A Verified Tableau Prover for Modal Logic K

Minchao Wu

April 1, 2019

Correctness by Construction

The more expressive the type system, the more specifications help programmers understand their goals and the programs that achieve them.

Modal Logic K

Definition (Syntax)

The syntax of formulas is given by the following grammar:

$$\mathbb{N} ::= 0 \mid S\mathbb{N}$$
$$\varphi ::= \mathbb{N} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \mid \Box\varphi \mid \Diamond\varphi$$

We work with a simpler language NNF given by the following grammar:

$$\mathbb{N} ::= 0 \mid S\mathbb{N}$$
$$\varphi ::= \mathbb{N} \mid \neg\mathbb{N} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \Box\varphi \mid \Diamond\varphi$$

Semantics

Definition (Kripke Models)

A Kripke model is a triple (S, R, V) where S is a set of states, and $R \subseteq S \times S$ and $V \subseteq \mathbb{N} \times S$ are two binary relations.

Definition (forcing)

Let $M = (S, R, V)$ be a Kripke model. The forcing relation $\Vdash$ between a pair (M, s) where $s \in S$ and a NNF φ is defined as follows:

$$
\begin{array}{ll}
(M,s) \Vdash n & \text{if } V(n,s) \\
(M,s) \Vdash \neg n & \text{if } \neg V(n,s) \\
(M,s) \Vdash \varphi \wedge \psi & \text{if } (M,s) \Vdash \varphi \text{ and } (M,s) \Vdash \psi \\
(M,s) \Vdash \varphi \vee \psi & \text{if } (M,s) \Vdash \varphi \text{ or } (M,s) \Vdash \psi \\
(M,s) \Vdash \Box\varphi & \text{if for all } t \in S, R(s,t) \text{ implies } (M,t) \Vdash \varphi \\
(M,s) \Vdash \Diamond\varphi & \text{if there exists } t \in S, R(s,t) \text{ and } (M,t) \Vdash \varphi
\end{array}
$$

Semantics

Definition (satisfiability)

Let M be a Kripke model. A state s of M satisfies a set Γ of NNF if for all $\varphi \in \Gamma$, $(M, s) \Vdash \varphi$. A set Γ of NNF is *satisfiable* if there is a Kripke model and a state that satisfy it. Otherwise, we say that Γ is *unsatisfiable*.

Definition (literals)

A NNF formula φ is a literal if $\varphi = n$ or $\varphi = \neg n$ for some $n \in \mathbb{N}$.

Tableau

Propositional rules:

$$(id)\ \frac{n; \neg n; \Gamma}{\text{unsatisfiable}} \qquad (\wedge)\ \frac{\varphi \wedge \psi; \Gamma}{\varphi; \psi; \Gamma} \qquad (\vee)\ \frac{\varphi \vee \psi; \Gamma}{\varphi; \Gamma \mid \psi; \Gamma}$$

Modal rule:

$$(\Diamond)\ \frac{\Diamond D; \Box B; \Gamma}{\varphi_0; B \| \ldots \| \varphi_n; B}$$

where $D = \{\varphi_0 \ldots \varphi_n\} \neq \emptyset$ and Γ contains only literals if not empty.
For each rule, we call the set of formulas above the horizontal line a parent node, and sets of formulas below the horizontal line separated by $\mid$ or $\|$ children nodes.

Theorem (construction of models)

Let Γ be a set of NNF formulas. If none of the above rules are applicable to Γ, then there exist M and s such that s satisfies Γ.

Tableau

Theorem (propogation of status ($\wedge$, $\Diamond$))

All the children of a $\Diamond(\wedge)$ rule are satisfiable if and only if the parent is satisfiable.

Theorem (propagation of status ($\vee$))

One of the children of a $\vee$ rule is satisfiable if and only if the parent is satisfiable.

Definition (tableau)

Let Γ be a set of NNF. T is a tableau of Γ if T is a tree such that
(1) the root node is Γ, and
(2) all children nodes are obtained from their parent node by applying one of the tableau rules.

Theorem (termination)

Let Γ be a set of NNF. Each tableau of Γ is finite.

Tableau

Theorem (decidability)

Let Γ be a set of NNF. Whether Γ is satisfiable or not is decidable.

Proof.

Let T be a tableau of Γ. By termination, T is finite. By the id rule and the theorem of construction of models, each leaf of T is either satisfiable or not. Then by propogation theorems we have the decision procedure. $\square$

Formalization

A tableau is a procedure of propagating information from children to parents. Decidability theorem tells us that the status of each node in a tableau is always decidable. Therefore, a tableau can be viewed as a function that takes a set Γ of NNF and returns the status of Γ.

```
def tableau : Π Γ : list nnf, node Γ := sorry

inductive node (Γ : list nnf) : Type
| closed : unsatisfiable Γ → node
| open_  : {s // sat builder s Γ} → node
```

Formalization

```
structure kripke (states : Type) :=
(val : ℕ → states → Prop)
(rel : states → states → Prop)

def force {states : Type} (k : kripke states) :
    states → nnf → Prop
| s (var n)     := k.val n s
| s (neg n)     := ¬ k.val n s
| s (and φ ψ)   := force s φ ∧ force s ψ
| s (or φ ψ)    := force s φ ∨ force s ψ
| s (box φ)     := ∀ s', k.rel s s' → force s' φ
| s (dia φ)     := ∃ s', k.rel s s' ∧ force s' φ
```

Formalization

```
def sat {st} (k : kripke st) (s) (Γ : list nnf) :=
∀ φ ∈ Γ, force k s φ

def unsatisfiable (Γ : list nnf) : Prop :=
∀ (st) (k : kripke st) s, ¬ sat k s Γ
```

Carrier

What is a suitable `states : Type` ?

We have a lot of choices: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$, topological spaces, homemade inductive types...

A good `states : Type` should be helpful to formalize theorems.

Problems with Carriers Like $\mathbb{N}$

Recall the crucial theorem:

Theorem (propogation of status ($\Diamond$))

All the children of a $\Diamond$ rule are satisfiable if and only if the parent is satisfiable.

Proof (first attempt).

($\Rightarrow$): Suppose we have a sequence of pairs $\langle (M_0, s_0), \ldots, (M_n, s_n) \rangle$ such that (M_i, s_i) satisfies φ_i, B. We construct a new model M_p and a new state s_p by defining R_p and V_p as follows ...
($\Leftarrow$): ...

It is awkward to define M_p because models of children nodes know nothing about each other. Their R and V might be conflicting because their S can contain same states. Moreover, since R_i and V_i are relations, it is inconvenient to extract information from them to build a new model. $\square$

Homemade Carriers

```
inductive model
| leaf : list ℕ → model
| cons : list ℕ → list model → model
```

Intuition behind:

The constructor `leaf` takes a list of $\mathbb{N}$ representing the variables true in the state.
The constructor `cons` takes a list of $\mathbb{N}$ representing the variables true in the state, and a list of `models` representing the reachable states.
There can be structurally similar models but there won't be conflicting models.
The R and V are inferrable from within a state.

Interpretation Functions and Builders

```
def mval : ℕ → model → bool
| p (leaf v) := if p ∈ v then tt else ff
| p (cons v r) := if p ∈ v then tt else ff

def mrel : model → model → bool
| (leaf v) m := ff
| (cons v r) m := if m ∈ r then tt else ff

def builder : kripke model :=
{val := λ n s, mval n s, rel := λ s₁ s₂, mrel s₁ s₂}
```

We now have a uniform way of building a Kripke model.

Tree Models

Theorem (propogation of status ($\Diamond$))

All the children of a $\Diamond$ rule are satisfiable if and only if the parent is satisfiable.

Proof (second attempt).

($\Rightarrow$): Suppose we have a list of pairs $\langle (M_0, s_0), \ldots, (M_n, s_n) \rangle$ such that (M_i, s_i) satisfies φ_i, B. We construct a new state simply by applying `cons` to the sequence of states $\langle s_0, \ldots, s_n \rangle$. The corresponding model M_p is given by the `builder`. It can be shown that the new state satisfies the parent.

($\Leftarrow$): ...

$\square$

Computing with the Modal Rule

How do we describe the children of the modal rule?

```
def unmodal (Γ : list nnf) : list (list nnf) :=
list.map (λ d, d :: (unbox Γ)) (undia Γ)
```

We want to call `tableau` recursively on all the elements of `unmodal Γ`.

Is `list.map` good for the task?

Problems with list.map

list.map computes everything in one shot. But we need early termination if one unsatisfiable child is found.

The compiler has no idea why `list.map tableau (unmodal Γ)` terminates.

We need a customized version of `list.map`.

Termination

Assume that we have the following proof:

```
def unmodal_size (Γ : list nnf) : ∀ (i : list nnf),
i ∈ unmodal Γ → (node_size i < node_size Γ) := sorry
```

The compiler does not know it's terminating, because in the term `list.map tableau (unmodal Γ)`, `tableau` is passed as an argument and not applied to anything.

Solution:

```
list.map (λ x, tableau x) (unmodal Γ)
```

Termination

Still not good enough because `x` is arbitrary.

Let's add more stuff to it.

```
list.map (λ x h, tableau x) (unmodal Γ)
```

`h` is an evidence (i.e., proof) saying that the size of `x` is smaller than the size of Γ, which is the original set of NNF that `tableau` is being called on.

What is the type of `h`?

`h : p x` where `p : α → Prop` is a predicate if we abstract the concrete meaning here.

Termination

Now `list.map` is not type-correct, but we can fix this by defining a customized version.

```
def tmap {p : list nnf → Prop}
    (f : Π Γ, p Γ → node Γ): Π Γ : list (list nnf),
    ?
```

`tmap` should also be designed in a way that it terminates as soon as an unsatisfiable child is found.

If an unsatisfiable child is found, it returns the child with a proof of it being unsatisfiable. Otherwise it returns a list of models with a proof saying that the ith element of the list satisfies the ith child.

Termination

```
def tmap {p : list nnf → Prop}
    (f : Π Γ, p Γ → node Γ): Π Γ : list (list nnf),
     psum {i // i ∈ Γ ∧ unsatisfiable i}
          {x : list model // batch_sat x Γ}
```

Is this function definable (provable) ? Not yet.

To define it we need to do recursion on Γ and call the function f on the head of Γ. When we do this, $P(head)$ becomes a proof obligation, but we can infer from nowhere that it holds.

Termination

So we need one more assumption.

```
def tmap {p : list nnf → Prop}
    (f : Π Γ, p Γ → node Γ): Π Γ : list (list nnf),
     (∀ i∈Γ, p i) → -- necessary and provable
     psum {i // i ∈ Γ ∧ unsatisfiable i}
          {x : list model // batch_sat x Γ}
```

Termination

```
...
| [] h := psum.inr ⟨[], bs_nil⟩
| (hd :: tl) h :=
match f hd (h hd (by simp)) with
| (node.closed _ pr _) := psum.inl ⟨hd, by simp, pr⟩
| (node.open_ w₁) :=
  match tmap tl (λ x hx, h x (by simp [hx])) with
  | (psum.inl uw) :=
          begin
            left, rcases uw with ⟨w, hin, h⟩,
            split, split, swap, exact h, simp [hin]
          end
  | (psum.inr w₂) := psum.inr ⟨(w₁.1::w₂),
    bs_cons _ _ _ _ w₁.2 w₂.2⟩
  end
end
```

Termination

```
def tmap {p : list nnf → Prop}
    (f : Π Γ, p Γ → node Γ): Π Γ : list (list nnf),
     (∀ i∈Γ, p i) → -- necessary and provable
     psum {i // i ∈ Γ ∧ unsatisfiable i}
          {x : list model // batch_sat x Γ}
```

`tmap` is essentially the modal rule formalized in a computational (runnable) way.

Backjumping

Recall the $(\vee)$ rule:

$$(\vee)\ \frac{\varphi \vee \psi; \Gamma}{\varphi; \Gamma \mid \psi; \Gamma}$$

We call the formula $\varphi \vee \psi$ in the parent a principal formula, and φ the left principal formula, ψ the right principal formula respectively.

The idea of backjumping is that if the left child of the rule is unsatisfiable, there is a chance that the right child is also unsatisfiable.

This happens when the principal formula is *not responsible* for a contradiction.

Backjumping

Define responsibility for each of the rules. Each node induces a set of formulas, called a marking set, representing the formulas responsible for contradictions.

Definition (responsibility)

$$(id)\ \frac{n; \neg n; \Gamma}{\text{unsatisfiable}}$$

Marking set: $M = \{n, \neg n\}$.

$$(\wedge)\ \frac{\varphi \wedge \psi; \Gamma}{\varphi; \psi; \Gamma}$$

Let M be the marking set of the child.

$$M_{parent} = \begin{cases} M \cup \{\varphi \wedge \psi\} & \text{if } \varphi \in M \text{ or } \psi \in M \\ M & \text{otherwise} \end{cases}$$

Backjumping

Definition (responsibility contd.)

$$(\vee)\ \frac{\varphi \vee \psi; \Gamma}{\varphi; \Gamma \mid \psi; \Gamma}$$

Let M_1, M_2 be the marking sets of the left and right child respectively.

$$M_{parent} = \begin{cases} M_1 \cup M_2 \cup \{\varphi \vee \psi\} & \text{if } \varphi \in M_1 \text{ or } \psi \in M_2 \\ M_1 \cup M_2 & \text{otherwise} \end{cases}$$

$$(\Diamond)\ \frac{\Diamond D; \Box B; \Gamma}{\varphi_0; B \| \ldots \| \varphi_n; B}$$

Let C be the first child which is unsatisfiable, and M_c the corresponding marking set.

$$M_{parent} = \Diamond(C.head) \cup \Box(C.tail \cap M_c)$$

Backjumping

$$(\vee)\ \frac{\varphi \vee \psi; \Gamma}{\varphi; \Gamma \mid \psi; \Gamma}$$

Theorem (jumping)

If the left principal formula (i.e., φ) in the $(\vee)$ rule is not in the marking set of the left child, then the parent is unsatisfiable.

Theorem (marking property)

For each node $\varphi; \Gamma$, if φ is not in its marking set, then Γ is unsatisfiable.

Backjumping

The marking property in its original form is awkward to prove.
Think about the $(\wedge)$ rule:

$$(\wedge)\ \frac{\varphi \wedge \psi; \Gamma}{\varphi; \psi; \Gamma}$$

Assumption: $\varphi \wedge \psi$ is not marked.
IH: the child satisfies the marking property.
Goal: Γ is unsatisfiable.

With structural induction we can show that $\psi; \Gamma$ is unsatisfiable.
But we need induction on the height of tableau trees to go further.

Now we have to cook up customized inductive principles which are likely to be inelegant.

Backjumping

Theorem (marking property revisited)

For each node Γ, if a subset $\Delta \subseteq \Gamma$ contains nothing in the marking set, then $\Gamma - \Delta$ is unsatisfiable.

Proof.

By structural induction on the tableau. □

But we have a big problem for the formalization: there is nothing to do induction on!

Recall the type of our `tableau`:

```
def tableau : Π Γ : list nnf, node Γ := sorry
```

It is a pure function.

Backjumping

Observation:

The function calls itself recursively. This sounds like a computational induction.

Idea:

Encode the inductive hypothese into the recursive calls. The tableau computes the proof. Each node propagates its proof to its parent so that the parent can use the proofs to construct its own proof.

With a strong type system, we can do this as proofs can be treated as data, and the property can be depending on a term (i.e., the marking set).

Backjumping

The inductive hypothesis is defined as:

```
def pmark (Γ m : list nnf) :=
∀ Δ, (∀ δ ∈ Δ, δ ∉ m) →
Δ <+ Γ →
unsatisfiable (list.diff Γ Δ)
```

We now force each closed node to carry a marking set with a proof of pmark.

```
inductive node (Γ : list nnf) : Type
| closed : Π m, unsatisfiable Γ → pmark Γ m → node
| open_  : {s // sat builder s Γ} → node
```

Backjumping

Now it is safe to write our new rule:

$$(J)\ \frac{\varphi \vee \psi; \Gamma}{\varphi; \Gamma} \quad \text{if } \varphi \notin M_{child} \text{ and } \varphi; \Gamma \text{ is unsatisfiable}$$

Note that since this is a new rule, we need to go back to define its parent's marking set and show that it respects the marking property.

Backjumping

Now we are ready to define our `tableau`.

It applies propositional rules and the jumping rule repeatedly to Γ. If a contradiction is found, it propagates the proof to the parent node. If Γ is fully saturated and contains at least one diamond, it applies the modal rule. If it reaches a node containing a `model_constructible` Γ, it constructs a Kripke model and propagates it to the parent node. This procedure terminates because it is provable that each recursive call is called on a smaller node.

Applications

```
structure topo_model (α : Type) extends
     topological_space α :=
(v : ℕ → set α)
(is_alex : ∀s, (∀t∈s, is_open t) → is_open ( ⋂₀ s))

def topo_force {α : Type} (tm : topo_model α) :
    α → nnf → Prop
| s (var n)     := tm.v n s
| s (neg n)     := ¬ tm.v n s
| s (and φ ψ)   := topo_force s φ ∧ topo_force s ψ
| s (or φ ψ)    := topo_force s φ ∨ topo_force s ψ
| s (box φ)     := @interior _ tm.to_topological_space
        (λ a, topo_force a φ) s
| s (dia φ)     := @closure _ tm.to_topological_space
        (λ a, topo_force a φ) s
```

Applications

```
def topo_to_kripke {α : Type} (tm : topo_model α) :
    kripke α :=
{ rel := λ s t, s ∈ @closure _
           tm.to_topological_space {t},
  val := λ n s, tm.v n s }

theorem trans_force_left {α : Type}
{tm : topo_model α} :
Π {s} {φ : nnf}, (topo_force tm s φ) →
force (topo_to_kripke tm) s φ := sorry
```

Thoughts

The whole formalization is a single definition of the function `tableau`. In this sense it is really a program. We can see that proofs and programs coerce.

Constructing a (proof) term of the type of `tableau` amounts to writing a program that has the same return type. In particular, we can make such a program efficient via the proofs it carries.

In a strong type system, proof searching and program synthesis are almost the same.

Dependent types are used almost everywhere in this formalization, especially for the purpose of carrying proofs to prove correctness. They also provide a trick for telling the compiler about termination.