More on rule-based proofs

Implication introduction, elimination, and hypothesis

You may have noticed that in the various examples we never used the rules for implication introduction and elimination, and hypothesis. They are *implicit* in the proof formulation.

Implication elimination is replaced by *rule application*. Consider the simple example of *modus ponens*:

```
mp : {A B : Prop} -> (f : A -> B) -> (a : A) -> B
mp {A} {B} f a = b where
  b : B
  b = f a
```

> ***Syntax aside:*** *Note that I wrote* `f : A -> B` *and* `a : A` *in the proposition itself, annotating the two hypotheses with their proofs. This is not necessary, but it is allowed. I am only doing it here for clarity.*

The last line `b = f a` is interpreted as *"Define proof* `b` *as the application of rule* `f` *to proof* `a` *"*. This is essentially the same as implication elimination, except that it has no name, but is written as rule application. We can make the rule application syntactically more

like function application by using brackets, although it's not necessary:

```
...
  b : B
  b = f (a)
```

This rule application is indeed just like function application in any programming language (or *method* in Java, or *procedure* in older languages). The only *syntactic* differences are:

a. the brackets are optional, i.e. `f (a)` is the same as `f a`;
b. multiple arguments are not bracketed, i.e. instead of `f (a, b, c)` we always write `f a b c`;
c. brackets are only required to disambiguate, e.g. `f g h` is a function `f` applied to arguments `g`, `h`, whereas `f (g h)` is a function `f` applied to argument `g h`;
d. arguments can be themselves functions (rules) because, remember, `f : A -> B` can be seen dually as a *"proof of* `A->B` *"* and as a *"rule mapping proofs of* `A` *into proofs of* `B`, as convenient.

If the following explanation is confusing, ignore it. But some people have found it useful:

> *Now going back to implication elimination, we could use the modus ponens rule as a "named" implication elimination rule. For example,*

```
comp : {A B C : Prop} -> (f : A -> B) -> (g : B -> C) -> (
```

```
a : A) -> C
comp {A} {B} {C} f g a = c where
  b : B
  b = f a
  c : C
  c = g b
```

> *could be written instead as:*

```
comp : {A B C : Prop} -> (f : A -> B) -> (g : B -> C) -> (
a : A) -> C
comp {A} {B} {C} f g a = c where
  b : B
  b = mp f a
  c : C
  c = mp g b
```

> *so that instead of directly applying `f` and `g` we always only apply the `mp` rule, with arguments `f` and `a` and, respectively, `g` and `c`.*

> *But this seems a bit overcooked since `f`, `g` are themselves rules. It is a bit like in the COBOL language where to call a function `F` with argument `A` the syntax is `CALL F A`. The `CALL` operation is a named function-application operation, just like `md` is a rule-application operation.*

Sometimes implication introduction is made redundant by the way we

write our propositions. Going back to `mp`:

```
mp : {A B : Prop} -> (f : A -> B) -> (a : A) -> B
mp {A} {B} f a = b where
  b : B
  b = f a
```

we could re-associate the proposition as

```
mp : {A B : Prop} -> (f : A -> B) -> (A -> B)
mp {A} {B} f = f
```

You can see how the two versions of `mp` compare:

- the first has an extra hypothesis `a : A` and needs to return a `b : B`
- the second has to return a rule (proof of implication) `f : A -> B`

If you insist on returning a rule, in the case of `mp` it works, but in the case of `comp`:

```
comp : {A B C : Prop} -> (f : A -> B) -> (g : B -> C) -> (
A -> C)
comp {A} {B} {C} f g = ? where
  b : B
  b = f a
```

```
  c : C
  c = g b
```

you would now really need something akin to *implication introduction* to formulate the rule for `A -> C`. This is done by defining it with a rule clause:

```
comp : {A B C : Prop} -> (f : A -> B) -> (g : B -> C) -> (
A -> C)
comp {A} {B} {C} f g = h where
  h : A -> C
  h a = c where
    b : B
    b = f a
    c : C
    c = g b
```

So you can see how “rule definitions” correspond to the implication introduction.

Finally the same rule-based style automatically extracts the hypotheses so that an explicit hypothesis rule is not required. For example in `comp` we can immediately see that all the available hypotheses are `f : A -> B`, `g : B -> C` and `a : A`.

Negation

Let us examine a little closer the difference between negation formulations seen in the first part of this module and seen now (`not A = A -> Falsity`).

In natural deduction the rule for implication introduction is

```
A
.
.
.
B
------
A -> B
```

This makes a proof of a negation simply an instance of this rule:

```
A
.
.
.
Falsity
--------------------
A -> Falsity = not A
```

So that a separate rule is not needed. The only negation-specific rules are `EFQ : Falsity -> A` (*Ex ~~nihilo~~ falso quodlibet*) and `DNE : (not`

`(not A)) -> A`.

Moreover, formulating negation as an implication gives us the extra advantage of having an extra hypothesis.

Consider *double negation introduction*:

```
DNI : {A : Prop} : A -> (not (not A))
```

This is rather difficult to approach, isn't it? However, if we unpack the negation:

```
DNI : {A : Prop} : A -> ((A -> Falsity) -> Falsity)
```

and remove redundant brackets

```
DNI : {A : Prop} : A -> (A -> Falsity) -> Falsity
```

This is simply an instance of *modus ponens*, with the rule and the premise swapped!

```
DNI : {A : Prop} : (a : A) -> (f : A -> Falsity) -> Falsit
y
DNI a f = x where
    x : Falsity
    x = f a
```

If we re-pack the negation this this the same as

```
DNI : {A : Prop} : (a : A) -> (not (not A))
DNI a f = x where
    x : Falsity
    x = f a
```

The dual view of `f : A -> B` as “proof” or “rule” is helpful and it applies to negation as well. Consider the rule for *triple negation elimination*:

```
TNE : {A : Prop} -> (not (not (not A))) -> not A
```

This also looks rather intimidating until we unpack the negation in a certain way, as convenient, and remove redundant brackets:

```
TNE : {A : Prop} -> (f : (not (not A)) -> Falsity) -> (a :
 A) -> Falsity
```

The idea of the proof is that we have a proof `a : A` and a rule `f : (not (not A)) -> Falsity`. How do we produce a proof of the `Falsity`?

Clearly we could apply `f`, if we manage to produce a proof of `not (not A)` somehow. But peeking above shows that we *can* produce that, out of `a : A`:

```
TNE : {A : Prop} -> (f : (not (not A)) -> Falsity) -> (a :
 A) -> Falsity
TNE {A} f a = x where
    y : not (not A)
    y = DNI a
    x : Falsity
    x = f y
```

Programming

The language we are using to describe proofs really is a programming language, and the rules really are functions we can compute with. But it is a programming language with a syntax deeply inspired by logic.

As a programming language (rather than a mere proof checker) the syntax is very flexible. All the definitions above can be *inlined*, just like in any language. The way we wrote the rules/functions so far was simply to emphasise the connection to natural deducation. But we could write things much more simply.

For exmaple, for `mp`:

```
mp : {A B : Prop} -> (f : A -> B) -> (a : A) -> B
mp {A} {B} f a = b where
  b : B
  b = f a
```

we can replace `b` with its definition `f a` so that no auxiliary clauses are needed:

```
mp : {A B : Prop} -> (f : A -> B) -> (a : A) -> B
mp {A} {B} f a = b where
  b : B
  b = f a
```

For `comp`:

```
comp : {A B C : Prop} -> (f : A -> B) -> (g : B -> C) -> (
a : A) -> C
comp f g = c where
   b : B
   b = mp f a
   c : C
   c = mp g c
```

we can replace both `c` and `b` with their respective definitions, resulting in

```
comp : {A B C : Prop} -> (f : A -> B) -> (g : B -> C) -> (
a : A) -> C
comp f g a = g (f a)
```

Similarly, for `DNI` and `TNE` we can inline everything, resulting in:

```
DNI : {A : Prop} : (a : A) -> (not (not A))
DNI a f = f a

TNE : {A : Prop} -> (f : (not (not A)) -> Falsity) -> (a :
 A) -> Falsity
TNE {A} f a = f (DNI a)
```

Note that because `DNI` is a *rule* we don't know how to replace it with its definition (yet).

We call these fully-inlined proofs with no auxiliary definitions **proof terms**.

Example

Let us conclude by looking at a more difficult proof, that of the *Law of Excluded Middle*:

```
LEM : {P : Prop} ->        P or (P -> Falsity)
LEM {P} = DNE goal where
  goal : (P or (P -> Falsity) -> Falsity) -> Falsity
  goal z = z u where
    v : P → Falsity
    v x = z w where
      w : P or (P -> Falsity)
      w = orIl x
    u : P or (P -> Falsity)
```

```
    u = orIr v
```

which can be simplified to

```
LEM : {P : Prop} ->         P or (P -> Falsity)
LEM {P} = DNE goal where
  goal : (P or (P -> Falsity) -> Falsity) -> Falsity
  goal z = z (orIr v) where
    v : P → Falsity
    v x = z (orIl x) where
```