A Formalization of a Henkin-style Completeness Proof for Propositional Modal Logic in Lean

Bruno Bentzen

Department of Philosophy
Carnegie Mellon University

January 7, 2019

Theorem (Strong completeness)

A system of propositional logic S is (strongly) complete if for every set of premises Γ, any formula p that follows semantically from Γ is also derivable from Γ. In symbols:

$$\Gamma \vDash_S p \Longrightarrow \Gamma \vdash_S p$$

That is, every semantic consequence is also a syntactic consequence.

Theorem (Strong completeness)

A system of propositional logic S is (strongly) complete if for every set of premises Γ, any formula p that follows semantically from Γ is also derivable from Γ. In symbols:

$$\Gamma \vDash_S p \Longrightarrow \Gamma \vdash_S p$$

That is, every semantic consequence is also a syntactic consequence.

Proof sketch (Henkin)

The proof follows by (reverse) contraposition and it is thus non-constructive.

Theorem (Strong completeness)

A system of propositional logic S is (strongly) complete if for every set of premises Γ, any formula p that follows semantically from Γ is also derivable from Γ. In symbols:

$$\Gamma \vDash_S p \Longrightarrow \Gamma \vdash_S p$$

That is, every semantic consequence is also a syntactic consequence.

Proof sketch (Henkin)

The proof follows by (reverse) contraposition and it is thus non-constructive. Simply put, we want to show that if $\Gamma \nvdash_S p$, then there exists a model $\mathcal{M}$ such that $\mathcal{M}$ satisfies Γ but not p.

Proof sketch (Henkin) [cont.]

The general method of the proof is the following:

1. $\Gamma \cup \{\neg p\}$ is consistent, for $\Gamma \nvdash_S p$;
2. Extend $\Gamma \cup \{\neg p\}$ to a maximal consistent set Δ as follows:

Proof sketch (Henkin) [cont.]

The general method of the proof is the following:

1. $\Gamma \cup \{\neg p\}$ is consistent, for $\Gamma \nvdash_S p$;
2. Extend $\Gamma \cup \{\neg p\}$ to a maximal consistent set Δ as follows:

$$\Delta_0 := \Gamma \cup \{\neg p\}$$

$$\Delta_{n+1} := \begin{cases} \Delta_n \cup \{\varphi_{n+1}\} & \text{if } \Delta_n \cup \{\varphi_{n+1}\} \text{ is consistent} \\ \Delta_n \cup \{\neg\varphi_{n+1}\} & \text{otherwise} \end{cases}$$

$$\Delta := \bigcup_{n \in \mathbb{N}} \Delta_n$$

Proof sketch (Henkin) [cont.]

The general method of the proof is the following:

1. $\Gamma \cup \{\neg p\}$ is consistent, for $\Gamma \nvdash_S p$;
2. Extend $\Gamma \cup \{\neg p\}$ to a maximal consistent set Δ as follows:

$$\Delta_0 := \Gamma \cup \{\neg p\}$$

$$\Delta_{n+1} := \begin{cases} \Delta_n \cup \{\varphi_{n+1}\} & \text{if } \Delta_n \cup \{\varphi_{n+1}\} \text{ is consistent} \\ \Delta_n \cup \{\neg\varphi_{n+1}\} & \text{otherwise} \end{cases}$$

$$\Delta := \bigcup_{n \in \mathbb{N}} \Delta_n$$

3. Prove that Δ is consistent, maximal and that $\Gamma \cup \{\neg p\} \subseteq \Delta$;

Proof sketch (Henkin) [cont.]

The general method of the proof is the following:

1. $\Gamma \cup \{\neg p\}$ is consistent, for $\Gamma \nvdash_S p$;
2. Extend $\Gamma \cup \{\neg p\}$ to a maximal consistent set Δ as follows:

$$\begin{aligned}
\Delta_0 &:= \Gamma \cup \{\neg p\} \\
\Delta_{n+1} &:= \begin{cases} \Delta_n \cup \{\varphi_{n+1}\} & \text{if } \Delta_n \cup \{\varphi_{n+1}\} \text{ is consistent} \\ \Delta_n \cup \{\neg\varphi_{n+1}\} & \text{otherwise} \end{cases} \\
\Delta &:= \bigcup_{n \in \mathbb{N}} \Delta_n
\end{aligned}$$

3. Prove that Δ is consistent, maximal and that $\Gamma \cup \{\neg p\} \subseteq \Delta$;
4. Construct a model $\mathcal{M}$ s.t. $[\![\varphi]\!]_{\mathcal{M}} = 1$ iff $\varphi \in \Delta$;

Proof sketch (Henkin) [cont.]

The general method of the proof is the following:

1. $\Gamma \cup \{\neg p\}$ is consistent, for $\Gamma \nvdash_S p$;
2. Extend $\Gamma \cup \{\neg p\}$ to a maximal consistent set Δ as follows:

$$\begin{aligned} \Delta_0 &:= \Gamma \cup \{\neg p\} \\ \Delta_{n+1} &:= \begin{cases} \Delta_n \cup \{\varphi_{n+1}\} & \text{if } \Delta_n \cup \{\varphi_{n+1}\} \text{ is consistent} \\ \Delta_n \cup \{\neg\varphi_{n+1}\} & \text{otherwise} \end{cases} \\ \Delta &:= \bigcup_{n\in\mathbb{N}} \Delta_n \end{aligned}$$

3. Prove that Δ is consistent, maximal and that $\Gamma \cup \{\neg p\} \subseteq \Delta$;
4. Construct a model $\mathcal{M}$ s.t. $[\![\varphi]\!]_{\mathcal{M}} = 1$ iff $\varphi \in \Delta$;
5. Show that $[\![\Gamma]\!]_{\mathcal{M}} = 1$ but $[\![p]\!]_{\mathcal{M}} = 0$. $\square$

What do we need for a formalization of a Henkin-style completeness proof?

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;
- The contexts of S;

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;
- The contexts of S;
- The proof system of S;

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;
- The contexts of S;
- The proof system of S;
- The class of models of S;

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;
- The contexts of S;
- The proof system of S;
- The class of models of S;

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;
- The contexts of S;
- The proof system of S;
- The class of models of S;

Remark

Implicit in the previous proof sketch are the assumptions that

- S has a (not necessarily primitive) logical connective for negation;

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;
- The contexts of S;
- The proof system of S;
- The class of models of S;

Remark

Implicit in the previous proof sketch are the assumptions that

- S has a (not necessarily primitive) logical connective for negation;
- S has an enumerable language.

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;
- The contexts of S;
- The proof system of S;
- The class of models of S;

Remark

Implicit in the previous proof sketch are the assumptions that

- S has a (not necessarily primitive) logical connective for negation;
- S has an enumerable language.
- S is a classical (as opposed to constructive) logic.

What do we need for a formalization of a Henkin-style completeness proof?

The structure of the implementation

The mechanization of the proof requires four basic implementations:

- The set of well-formed formulas of S;
- The contexts of S;
- The proof system of S;
- The class of models of S;

Remark

Implicit in the previous proof sketch are the assumptions that

- S has a (not necessarily primitive) logical connective for negation;
- S has an enumerable language.
- S is a classical (as opposed to constructive) logic.

In this talk we present a formalization of a Henkin-style completeness proof for the propositional modal logic K using the Lean Theorem Prover.

In this talk we present a formalization of a Henkin-style completeness proof for the propositional modal logic K using the Lean Theorem Prover. The full source code is available at:

`https://github.com/bbentzen/metalogic/`

1. The proof: general idea
   - The aim of this talk

2. The propositional modal logic K
   - The proof system
   - Semantics

3. The mechanization of the proof
   - Some basic implementations
   - The completeness proof

1. **The proof system of K**. We shall work in a Hilbert-style system:
   1. **Axioms.**

(pl1) $\Gamma \vdash_k p \supset (q \supset p)$;
(pl2) $\Gamma \vdash_k (p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r))$;
(pl3) $\Gamma \vdash_k ((\neg p) \supset \neg q) \supset (((\neg p) \supset q) \supset p)$;
(k) $\Gamma \vdash_k (p \supset q) \supset (\Box p \supset \Box q)$.

1. **The proof system of K**. We shall work in a Hilbert-style system:
   1. **Axioms.**

      (pl1) $\Gamma \vdash_k p \supset (q \supset p)$;
      (pl2) $\Gamma \vdash_k (p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r))$;
      (pl3) $\Gamma \vdash_k ((\neg p) \supset \neg q) \supset (((\neg p) \supset q) \supset p)$;
      (k) $\Gamma \vdash_k (p \supset q) \supset (\Box p \supset \Box q)$.
   2. **Rules of inference.**

$$\text{(ax)}\ \frac{p \in \Gamma}{\Gamma \vdash_k p} \qquad \text{(mp)}\ \frac{\Gamma \vdash_k p \supset q \quad \Gamma \vdash_k p}{\Gamma \vdash_k q}$$

$$\text{(nec)}\ \frac{\vdash_k p}{\Gamma \vdash_k \Box p}$$

1. **The semantics of K**. The semantics for our modal logic will be given using Kripke semantics.

1. **The semantics of K**. The semantics for our modal logic will be given using Kripke semantics. A Kripke model is a triple $\langle \mathcal{W}, \mathcal{R}, v \rangle$ where
   - $\mathcal{W}$ is a set of objects called possible worlds;
   - $\mathcal{R}$ is a binary relation on possible worlds;
   - v specifies the truth value of a formula at a world.

1. **The semantics of K**. The semantics for our modal logic will be given using Kripke semantics. A Kripke model is a triple $\langle \mathcal{W}, \mathcal{R}, v \rangle$ where
   - $\mathcal{W}$ is a set of objects called possible worlds;
   - $\mathcal{R}$ is a binary relation on possible worlds;
   - v specifies the truth value of a formula at a world.

We define the truth of a formula at a world in a model recursively:

(var) $w \vDash p$ if $v(p, w) = 1$;

($\bot$) $w \nvDash \bot$;

($\supset$) $w \vDash p \to q$ if $w \nvDash p$ or $w \vDash p$;

($\Box$) if for every world $v \in \mathcal{W}$, $\mathcal{R}(w, v)$ implies $v \vDash p$

1. The proof: general idea
   - The aim of this talk
2. The propositional modal logic K
   - The proof system
   - Semantics
3. The mechanization of the proof
   - Some basic implementations
   - The completeness proof

1 Well-formed formulas

We define an inductive type `form` for well-formed formulas.

```
inductive form {σ :nat} : Type
| atom : var σ → form
| bot : form
| impl : form → form → form
| box : form → form
```

1 Well-formed formulas

We define an inductive type `form` for well-formed formulas.

```
inductive form {σ :nat} : Type
| atom : var σ → form
| bot : form
| impl : form → form → form
| box : form → form
```

Some useful notation:

```
notation `#` := form.atom
notation `⊥` := form.bot _
notation `∼` p := (form.impl p (form.bot _))
notation p `⊃` q := (form.impl p q)
notation `□` p := (form.box p)
notation `◇` p := (∼ (□ (∼ p)))
```

2 Contexts

We define contexts as sets of formulas, i.e., `set (form` σ`)`.

2 Contexts

We define contexts as sets of formulas, i.e., `set (form σ)`.

```
@[reducible] def ctx : Type := set (form σ)

notation '·' := {}
notation Γ '·' p := set.insert p Γ
notation Γ '⊔' Δ := set.union Γ Δ
```

Sets are predicates in Lean (*set* $\alpha := \alpha \to$ `Prop`).

3. **The proof system**

We define an inductive type `prf` that represents k-provability.

3 The proof system

We define an inductive type `prf` that represents *k*-provability.

```
inductive prf : ctx σ → form σ → Prop
| ax {Γ : ctx σ} {p : form σ} (h : p ∈ Γ) : prf Γ p
| pl1 {Γ : ctx σ} {p q : form σ} : prf Γ (p ⊃ (q ⊃ p))
| pl2 {Γ : ctx σ} {p q r : form σ} : prf Γ ((p ⊃ (q ⊃ r)) ⊃ ((p ⊃ q) ⊃ (p ⊃ r)))
| pl3 {Γ : ctx σ} {p q : form σ} : prf Γ (((∼p) ⊃ ∼q) ⊃ (((∼p) ⊃ q) ⊃ p))
| mp {Γ : ctx σ} {p q : form σ} (hpq: prf Γ (p ⊃ q)) (hp : prf Γ p) : prf Γ q
| k {Γ : ctx σ} {p q : form σ} : prf Γ ((□(p ⊃ q)) ⊃ ((□p) ⊃ (□q)))
| nec {Γ : ctx σ} {p : form σ} (h : prf · p) : prf Γ (□p)

notation Γ `⊢ₖ` p := prf Γ p
notation Γ `⊬ₖ` p := prf Γ p → false
```

4 Semantics

We implement Kripke models as structures: triples given by a domain `wrlds`, an accessibility relation `access`, and a valuation function `val`.

4 Semantics

We implement Kripke models as structures: triples given by a domain `wrlds`, an accessibility relation `access`, and a valuation function `val`.

```
@[reducible] def wrld (σ : nat) : Type := set (form σ)

variable {σ : nat}

structure model := (wrlds : set (wrld σ))
                   (access : wrld σ → wrld σ → bool)
                   (val : var σ → wrld σ → bool)
```

The truth-at-a-world relation is a function `form` $\sigma \to$ `wrld` $\to$ `bool` indexed by a model.

4 Semantics

We implement Kripke models as structures: triples given by a domain `wrlds`, an accessibility relation `access`, and a valuation function `val`.

```
@[reducible] def wrld (σ :nat) : Type := set (form σ)

variable {σ :nat}

structure model := (wrlds : set (wrld σ))
                   (access : wrld σ → wrld σ → bool)
                   (val : var σ → wrld σ → bool)
```

The truth-at-a-world relation is a function `form σ → wrld → bool` indexed by a model. It can be defined as follows:

```
noncomputable def form_tt_in_wrld (M : model) : form σ → wrld σ → bool
| (#p)    := λ w, M.val p w
|  ⊥      := λ w, ff
| (p ⊃ q) :=λ w, (bnot (form_tt_in_wrld p w)) ||(form_tt_in_wrld q w)
| (□p)   := λ w,
if
  (∀ v ∈ M.wrlds, w ∈ M.wrlds → M.access w v =tt → form_tt_in_wrld p v =tt)
then
  tt
else
  ff
```

4 Semantics

A model satisfies a formula if it is true at all possible worlds.

```
notation M `[[`p`]]` w := form_tt_in_wrld M p w

inductive stsf (M : model) (p : form σ) : Prop
| is_true (m : Π w, (M [[p]] w) = tt) : stsf

notation M `⊨ₖ` p := stsf M p
```

4 Semantics

A model satisfies a formula if it is true at all possible worlds.

```
notation M `[[`p`]]` w := form_tt_in_wrld M p w

inductive stsf (M : model) (p : form σ) : Prop
| is_true (m : Π w, (M [[p]] w) = tt) : stsf

notation M `⊨ₖ` p := stsf M p
```

A model satisfies a context if it satisfies each formula individually.

```
local attribute [instance] classical.prop_decidable

noncomputable def ctx_tt_in_wrld (M : model) (Γ : ctx σ) : wrld σ → bool :=
assume w, if (∀ p, p ∈ Γ → form_tt_in_wrld M p w = tt) then tt else ff

notation M `[[`Γ`]]` w := ctx_tt_in_wrld M Γ w

inductive sem_csq (Γ : ctx σ) (p : form σ) : Prop
| is_true (m : Π (M : model) (w : wrld σ), ((M [[Γ]] w) = tt) → (M [[p]] w) = tt) : sem_csq

notation Γ `⊨ₖ` p := sem_csq Γ p
```

Proof sketch (Henkin)

Recall the proof's strategy:

1. Show that if $\Gamma \nvdash_S p$, then $\Gamma \cup \{\neg p\}$ is consistent;
2. Extend $\Gamma \cup \{\neg p\}$ to a maximal consistent set Δ:

$$\Delta_0 := \Gamma \cup \{\neg p\}$$

$$\Delta_{n+1} := \begin{cases} \Delta_n \cup \{\varphi_{n+1}\} & \text{if } \Delta_n \cup \{\varphi_{n+1}\} \text{ is consistent} \\ \Delta_n \cup \{\neg\varphi_{n+1}\} & \text{otherwise} \end{cases}$$

$$\Delta := \bigcup_{n \in \mathbb{N}} \Delta_n$$

3. Prove that Δ is consistent, maximal and that $\Gamma \cup \{\neg p\} \subseteq \Delta$;
4. Construct a model $\mathcal{M}$ s.t. $[\![\varphi]\!]_{\mathcal{M}} = 1$ iff $\varphi \in \Delta$;
5. Show that $[\![\Gamma]\!]_{\mathcal{M}} = 1$ but $[\![p]\!]_{\mathcal{M}} = 0$. $\square$

1. **Consistency**

Consistency is defined as usual

1 Consistency

Consistency is defined as usual

```
def is_consist (Γ : ctx σ) : Prop := Γ ⊬ₖ ⊥

def not_prvb_to_neg_consist {Γ : ctx σ} {p : form σ} :
(Γ ⊬ₖ p) → is_consist (Γ · ∼ p) :=
λ hnp hc, hnp (prf.mp prf.dne (prf.deduction hc))
```

❷ Maximal consistent extensions

First we define a function $\texttt{ctx}\ \sigma \to \texttt{nat} \to \texttt{ctx}\ \sigma$.

2. **Maximal consistent extensions**

First we define a function $\mathtt{ctx}\ \sigma \to \mathtt{nat} \to \mathtt{ctx}\ \sigma$. It takes contexts and codes of formulas as arguments, and then performs consistently-wise decisions that either include that formula or its negation to context.

2 Maximal consistent extensions

First we define a function ctx $\sigma \to$ nat $\to$ ctx σ. It takes contexts and codes of formulas as arguments, and then performs consistently-wise decisions that either include that formula or its negation to context.

```
def ext_ctx_with_form (Γ : ctx σ) : nat → ctx σ :=
λ n, option.rec_on (encodable.decode (form σ) n) Γ
  (λ p, decidable.rec_on (prop_decidable (is_consist (Γ · p)))
    (λ hn, Γ · ~p)
    (λ h, Γ · p)
  )
```

2 Maximal consistent extensions

First we define a function ctx $\sigma \to$ nat $\to$ ctx σ. It takes contexts and codes of formulas as arguments, and then performs consistently-wise decisions that either include that formula or its negation to context.

```
def ext_ctx_with_form (Γ : ctx σ) : nat → ctx σ :=
λ n, option.rec_on (encodable.decode (form σ) n) Γ
  (λ p, decidable.rec_on (prop_decidable (is_consist (Γ ⸴ p)))
    (λ hn, Γ ⸴ ∼p)
    (λ h, Γ ⸴ p)
  )
```

Note: our language is enumerable.

```
instance of_form : encodable (form σ) :=
⟨ encode_form , decode_form σ , encodek_form ⟩
```

2 Maximal consistent extensions

Next, we apply `ext_ctx_with_form` to all formulas

2 Maximal consistent extensions

Next, we apply `ext_ctx_with_form` to all formulas

```
def ext_ctx_to_max_set_at (Γ : ctx σ) : nat → ctx σ :=
| 0      := ext_ctx_with_form Γ 0
| (n+1)  := ext_ctx_with_form (ext_ctx_to_max_set_at n) (n+1)
```

2 Maximal consistent extensions

Next, we apply `ext_ctx_with_form` to all formulas

```
def ext_ctx_to_max_set_at (Γ : ctx σ) : nat → ctx σ :=
| 0      := ext_ctx_with_form Γ 0
| (n+1)  := ext_ctx_with_form (ext_ctx_to_max_set_at n) (n+1)
```

thus obtaining a maximal set:

```
def ext_ctx_to_max_set (Γ : ctx σ) : ctx σ :=
⋃₀ (image (λ n, ext_ctx_to_max_set_at Γ n) {n | true})
```

3 Maximal consistent extensions are well-behaved

Γ is a subset of its maximal extension, `ext_ctx_to_max_set` Γ.

3 Maximal consistent extensions are well-behaved

Γ is a subset of its maximal extension, ext_ctx_to_max_set Γ.

```
def ctx_is_subctx_of_max_ext {Γ : ctx σ} :
Γ ⊆ ext_ctx_to_max_set Γ :=
begin
  intros _ _, apply ext_ctx_at_is_sub_max_set,
  apply ctx_is_sub_ext_ctx_at, repeat {assumption}
end
```

3 Maximal consistent extensions are well-behaved

This extension `ext_ctx_to_max_set` Γ is indeed maximal.

3 Maximal consistent extensions are well-behaved

This extension `ext_ctx_to_max_set` Γ is indeed maximal.

```
def ext_ctx_with_form_of_its_code {Γ : ctx σ} {p : form σ} :
(p ∈ ext_ctx_with_form Γ (encodable.encode p))
∨
((∼ p) ∈ ext_ctx_with_form Γ (encodable.encode p)) :=
begin
  unfold ext_ctx_with_form,
    rw (encodable.encodek p),
      simp, induction (prop_decidable _),
        simp, right, apply trivial_mem_left,
        simp, left, apply trivial_mem_left
end
```

3 Maximal consistent extensions are well-behaved

```
def ext_ctx_is_max {Γ : ctx σ} (p : form σ) :
(p ∈ ext_ctx_to_max_set Γ) ∨ ((∼p) ∈ ext_ctx_to_max_set Γ) :=
begin
  cases ext_ctx_with_form_of_its_code,
    left,
      apply ext_ctx_at_is_sub_max_set,
      apply ext_ctx_form_is_sub_ext_ctx_at,
      apply no_code_is_zero p, assumption,
    right,
      apply ext_ctx_at_is_sub_max_set,
      apply ext_ctx_form_is_sub_ext_ctx_at,
      apply no_code_is_zero p, assumption,
end
```

❸ Maximal consistent extensions are well-behaved

Maximal consistent extensions preserve consistency

```
def max_ext_preserves_consist {Γ : ctx σ} :
is_consist Γ → is_consist (ext_ctx_to_max_set Γ) :=
by intros hc nc; cases ext_ctx_lvl nc;
  apply ctx_consist_ext_ctx_at_consist; repeat {assumption}
```

❸ Maximal consistent extensions are well-behaved

Maximal consistent extensions preserve consistency

```
def max_ext_preserves_consist {Γ : ctx σ} :
is_consist Γ → is_consist (ext_ctx_to_max_set Γ) :=
by intros hc nc; cases ext_ctx_lvl nc;
  apply ctx_consist_ext_ctx_at_consist; repeat {assumption}
```

This implies that maximal consistent sets are closed under derivability.

```
def max_set_clsd_deriv {Γ : ctx σ} {p : form σ} (hc : is_consist Γ) :
(ext_ctx_to_max_set Γ ⊢ₖ p) → p ∈ ext_ctx_to_max_set Γ :=
begin
  intro h,
    cases ext_ctx_is_max p,
      assumption,
      apply false.rec,
        apply max_ext_preserves_consist, assumption,
        apply prf.mp, apply prf.ax, assumption, assumption
end
```

4 The canonical model

The set of all possible worlds $\mathcal{W}$ is the set of all maximal consistent sets.

```
def set_max_wrlds (σ :nat) : set (wrld σ) :=
image (λ w, ext_ctx_to_max_set w) {w | is_consist w }
```

4 The canonical model

The accessibility relation $\mathcal{R}$ is given via the ‘unbox’ operation

```
def unbox_form_in_wrld (w : wrld σ) :nat → wrld σ :=
λ n, option.rec_on (encodable.decode (form σ) n) ·
(λ p, form.rec_on p
  (λ v, ·) · (λ q r _ _, ·)
  (λ q _, if (□q) ∈ w then {q} else · )
)
```

4 The canonical model

The accessibility relation $\mathcal{R}$ is given via the 'unbox' operation

```
def unbox_form_in_wrld (w : wrld σ) : nat → wrld σ :=
λ n, option.rec_on (encodable.decode (form σ) n) ·
(λ p, form.rec_on p
  (λ v, ·) · (λ q r _ _, ·)
  (λ q _, if (□q) ∈ w then {q} else · )
)

def unbox_wrld (w : wrld σ) : wrld σ :=
⋃₀ (image (λ n, unbox_form_in_wrld w n) {n | true})
```

4 The canonical model

The accessibility relation $\mathcal{R}$ is given via the 'unbox' operation

```
def unbox_form_in_wrld (w : wrld σ) :nat → wrld σ :=
λ n, option.rec_on (encodable.decode (form σ) n) ·
(λ p, form.rec_on p
  (λ v, ·) · (λ q r _ _, ·)
  (λ q _, if (□q) ∈ w then {q} else · )
)

def unbox_wrld (w : wrld σ) :wrld σ :=
⋃₀ (image (λ n, unbox_form_in_wrld w n) {n | true})

noncomputable def wrlds_to_access : wrld σ → wrld σ → bool :=
assume w v, if (unbox_wrld w ⊇ v) then tt else ff
```

4 The canonical model

In particular,

```
def in_unbox_box_in_wrld {p : form σ} {w :wrld σ} :
p ∈ unbox_wrld w ↔ (□p) ∈ w :=
begin
  apply iff.intro,
    intro h, cases h, cases h_h,
      cases h_h_w, cases h_h_w_h, cases h_h_w_h_right,
        revert h_h_h, induction (encodable.decode (form σ) _),
          simp, intro, apply false.rec, assumption,
          simp, induction val,
            repeat {simp, intro h, apply false.rec, assumption},
            simp, unfold ite, induction (prop_decidable _),
              simp, intro, apply false.rec, assumption,
              simp, intro h, cases h, assumption,
    intro h, unfold unbox_wrld image sUnion,
      constructor, constructor, constructor, constructor,
        trivial, reflexivity,
        exact encodable.encode (□p),
        unfold unbox_form_in_wrld ite,
          rw (encodable.encodek □p),
          simp, induction p,
            repeat {
              induction prop_decidable _,
              contradiction, simp,
            }
end
```

4 The canonical model

Useful corollaries are:

```
def not_box_in_wrld_unbox_not_prvble {p : form σ} {w :wrld σ} (hw :w ∈ set_max_wrlds σ) :
(∼□p) ∈ w → (unbox_wrld w ⊬k p) :=
begin
  intros h nhp,
    apply all_wrlds_are_consist hw,
      apply prf.mp,
        apply prf.ax h,
        apply prf.ax (unbox_prvble_box_in_wrld hw nhp)
end

def not_box_in_wrld_to_consist_not {p : form σ} {w :wrld σ} (hw :w ∈ set_max_wrlds σ) :
(∼□p) ∈ w → is_consist (unbox_wrld w ､ (∼p)) :=
λ hn, not_prvb_to_neg_consist (not_box_in_wrld_unbox_not_prvble hw hn)
```

4 The canonical model

The valuation function v can be defined as follows:

```
noncomputable def wrlds_to_val : var σ → wrld σ → bool :=
assume p w, if w ∈ set_max_wrlds σ ∧ (#p) ∈ w then tt else ff
```

4 The canonical model

The valuation function v can be defined as follows:

```
noncomputable def wrlds_to_val : var σ → wrld σ → bool :=
assume p w, if w ∈ set_max_wrlds σ ∧ (#p) ∈ w then tt else ff
```

By putting all the pieces together we have:

```
noncomputable def canonical_model : @model σ :=
begin
  apply model.mk,
    apply set_max_wrlds,
    apply wrlds_to_access,
    apply wrlds_to_val
end
```

4 The canonical model

Now we show that truth is membership in the canonical model

```
def tt_iff_in_wrld {p : form σ} :
∀ (w :wrld σ) (wm :w ∈ set_max_wrlds σ), (canonical_model [[p]] w) =tt ↔ p ∈ w :=
begin
  induction p,
    sorry, sorry, sorry, /-- we will not discuss the atom, bot, and impl cases --/

    unfold form_tt_in_wrld, simp, intros, -- box
      apply iff.intro,
        intro h, cases all_wrlds_are_max wm □p_a, assumption,
          apply false.rec, apply max_ext_preserves_consist,
            apply not_box_in_wrld_to_consist_not wm h_1,
              apply prf.mp, apply prf.ax,
                  apply ctx_is_subctx_of_max_ext, exact trivial_mem_left,
                apply prf.ax, apply (p_ih _ (max_cons_set_in_all_wrlds
                    (not_box_in_wrld_to_consist_not wm h_1))).1,
                    apply h, assumption,
                      exact max_cons_set_in_all_wrlds
                        (not_box_in_wrld_to_consist_not wm h_1),
                      unfold canonical_model wrlds_to_access, simp,
                        intros p pm, apply ctx_is_subctx_of_max_ext,
                          apply mem_ext_cons_left, assumption,
        intros h v, unfold canonical_model wrlds_to_access,
          simp, intros ww vw rwv, apply (p_ih _ vw).2,
              apply rwv, apply in_unbox_box_in_wrld.2, assumption
end
```

4. **The canonical model**

Informally we have:

(IH) $(\texttt{canonical\_model}[[p]]w) = tt \leftrightarrow p \in w$

4 The canonical model

Informally we have:

(IH) $(\texttt{canonical\_model}[[p]]w) = tt \leftrightarrow p \in w$

$\rightarrow$ Assume that $(\texttt{canonical\_model}[[\Box p]]w) = tt$ and that $\sim \Box p \in w$.

4 The canonical model

Informally we have:

(IH) $(\texttt{canonical\_model}[[p]]w) = tt \leftrightarrow p \in w$

$\rightarrow$ Assume that $(\texttt{canonical\_model}[[\Box p]]w) = tt$ and that $\sim \Box p \in w$. But then $\texttt{unbox\_wrld w}$ $(\sim p)$ is consistent and can be extended to a possible world.

4. **The canonical model**

Informally we have:

(IH) $(\texttt{canonical\_model}[[p]]w) = tt \leftrightarrow p \in w$

$\rightarrow$ Assume that $(\texttt{canonical\_model}[[\Box p]]w) = tt$ and that $\sim \Box p \in w$. But then $\texttt{unbox\_wrld w}\ (\sim p)$ is consistent and can be extended to a possible world. It is accessible to w because $\texttt{unbox\_wrld}$ $w \subseteq ext\_ctx\_to\_max\_set(\texttt{unbox\_wrld}\ w, (\sim p))$, so p should be true at w.

4. **The canonical model**

Informally we have:

(IH) $(\texttt{canonical\_model}[[p]]w) = tt \leftrightarrow p \in w$

$\rightarrow$ Assume that $(\texttt{canonical\_model}[[\Box p]]w) = tt$ and that $\sim \Box p \in w$. But then unbox_wrld w $(\sim p)$ is consistent and can be extended to a possible world. It is accessible to w because unbox_wrld $w \subseteq ext\_ctx\_to\_max\_set(\texttt{unbox\_wrld}\ w, (\sim p))$, so p should be true at w. But $p \notin ext\_ctx\_to\_max\_set(\texttt{unbox\_wrld}\ w, (\sim p))$ because it is consistent.

4 The canonical model

Informally we have:

(IH) $(\texttt{canonical\_model}[[p]]w) = tt \leftrightarrow p \in w$

$\rightarrow$ Assume that $(\texttt{canonical\_model}[[\Box p]]w) = tt$ and that $\sim \Box p \in w$. But then $\texttt{unbox\_wrld w}$ $(\sim p)$ is consistent and can be extended to a possible world. It is accessible to w because $\texttt{unbox\_wrld}$ $w \subseteq \mathit{ext\_ctx\_to\_max\_set}(\texttt{unbox\_wrld}\ w, (\sim p))$, so p should be true at w. But $p \notin \mathit{ext\_ctx\_to\_max\_set}(\texttt{unbox\_wrld}\ w, (\sim p))$ because it is consistent.

$\leftarrow$ Assume that $\Box p \in w$. Given $v \in \texttt{M.wrld}$ and $\texttt{M.access}\ w\ v = tt$, we have to show that $(\texttt{canonical\_model}[[p]]v) = tt$. By our IH, it suffices to show that $p \in v$, but $\texttt{unbox\_wrld}\ w \subseteq v$ and $\Box p \in w$.

6 The completeness proof

We complete the proof by showing that the canonical model falsifies p at the possible world ext_ctx_to_max_set $(\Gamma, \sim p)$

```
def ctx_is_tt (Γ : ctx σ) (wm :Γ ∈ set_max_wrlds σ) :
(canonical_model [[Γ]] Γ) =tt :=
mem_tt_to_ctx_tt Γ (λ p pm, (tt_iff_in_wrld _ wm).2 pm)

def cmpltnss {Γ : ctx σ} {p : form σ} :
(Γ ⊨ₖ p) → (Γ ⊢ₖ p) :=
begin
  apply not_contrap, intros nhp hp, cases hp,
  have c : is_consist (Γ ⸴ ∼ p) :=not_prvb_to_neg_consist nhp,
  apply absurd,
    apply hp,
      apply cons_ctx_tt_to_ctx_tt,
        apply ctx_tt_to_subctx_tt,
          apply ctx_is_tt (ext_ctx_to_max_set (Γ ⸴ ∼ p)),
            apply max_cons_set_in_all_wrlds c,
            apply ctx_is_subctx_of_max_ext,

      simp, apply neg_tt_iff_ff.1, apply and.elim_right, apply cons_ctx_tt_iff_and.1,
        apply ctx_tt_to_subctx_tt,
          apply ctx_is_tt (ext_ctx_to_max_set (Γ ⸴ ∼ p)),
            apply max_cons_set_in_all_wrlds c,
            apply ctx_is_subctx_of_max_ext,
end
```

Thank you!

References

- Bruno Bentzen. Metalogic, an implementation of the metatheorems of some logics in Lean. URL: `https://github.com/bbentzen/metalogic/`. Online.