texttt{True} \text{ follows from } \Gamma \vDash p$$

▷ So a way of proving the completeness theorem is to show that we can indeed build such a `prove` function, and that *it terminates without error at semantically valid conjectures*.

Contents

Note 1: Syntax & semantics: connectives & truth functions 3 ☞

The propositional connectives are *syntactic* – used to form propositions from other propositions, whereas the truth functions are *semantic* – they operate on boolean values.

In order to make it clear that the truth functions are distinct from the propositional connectives to which they correspond, despite being represented by symbols of the roughly same shape, we have emphasized them *like this* here.

The definitions of the truth functions $\neg, \wedge, \vee$ are completely straightforward; and the definition of $\leftrightarrow$ can be understood once one has accepted the definition of $\rightarrow$.

On the other hand, the definition of $\rightarrow$ probably needs some explanation. One way of explaining it is that it should be true exactly when the truth value on the left is no weaker than that on the right.

Note 2: 4 ☞

We aren't forced to use the liberal notation for these lectures to work, providing we don't mind putting up with a bit of clutter.

```
type AtomName = String
data Prop     = Absurd
              | Atomic AtomName
              | Not Prop
              | Prop :/\  Prop
              | Prop :\/  Prop
              | Prop :->  Prop
              | Prop :<-> Prop
              deriving (Show, Eq)
infixl 8 :/\
infixl 7 :\/
infixr 6 :->
infixl 5 :<->
```

Note 3: Tips for making truth tables 11 ☞

- ▷ To ensure completeness of a truth table: write down the "relevant valuation" part of the table systematically (think binary!)
- ▷ To achieve conciseness: write the value of each nonatomic sub-proposition beneath its main connective.
- ▷ To achieve accuracy: fill in each row in bottom-up order.

▷ To avoid clutter: don't repeat the atomic values on the right hand side of the table.

Note 4: A search problem modelled in propositional logic 12 ☞
Although discussion of this topic is beyond the scope of the course, it's worth noting here that we can formalize many search problems as propositional satisfiability problems.

▷ For example in Sudoku:

- there is 9x9 grid divided into 3x3 squares
- a problem is posed by placing digits 1-9 on the grid
- it is solved by placing digits 1 to 9 on the unfilled part of the grid so that each line, each column and each 3x3 square contains each digit exactly once; for example:

Here's a solvable Sudoku problem, and its solution; most are *much* harder than this

```
8 3 7 | 5 9 2 | 1 4 6        8 3 7 | 5 9 2 | 1 4 6
5 _ 1 | 4 _ 8 | 9 _ 2        5 6 1 | 4 7 8 | 9 3 2
4 9 2 | 1 3 6 | 7 8 5        4 9 2 | 1 3 6 | 7 8 5
------+-------+------        ------+-------+------
3 8 9 | 7 6 5 | 4 2 1        3 8 9 | 7 6 5 | 4 2 1
2 _ 5 | 8 _ 9 | 3 _ 7        2 4 5 | 8 1 9 | 3 6 7
7 1 6 | 3 2 4 | 8 5 9        7 1 6 | 3 2 4 | 8 5 9
------+-------+------        ------+-------+------
9 5 3 | 2 _ 1 | 6 7 4        9 5 3 | 2 8 1 | 6 7 4
1 _ 4 | 6 _ 3 | 2 _ 8        1 7 4 | 6 5 3 | 2 9 8
6 2 8 | 9 _ 7 | 5 1 3        6 2 8 | 9 4 7 | 5 1 3
```

A Sudoku puzzle can be coded straightforwardly as a propositional satisfiability problem using 729 = $(9 \times 9 \times 9)$ propositional atoms $S_{ijd}(1 \le i, j, d \le 9)$ where S_{ijd} is interpreted as "digit d is placed on the grid at (i, j).

The simplest straightforward encoding of the problem as a proposition specifies it with four main conjuncts each of which is systematically constructed from conjunctions of disjunctions of atomic propositions or their negations:

1. At least one number at each grid location[2]

[2] The $\bigvee$, and $\bigwedge$ notations are analogous to the familiar $\sum$ and $\prod$ notations. They are concise ways of writing systematically formed conjunctions and disjunctions. For example, $\bigvee_{1 \le d \le 9} S_{ijd}$ means $S_{ij1} \vee S_{ij2} \vee \ldots \vee S_{ij9}$.

$$\bigwedge_{1 \le i \le 9} \bigwedge_{1 \le j \le 9} \bigvee_{1 \le k \le 9} S_{ijk}$$

2. each number appears at most once in each column:

$$\bigwedge_{1 \le j \le 9} \bigwedge_{1 \le d \le 9} \bigwedge_{1 \le i \le 8} \bigwedge_{i < k \le 9} (S_{ijd} \to \neg S_{kjd})$$

3. each number appears at most once in each row:
4. each number appears at most once in each 3x3 square

We leave the last two conjuncts of the encoding as exercises for the time-rich. There's an interesting discussion of the problem, and how additional constraints can be added that make it easier to solve by particular SAT algorithms, at www.cs.ox.ac.uk/joel.ouaknine/publications/sudoku05abs.html

The bottom line of the discussion there is that even the simplest encoding requires the evaluation of around 8000 (binary) truth functions; and that the brute-force method of searching would require this to be done 2^{729} times!

Note 5: 14 ☞
We will not dwell, at this point, on the potential circularities involved in describing rigorously and proving sound the logic in which we do our metaproof.

Note 6: 16 ☞
In our earlier material we used (without explaining them) rules called "premiss" and "assumption". No logical purpose would be served here by distinguishing between a (global) premiss and a (local) assumption, so here we have lumped the two "rules" together, and called them "hyp". The essential structure of our account of proofs will stay the same, but be less cluttered.

Note 7: Membership in a Collection 19 ☞
We use the obvious definition of membership of a proposition in a collection of propositions

```
(∊):: Prop -> [Prop] -> Bool
p ∊ ps = or (map (==p) ps)
```

Note 8: 19 ☞
Of course, if the proposition added to the collection as an assumption is already present in the collection (for some other reason), then it can be used outside the subproof.

Note 9: 28 ☞
The weaken rule (sometimes called **thin**) allows (but does not *require*) the elimination of irrelevant hypotheses in the search for a proof.

Exercise: Think about what happens if ϕ is $\neg\phi_1$. Does this make the weaken rule unsound?

Note 10: Notation for collections of formulae 29 ☞

- ▷ In presenting the admissible rules we have, incidentally, for the first time used the standard logician's convention of using a single greek letter (usually Γ or Δ) to stand for a collection of formulae.
- ▷ We also use the convention that when Γ and Δ stand for collection Γ, Δ stands for their "union".
- ▷ We also use the convention that when ϕ is a formula ϕ, Γ (and Γ, ϕ) stand for the collection extended by the formula.

Note 11: Exercises 29 ☞

- ▷ Exercise: give a convincing argument of the admissibility of **weaken** and **permute**.
- ▷ Exercise: give a convincing argument that adding these admissible rules to the rules of Natural Deduction would still leave the rules sound.

Note 12: 31 ☞
I am not a professional logician, but my instincts tell me that (at least in the face of logics with which one is experimenting) it would be more robust to have a (meta-proof) of a result like this without having to have recourse to completeness.

Note 13: 33 ☞
Although we have not and will not prove it formally any proof in our new calculus can be transformed into a proof in ND.

Note 14: 37 ☞
Exercise (for the very interested) Compare this linearized proof in the (single-conclusion) sequent calculus to the "pure" natural deduction proof of the same conjecture in Chapter 1 of these notes. What differences do you notice apart from the different names for the introduction rules?

Note 15: 40 ☞
Other functional languages, such as ML and F#, have explicit notations for declaring abstract types.

Note 16: 41 ☞
It would probably be more convenient (and efficient) for us to define `prove` so that it find a correct proof of a conjecture with the same conclusion, and no redundant premisses; and then use appropriate admissible rules at the end to "fix up" the proof.

It should be fairly obvious that this can be done if

$$thm(prove(\Gamma \vdash p)) = \Gamma' \boxed{\vdash} p \quad \textbf{and} \quad \Gamma' \subseteq \Gamma$$

for we just need to find an "edit" composed of insertions and deletions that takes the collection Γ' to the collection Γ, then generate the corresponding uses of the weaken, and contract rules.

Note 17: How to build a proof procedure 41 ☞
Although it is for a more basic Hilbert-style logic (in which the Natural Deduction rules are admissible) James J. Leifer gives a fine account of how to build such a proof procedure in his Oxford undergraduate project disseration. He also gives a proof of its correctness. It is a remarkable testament to his ingenuity and determination that he used a very early version of Jape to make his correctness proof completely formal.

A good starting point for reading about proof procedures and how to build them for more expressive logics is the brief account of the history of the HOL (Higher Order Logic) system at http://www.cl.cam.ac.uk/research/hvg/HOL/history.html