CSC324 Assignment 4. Theorem Proving in miniKanren

In this assignment, we will write a basic proof checker for purely implicational minimal logic in miniKanren. Like how we ran a miniKanren interpreter "backwards" to synthesize programs, we will run this proof checker "backwards" to create a rudimentary theorem prover.

Before we start, remember these general guidelines for all labs and assignments:

- You may not import any libraries or modules unless explicitly told to do so.
- You may not use the function "eval" for any lab or assignment unless explicitly told otherwise.
- You may not use any iterative or mutating functionality unless explicitly allowed. Remember that a big goal of this course is to learn about different models and styles of programming!
- You may write helper functions freely; in fact, you are encouraged to do so to keep your code easy to understand.

Breaking any of these above rules can result in a grade of 0.

- Code that cannot be imported (e.g., due to a syntax error, compilation error, or runtime error during import) will receive a grade of zero! Please make sure to run all of your code before your final submission, and test it on the Teaching Lab environment (which is the environment we use for testing).
- The `(provide ...)` and `module (...) where` code in your files are very important. Please follow the instructions, and don't modify those lines of code! If you do so, your code will not be able to run and you will receive a grade of zero.

Starter code

- `a4.rkt`
- `mk.rkt`

You will only be submitting `a4.rkt` and not any of the other files. We'll be supplying our own version of the remaining files to test your code.

Background: Purely Implicational Minimal Logic

A **logic** describes a way to write down **propositions** (statements that make some claim, which can be true or false) and possible **proofs** or ways to explain how a proposition is true. The logic that we will be implementing is called **purely implicational minimal logic**, chosen for its simplicity.

Propositions

A propositions is a logical statements that makes some claim. In our logic, the only types of propositions that we allow are:

- **Constants**, like `A`, `B`, or other symbols.
- **Implications**, like `(A -> B)`, which can be verbalized as "A implies B".

The left-hand side of an implication is called its **hypothesis**, and the right-hand side is called its **conclusion**. For example, the hypothesis of `(A -> B)` is `A`, and the conclusion of `(A -> B)` is `B`.

Propositions will be represented as data with a shape given by the following grammar:

```
<proposition> :: = IDENTIFIER | '(' <proposition> '->' <proposition> ')'
```

That is, a proposition may either be a Racket symbol, or a 3-element list where the second element is the symbol `->`, and the first and third elements are propositions. Note the recursive structure.

For example:

- `A`, `B`, `foo`, `bar`, are all propositions.
- `(foo -> bar)` is a proposition.
- `(A -> ((B -> C) -> D))` is a proposition.

Proofs

A Proof is a logical argument that explains how a proposition is true. You might intuitively see that the propositions `(A -> A)` and `(A -> (B -> A))` are both true. However, in a logic system, we use precise rules about the types of reasoning allowed. In particular, there are three rules that we can use in a proof:

- The rule `use` makes use of a proposition that is already available as an assumption.
- The rule `assume` makes a new proposition available for use as an assumption.
- The rule `modus-ponens` uses the proof of an implication's hypothesis to prove the implication's conclusion. This rule is sometimes called the *elimination rule* since it it is used to "eliminate" implications.

Intuitively, if a proposition is currently available as an assumption, we can prove that same proposition by using the assumption. We call this rule "use". For example, if `A` is available as an assumption, then we can simply prove `A` by using that assumption.

If we can prove the conclusion of an implication while assuming its hypothesis, then we can prove the implication itself. This follows the introduction rule for implication, which we call "assume". For example, we can prove the implication `(A -> A)` because we can prove the implication's conclusion `A` by using the assumption `A`, which is the implication's hypothesis.

If both an implication and its hypothesis can be proven, then the implication's conclusion can also be proven. This follows the elimination rule for implication, which we call "modus ponens". For example, if we've proved both `(A -> B)` and `A`, then we can use those proofs to prove `B` by invoking "modus ponens".

Proofs will be represented as data with a shape given by the following grammar:

```
<proof> ::= '(' use <proposition> ')'
          | '(' assume <proposition> <proof> ')'
          | '(' modus-ponens <proof> <proof> ')'
```

That is, a proof is always a list of at least two elements, where the first element is a symbol indicating which proof rule is being used. Depending on the kind of rule being used, a proof may reference propositions and/or sub-proofs.

For example:

- `(assume A (use A))` is a proof of `(A -> A)`.
- `(assume A (assume (A -> B) (modus-ponens (use A) (use (A -> B)))))` is a proof of `(A -> ((A -> B) -> B))`.
- `(assume (C -> D) (assume C (modus-ponens (use C) (use (C -> D)))))` is a proof of `((C -> D) -> (C -> D))`.
- `(assume (C -> D) (use (C -> D)))` is also a proof of `((C -> D) -> (C -> D))`.

As always, we use lists in Racket to represent propositions and proofs. In Racket, if we want to represent the proof `(assume A (use A))`, we write `'(assume A (use A))` or `(list 'assume 'A (list 'use 'A))`.

Task 1: Using a proof checking function in Racket

This question came up in Prof. Michael Miljanovic's CSC236 course:

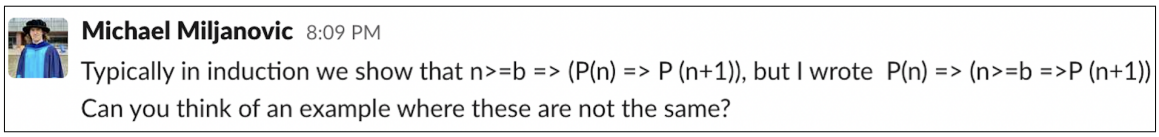

Figure 1: Message from Prof. Miljanovic

Our first task is to help convince Prof. Miljanovic that he did not make a mistake, **and that we can write a proof for the theorem `((A -> (B -> C)) -> (B -> (A -> C)))`.**

Write the proof in the grammar provided above, and assign it to the variable `my-proof` in the file `a4.rkt`.

Unfortunately, Prof. Miljanovic is not easily convinced, so we will use a **proof checker** to check our proof. In particular, a proof checker function `proof?` is written for you in the starter code.

The function `proof?` takes a proposition and a proof, and returns whether the second argument is a valid proof of the first argument. Here are some example calls to `proof?`:

```
> (proof? '(A -> A) '(assume A (use A)))
#t
> (proof? '(A -> B) '(assume A (use A)))
#f
> (proof? 'A '(use A))
#f  ; (because `A` is not available as an assumption where it is used)
```

We will be testing to see whether `(proof? '((A -> (B -> C)) -> (B -> (A -> C))) my-proof)` returns true.

Advice: You may find it helpful to study some of the proofs that are provided to you. The `modus-ponens` may be unintuitive to use. You may also choose to skip to Task 2, and synthesize the proof in Task 2 instead.

Task 2: A proof checking relation

In this part of the assignment, we will define the relation `proofo` in miniKanren. The miniKanren relation `proofo` is analogous to the Racket procedure `proof?`.

- The first argument is a proposition.
- The second argument is a proof.
- The relation holds if the second argument is a valid proof of the first argument.

For example:

- `(proofo '(A -> A) '(assume A (use A)))` should succeed.
- `(proofo '(A -> B) '(assume A (use A)))` should fail.
- `(proofo 'A '(use A))` should also fail (because `A` is not available as an assumption where it is used).

Because `proof?` returns a boolean, `proofo` does not need an extra argument for representing the result. Instead, `proofo` will succeed when `proof?` would return `#t`, and fail when `proof?` would return `#f`.

You should implement `proofo` in terms of a helper relation `proof-helpero` that, in addition to the proposition and proof arguments, also maintains a list of the currently-valid assumptions.

We implemented relations by referencing their functional equivalents. You may find it helpful to think of `proof-helpero` as the relational form of `proof/asmpt`, and think of the parameter `prop` of `proof-helpero` as the return value of `proof/asmpt`.

Alternatively, you may choose to implement `proof-helpero` by reasoning about how the proposition, proof, and assumptions should relate to one another:

- Start by considering the three different kinds of proofs. You will want to treat these separately. In the functional setting, you may want to use `cond`, `if`, or pattern matching to identify these cases. What miniKanren construct is equivalent?
- For `use` proofs, what would need to be true about the proposition that you are trying to prove (vs the proposition that you are trying to use)? What would need to be true about the proposition that you are trying to use (vs the assumptions that you have?)
- For `assume` proofs, what does your proposition need to look like? What sub-proof would you need to check? What `fresh` variables do you need to introduce in order to express these constraints?
- For `modus-ponens` proofs, you will need to separately prove the two sub-proofs. But what propositions do the hypothesis and conclusion sub-proofs prove? What assumptions can you use in these sub-proofs? What `fresh` variables do you need to introduce in order to express these constraints?

As always **trust the recursion!**

Task 3: Prove theorems by running "backwards"

Define a theorem prover `prove` in Racket, which runs the `proofo` relation **backwards** to attempt to prove a theorem. In other words, `prove` should be a procedure that takes a proposition and returns either a proof for the proposition, or `#f`. When there is no proof for the proposition, the procedure may also fail to terminate.

- The argument is a proposition.
- If the proposition has any proof, a proof is returned.
- If the proposition cannot be proven, then either `#f` is returned, or the procedure will fail to terminate.

You should define `prove` in terms of a miniKanren `run 1` expression that makes use of `proofo`. Remember that `run` returns a list of answers, so you will need to unwrap any answer produced.

You may wish to use `prove` to find proofs for the following theorems:

- `(A -> (B -> A))`
- `(A -> (B -> B))`
- `((A -> B) -> (B -> C) -> (A -> C))`
- `(((A -> (B -> C)) -> C) -> ((B -> (A -> C)) -> C))`

Theorem Proving and Programming Languages

Very astute readers might see some connection between the theorem checker `proof?` (or even `proofo`) and the interpreter for the Lambda Calculus that we wrote back in weeks 3-4 of class. There are three kinds of expressions in the Lambda Calculus, and there are three kinds of theorems our minimal logic. The "use" rule is similar to identifier lookup, the "assume" rule is similar to lambda expressions, and the "modus-ponens" rule is similar to application.

It turns out that there is a correspondence between proofs and programs—and in particular, proof checkers and type checkers. You can find out more by reading about the Curry-Howard correspondence.