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
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
Theorem (Strong completeness)

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

$$\Gamma \models_S p \implies \Gamma \vdash_S p$$

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 $\Gamma$, any formula $p$ that follows semantically from $\Gamma$ is also derivable from $\Gamma$. In symbols:

$$\Gamma \models_S p \implies \Gamma \vdash_S p$$

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 $\Gamma$, any formula $p$ that follows semantically from $\Gamma$ is also derivable from $\Gamma$. In symbols:

$$\Gamma \models_S p \iff \Gamma \vdash_S p$$

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 $\Gamma \not\models_S p$, then there exists a model $\mathcal{M}$ such that $\mathcal{M}$ satisfies $\Gamma$ but not $p$. 
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 \( \Delta \) as follows:

\[
\Delta_0 := \Gamma \cup \{ \neg p \} \\
\Delta_{n+1} := \begin{cases} 
\Delta_n \cup \{ \phi \} & \text{if } \Delta_n \cup \{ \phi \} \text{ is consistent} \\
\Delta_n \cup \{ \neg \phi \} & \text{otherwise}
\end{cases}
\]

3. Prove that \( \Delta \) is consistent, maximal and that \( \Gamma \cup \{ \neg p \} \subseteq \Delta \);
4. Construct a model \( M \) s.t. \( J_{\phi}^K_M = 1 \) iff \( \phi \in \Delta \);
5. Show that \( J_{\Gamma}^K_M = 1 \) but \( J_p^K_M = 0 \). □
Proof sketch (Henkin) [cont.]

The general method of the proof is the following:

1. $\Gamma \cup \{\neg p\}$ is consistent, for $\Gamma \not\vdash p$;
2. Extend $\Gamma \cup \{\neg p\}$ to a maximal consistent set $\Delta$ 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 $\Delta$ is consistent, maximal and that $\Gamma \cup \{\neg p\} \subseteq \Delta$;
4. Construct a model $M$ s.t. $\mathcal{J}_{\varphi}^M = 1$ iff $\varphi \in \Delta$;
5. Show that $\mathcal{J}_{\Gamma}^M = 1$ but $\mathcal{J}_p^M = 0$. □
Proof sketch (Henkin) [cont.]

The general method of the proof is the following:

1. \( \Gamma \cup \{\neg p\} \) is consistent, for \( \Gamma \not\vdash S p \);

2. Extend \( \Gamma \cup \{\neg p\} \) to a maximal consistent set \( \Delta \) 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 \( \Delta \) is consistent, maximal and that \( \Gamma \cup \{\neg p\} \subseteq \Delta \);
The general method of the proof is the following:

1. \( \Gamma \cup \{\neg p\} \) is consistent, for \( \Gamma \not\models S\ p; \)
2. Extend \( \Gamma \cup \{\neg p\} \) to a maximal consistent set \( \Delta \) 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 \( \Delta \) is consistent, maximal and that \( \Gamma \cup \{\neg p\} \subseteq \Delta; \)
4. Construct a model \( \mathcal{M} \) s.t. \( \models p \mathcal{M} = 1 \) iff \( p \in \Delta; \)
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 $\Delta$ 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 $\Delta$ 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 $[p]_{\mathcal{M}} = 0$. □
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 $S$;

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 $S$;

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 $S$;
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;
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/
The proof: general idea
- The aim of this talk

The propositional modal logic K
- The proof system
- Semantics

The mechanization of the proof
- Some basic implementations
- The completeness proof
The proof system of $K$. We shall work in a Hilbert-style system:

**Axioms.**

1. \( \Gamma \vdash_k p \supset (q \supset p) \);
2. \( \Gamma \vdash_k (p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r)) \);
3. \( \Gamma \vdash_k ((\neg p) \supset \neg q) \supset (((\neg p) \supset q) \supset p) \);
4. \( \Gamma \vdash_k (p \supset q) \supset (\Box p \supset \Box q) \).
A Formalization of a Henkin-style Completeness Proof for Propositional Modal Logic in Lean

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.**
      - (ax) \[ p \in \Gamma \]
      \[ \Gamma \vdash_k p \]
      - (mp) \[ \Gamma \vdash_k p \supset q \]
      \[ \Gamma \vdash_k p \]
      \[ \Gamma \vdash_k q \]
      - (nec) \[ \Gamma \vdash_k p \]
      \[ \Gamma \vdash_k \Box p \]
The semantics of $K$. The semantics for our modal logic will be given using Kripke semantics.
The semantics of $K$. The semantics for our modal logic will be given using Kripke semantics. A Kripke model is a triple $\langle W, R, \nu \rangle$ where

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

We define the truth of a formula at a world in a model recursively:
1. $(\text{var})$ $w \models p$ if $\nu(p, w) = 1$;
2. $(\bot)$ $w \not\models \bot$;
3. $(\rightarrow)$ $w \models p \rightarrow q$ if $w \not\models p$ or $w \models p$;
4. $(\Box)$ if for every world $v \in \mathcal{W}$, $\mathcal{R}(w, v)$ implies $v \models p$
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
Well-formed formulas

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

```lean
inductive form (σ : nat) : Type
  | atom : var σ → form
  | bot : form
  | impl : form → form → form
  | box : form → form
```

Some useful notation:
- `# := form . atom`
- `⊥ := form . bot _`
- `∼ p := (form . impl p (form . bot _))`
- `p ⊃ q := (form . impl p q)`
- `□ p := (form . box p)`
- `⋄ p := (∼ (□ (∼ p)))`
Well-formed formulas

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

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

Some useful notation:

```lean
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., \( \text{set} \ (\text{form} \ \sigma) \).

@[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 \( \alpha \) := \( \alpha \rightarrow \text{Prop} \)).
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 $\Gamma \vdash_k p := \text{prf } \Gamma p$
notation $\Gamma \not\vdash_k p := \text{prf } \Gamma p \rightarrow \text{false}$
Semantics

We implement Kripke models as structures: triples given by a domain \( \text{wrlds} \), an accessibility relation \( \text{access} \), and a valuation function \( \text{val} \).
4 Semantics

We implement Kripke models as structures: triples given by a domain \( \text{wrl}ds \), an accessibility relation \( \text{access} \), and a valuation function \( \text{val} \).

\[
@\text{reducible} \ \text{def} \ \text{wrld} (\sigma : \text{nat}) : \text{Type} := \text{set} (\text{form} \ \sigma)
\]

\[
\text{variable} \ \{\sigma : \text{nat}\}
\]

\[
\text{structure} \ \text{model} := (\text{wrl}ds : \text{set} (\text{wrld} \ \sigma))
\]

\[
(\text{access} : \text{wrld} \ \sigma \to \text{wrld} \ \sigma \to \text{bool})
\]

\[
(\text{val} : \text{var} \ \sigma \to \text{wrld} \ \sigma \to \text{bool})
\]

The truth-at-a-world relation is a function \( \text{form} \ \sigma \to \text{wrld} \to \text{bool} \) indexed by a model.
Semantics

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

\begin{verbatim}
@[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 \texttt{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) := \(\lambda w, M.\val p w\)
| ⊥ := \(\lambda w, ff\)
| (p ⊃ q) := \(\lambda w, (\bnot (form_tt_in_wrld p w)) \| (form_tt_in_wrld q w)\)
| (□p) := \(\lambda w,\) if 
    \((\forall v ∈ M.\texttt{wrld}s, w ∈ M.\texttt{wrld}s \rightarrow M.\texttt{access} w v = tt \rightarrow form_tt_in_wrld p v = tt)\) 
then 
  tt 
else 
  ff
\end{verbatim}
Semantics

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

\[
\text{notation } M \models \phi \text{ w := form tt in wrld } M \phi w
\]

\[
\text{inductive stsf (M : model) (p : form } \sigma) : \text{Prop | is_true (m : } \Pi \text{ w, (M } \phi \text{ w) = tt) : stsf}
\]

\[
\text{notation } M \vDash_k \phi \text{ := stsf } M \phi
\]
Semantics

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

\[
\text{notation } M \left['\left[\left[p\right]\right]\right]' w := \text{form\_tt\_in\_wrld } M p w
\]

\[
\text{inductive } \text{stsf} (M : \text{model}) (p : \text{form } \sigma) : \text{Prop} \\
\mid \text{is\_true} (m : \Pi w, (M \left[\left[p\right]\right] w) = \text{tt}) : \text{stsf}
\]

\[
\text{notation } M \left['\left[k\right]\right]' p := \text{stsf } M p
\]

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

\[
\text{local attribute [instance] classical.prop_decidable}
\]

\[
\text{noncomputable def } \text{ctx\_tt\_in\_wrld} (M : \text{model}) (\rightarrow : \text{ctx } \sigma) : \text{wrld } \sigma \rightarrow \text{bool} := \\
\text{assume } w, \text{ if } (\forall p, p \in \Gamma \rightarrow \text{form\_tt\_in\_wrld } M p w = \text{tt}) \text{ then tt else ff}
\]

\[
\text{notation } M \left['\left[\Gamma\right]\right]' w := \text{ctx\_tt\_in\_wrld } M \Gamma w
\]

\[
\text{inductive } \text{sem\_csq} (\Gamma : \text{ctx } \sigma) (p : \text{form } \sigma) : \text{Prop} \\
\mid \text{is\_true} (m : \Pi (M : \text{model}) (w : \text{wrld } \sigma), ((M \left[\left[\Gamma\right]\right] w) = \text{tt}) \rightarrow (M \left[\left[p\right]\right] w) = \text{tt}) : \text{sem\_csq}
\]

\[
\text{notation } \Gamma \left['\left[k\right]\right]' p := \text{sem\_csq } \Gamma p
\]
Proof sketch (Henkin)

Recall the proof’s strategy:

1. Show that if $\Gamma \not\models S \neg p$, then $\Gamma \cup \{\neg p\}$ is consistent;
2. Extend $\Gamma \cup \{\neg p\}$ to a maximal consistent set $\Delta$:

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

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

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

3. Prove that $\Delta$ 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 $[p]_\mathcal{M} = 0$. 

□
1 Consistency

Consistency is defined as usual
Consistency

Consistency is defined as usual

```lean
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))
```
Maximal consistent extensions

First we define a function $\text{ctx } \sigma \rightarrow \text{nat } \rightarrow \text{ctx } \sigma$. 
Maximal consistent extensions

First we define a function $\text{ctx} \sigma \rightarrow \text{nat} \rightarrow \text{ctx} \sigma$. 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.
Maximal consistent extensions

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

```lean
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.
Maximal consistent extensions

First we define a function \( \text{ctx} \sigma \rightarrow \text{nat} \rightarrow \text{ctx} \sigma \). 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.

\[
\text{def} \quad \text{ext_ctx_with_form} \ (\Gamma : \text{ctx} \sigma) : \text{nat} \rightarrow \text{ctx} \sigma := \\
\lambda \ n, \ \text{option.rec_on} \ ((\text{encodable.decode} \ (\text{form} \sigma) \ n) \ \Gamma \\
(\lambda \ p, \ \text{decidable.rec_on} \ ((\text{prop_decidable} \ (\text{is_consistent} \ (\Gamma . \ p)))) \\
(\lambda \ hn, \ \Gamma . \ \sim p) \\
(\lambda \ h, \ \Gamma . \ p)
\)
\]

Note: our language is enumerable.

\[
\text{instance of_form : encodable} \ (\text{form} \sigma) := \\
(\text{encode_form} , \ \text{decode_form} \sigma , \ \text{encodek_form})
\]
Maximal consistent extensions

Next, we apply `ext_ctx_with_form` to all formulas
Maximal consistent extensions

Next, we apply `ext_ctx_with_form` to all formulas

```lean
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)
```
Maximal consistent extensions

Next, we apply `ext_ctx_with_form` to all formulas

```lean
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:

```lean
def ext_ctx_to_max_set (Γ : ctx σ) : ctx σ :=
\bigcup_0 (image (λ n, ext_ctx_to_max_set_at Γ n) {n | true})
```
Maximal consistent extensions are well-behaved

Γ is a subset of its maximal extension, \( \text{ext\_ctx\_to\_max\_set} \Gamma \).
Maximal consistent extensions are well-behaved

Γ is a subset of its maximal extension, \( \text{ext}_\text{ctx\_to\_max\_set} \Gamma \).

```lean
def ctx_isSubctx_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
```
Maximal consistent extensions are well-behaved

This extension $\text{ext}_{\text{ctx}} \text{to}_{\text{max_set}} \Gamma$ is indeed maximal.
3 Maximal consistent extensions are well-behaved

This extension ext_ctx_to_max_set Γ is indeed maximal.

\[
\text{def ext_ctx_with_form_of_its_code} \{\Gamma : \text{ctx } \sigma\} \{p : \text{form } \sigma\} : \\
(p \in \text{ext_ctx_with_form } \Gamma (\text{encodable.encode } p)) \\
\lor \\
((\sim p) \in \text{ext_ctx_with_form } \Gamma (\text{encodable.encode } p)) := \\
\begin{align*}
&\text{begin} \\
&\text{unfold ext_ctx_with_form,} \\
&\text{rw (encodable.encodek p),} \\
&\text{simp, induction (prop_decidable _),} \\
&\text{simp, right, apply trivial_mem_left,} \\
&\text{simp, left, apply trivial_mem_left} \\
&\text{end}
\end{align*}
\]
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
```
Maximal consistent extensions are well-behaved

Maximal consistent extensions preserve consistency

```lean
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}
```
Maximal consistent extensions are well-behaved

Maximal consistent extensions preserve consistency

```lean
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.

```lean
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.

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

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

```lean
def unbox_form_in_world (w : world σ) : nat → world 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 ·)
)
```

Bruno Bentzen

25 / 33
4 The canonical model

The accessibility relation $R$ is given via the ‘unbox’ operation

```lean
def unbox_form_in_world \((w : \text{world } \sigma) : \text{nat} \to \text{world } \sigma := \lambda n, \text{option.rec_on } (\text{encodable.decode } (\text{form } \sigma) n) \cdot \lambda p, \text{form.rec_on } p \cdot \lambda v, \cdot \cdot \lambda q r, \cdot \cdot \lambda q, \text{if } (\Box q) \in w \text{ then } \{q\} \text{ else } \cdot \)

def unbox_world \((w : \text{world } \sigma) : \text{world } \sigma := \bigcup_0 (\text{image } (\lambda n, \text{unbox_form_in_world } w n) \{n \mid \text{true}\})
```
The canonical model

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

```lean
def unbox_form_in_wrld (w : world σ) : nat → world 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 : world σ) : world σ :=
Union₀ (image (λ n, unbox_form_in_wrld w n) {n | true})

noncomputable def worlds_to_access : world σ → world σ → bool :=
assume w v, if (unbox_wrld w ⊇ v) then tt else ff
```
The canonical model

In particular,

```lean
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, 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:

```lean
def not_box_in_world_unbox_not_prvble {p : form σ} {w : world σ} (hw : w ∈ set_max_worlds σ) :
(¬ □ p) ∈ w → (unbox_world w ⊬ k p) :=
begin
  intros h hnp,
  apply all_worlds_are_consistent hw,
  apply prf.mp,
  apply prf.ax h,
  apply prf.ax (unbox_prvble_box_in_world hw hnp)
end

def not_box_in_world_to_consist_not {p : form σ} {w : world σ} (hw : w ∈ set_max_worlds σ) :
(¬ □ p) ∈ w → is_consistent (unbox_world w . (¬ p)) :=
λ hn, not_prvb_to_neg_consist (not_box_in_world_unbox_not_prvble hw hn)
```
The canonical model

The valuation function $\nu$ can be defined as follows:

```lean
noncomputable def worlds_to_val : var $\sigma$ $\rightarrow$ world $\sigma$ $\rightarrow$ bool :=
assume p w, if w $\in$ set_max_worlds $\sigma$ $\land$ (#p) $\in$ w then tt else ff
```
The canonical model

The valuation function $v$ can be defined as follows:

```lean
noncomputable def worlds_to_val : var σ → world σ → bool :=
assume p w, if w ∈ set_max_worlds σ ∧ (#p) ∈ w then tt else ff
```

By putting all the pieces together we have:

```lean
noncomputable def canonical_model : @model σ :=
begin
  apply model.mk,
  apply set_max_worlds,
  apply worlds_to_access,
  apply worlds_to_val
end
```
The canonical model

Now we show that truth is membership in the canonical model

```lean
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 vv rwv, apply (p.ih _ vv).2,
  apply rwv, apply in_unbox_box_in_wrld.2, assumption
end
```
4 The canonical model

Informally we have:

\[(IH) (\text{canonical\_model}[[p]]_w) = tt \leftrightarrow p \in w\]
The canonical model

Informally we have:

\[(IH) \ (\text{canonical\_model}[[p]]_w) = \text{tt} \iff p \in w\]

→ Assume that \((\text{canonical\_model}[[\Box p]]_w) = \text{tt}\) and that \(\neg \Box p \in w\).
The canonical model

Informally we have:

\[(\text{IH}) \ (\text{canonical\_model}[[p]]w) = tt \iff p \in w\]

→ Assume that \((\text{canonical\_model}[[\Box p]]w) = tt\) and that \(\neg \Box p \in w\).

But then \(\text{unbox\_wrld} \ w \cdot (\neg p)\) is consistent and can be extended to a possible world.
The canonical model

Informally we have:

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

→ Assume that (\text{canonical\_model}[[\Box p]]w) = tt and that \sim \Box p \in w.
But then unbox\_wrl1d w.(\sim p) is consistent and can be extended to a possible world. It is accessible to w because unbox\_wrl1d w \subseteq \text{ext\_ctx\_to\_max\_set}(\text{unbox\_wrl1d} w, (\sim p)), so p should be true at w.
The canonical model

Informally we have:

$$(\text{IH}) \ (\text{canonical} \_ \text{model}[[p]]w) = tt \leftrightarrow p \in w$$

$$\Rightarrow$$ Assume that $$(\text{canonical} \_ \text{model}[[\Box p]]w) = tt$$ 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 \text{ext} \_ \text{ctx} \_ \text{to} \_ \text{max} \_ \text{set}(\text{unbox} \_ \text{wrld} \_ w, (\sim p))$$, so \_ p should be true at \_ w. But \_ p \notin \text{ext} \_ \text{ctx} \_ \text{to} \_ \text{max} \_ \text{set}(\text{unbox} \_ \text{wrld} \_ w, (\sim p))$$ because it is consistent.
4 The canonical model

Informally we have:

\[(\text{IH}) \quad (\text{canonical\_model}[[p]]w) = \text{tt} \iff p \in w\]

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

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

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

```lean
def ctx_is_tt (Γ : ctx σ) (wm : Γ ∈ set_max_worlds σ) :
(cannotical_model [[Γ]] Γ) = tt :=
mem_tt_to_ctx_tt Γ (λ p pm, (tt_iff_in_world _ wm).2 pm)

def compltnss {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_worlds 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_worlds c,
  apply ctx_is_subctx_of_max_ext,
end
```

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