Theorem Prover Combinators

by Sven Nilsen, 2017

In this paper I introduce some concepts required to build theorem prover combinators: A technique for combining smaller context-dependent solvers into larger multi-context solvers. I formalize the modality for referencing rules and facts from sub-solvers, including a new axiom that preserves meaning.

The major result of this paper is an axiom that preserves meaning across solvers:

(p in X ∧ p in Y) <=> (p in X <=> p in Y)

A theorem prover combinator is a way to create powerful tools for automated reasoning using simple building blocks. The motivation to use a theorem prover combinator is that each solver can be consisting of a simple set of rules, and the complexity can be abstracted away by moving lots of information into how the solvers are connected. It makes solvers reusable and composable.

For example, one person A knows a person B, then A and B know each other:

KnowEachOther(A, B)	world 1
Know(A, B) ∧ Know(B, A)	world 2
KnowEachOther(A, B) <=> Know(A, B) ∧ Know(B, A)	world 1 <=> world 2

This knowledge representation can be inferred across what logicians call “worlds”. Each world contains facts and rules that can vary from world to world. Think of it as the sky can be blue in one place on the earth while red in another place, but theorem prover combinators do not use meaningless words: If the sky is blue in two places, then it actually means the same thing!

A rule in world 2 can be reduced to world 1. Since world 1 has a more compact representation, one could improve performance by reasoning about bidirectional knowledge relationships in world 1 and directional knowledge relationships in world 2. The theorem prover combinator of world 1 and 2 should be able to solve all problems even when all directional knowledge relationships are erased from world 2. If world 1 and world 2 uses the same language, the cost can be known and reasoned about at a meta level.

In the simple example above, there is only a reduction of 2N facts to N. More advanced applications might e.g. reduce from N^2 to N or perhaps even from infinity to a single fact. This means the possible performance gain is potentially super-exponential, and since it is easy to combine theorem provers this way, one might even solve problems that otherwise would be out of reach for the human mind.

Motivation to creating theorem prover combinators:

1. Optimize performance without human input
    1. By using simpler or faster representations of redundant concepts
    2. By deriving rules of inference in a desired world
    3. By erasing slower rules of inference from memory
2. Simplify construction of complex solvers
    1. Programmers can create solvers that e.g. focuses on a single category (category theory)
    2. Building blocks for solver become easier to understand and verify
    3. Engineering complex solvers can be done by putting together building blocks

It is important to notice the significant motivation for theorem prover combinators, because the design restrictions to make it possible might seem unwanted or undesirable. There is a tradeoff in this design space that stand-alone theorem provers do not have. The rest of this paper is introducing terminology and concepts to enable the realization of such combinators.

To decouple theorem provers into smaller building blocks, one needs to introduce modality. Instead of truth values that are universal, one uses truth values that are observed in a particular world:

```
A in X          `A` is true in world `X`
B in Y          `B` is true in world `Y`
```

This might seem confusing at first, but remember from the first example that `KnowEachOther(A, B)` is bidirectional while `Know(A, B)` is directional. Thus, it is actually less about truth and more about encoding knowledge graphs in different ways with different axioms that apply to different kinds of graphs. When describing how nodes in one graph relates to nodes in another graph and the operations one them, the `in` keyword is used to assign the world to the statement.

```
( A ∧ B ) in X <=> ( A in X ∧ B in X )
```

Alternative notation:

```
X { A ∧ B } <=> X { A } ∧ X { B }
```

For example:

```
GoesToSchool(Lisa, SpringfieldSchool) in Tomorrow
Tomorrow { GoesToSchool(Lisa, SpringfieldSchool) }
```

When no world is specified, it is assumed to be universally true across all worlds in context. This means that rules for how rules are transformed between worlds do not need to be assigned a world. It also makes it easy to define local rules for a theorem provers and automatically solve compatibility when combining it with other solvers.

Sometimes it is useful to refer to transformation rules implicitly.
This is written with a double `in` or `→`:

```
A in X in Y
X → Y { A }
```

A transformation rule is not guaranteed to exist. If no such rule is given, then it is not possible to transfer the rule from one world to another.

The identity world transformation:

p in X in X <=> p in X
X { X { p } } <=> X { p }

There is a special rule when there is no loss of information when transferring rules:

p in X in Y in Z <=> p in X in Z
Z { Y { X { p } } } <=> X → Y → Z { p } <=> X → Z { p }

The transformation of a fact or rule can be probabilistic:

p in X in Y(c) => p in Y(0.5c)
Y(c) { X { p } } => Y(0.5c) { p }

Here `Y` is assigned a confidence factor that is 0% when it is unknown whether something is true and 100% when it is certainly true. When `Y` is confident something is false, the confidence factor can be negative.

Notice that it does not matter that one solver is 100% confident in something that might be wrong, as long as the results are interpreted correctly. For example, one solver can be used to simulate a biased theorem prover on purpose and used when the goal is to find what the biased theorem prover will prove, while acting on that meta-information instead of using the information directly.

Implication follows a special rule, where there ought to be translation rule:

(p in X => q in Y) ∧ (q => r) in Y => (p => r in Y) in X
(X { p } => Y { q }) ∧ Y { q => r } => X { q => Y { r } }

Notice that erasing the text in color gives normal transport for implication in logic.

Equivalence follows a rule that can directly transport a fact or rule from one world to another without any translation:

(p in X <=> q in Y) ∧ (p <=> r) in X => r in Y

One can use the following rule to derive logical equivalent facts or rules:

(p => q) ∧ (q => p) <=> (p <=> q)

Finally, when the same fact or rule is made in two worlds, it is implicitly telling that this is transferrable:

$$(p \text{ in } X \wedge p \text{ in } Y) \Leftrightarrow (p \text{ in } X \Leftrightarrow p \text{ in } Y)$$

This is because a statement in the same identical language semantics has a unique meaning across all possible worlds.