Gmail **Walter Schulze <awalterschulze@gmail.com>**

Session6 - 2019-02-19: Summary

5 messages

Paul Cadman <pcadman@gmail.com> 20 February 2019 at 22:48
To: The Little Typer Study Group London <the-little-typer-study-group-london@googlegroups.com>

In the session on Tuesday we discussed solutions to the exercises for chapters 8 and 9:

https://github.com/paulcadman/the-little-typer/blob/master/exercises/equality-Nat-crib.rkt

Some of Ayman's solutions are in this gist:

https://gist.github.com/aymanosman/02a84d369eb83b402d61001fe7320d98

We also discussed how proof work in Coq and Idris.

Thanks all for coming,

Paul

Walter Schulze <awalterschulze@gmail.com> 25 February 2019 at 21:48
To: Paul Cadman <pcadman@gmail.com>
Cc: The Little Typer Study Group London <the-little-typer-study-group-london@googlegroups.com>

Since I was sick, I am still busy with Chapter 9, which I am finding quite tough.
I am having some trouble totally grasping replace, but I think I might finally be onto something.

I would like to think of replace like this:

```
(replace with-new
  in-here
  where-old)
```

where-old is some type, which contains the old expression I want to replace (.. old ..), this could be (same (add1 (add1 (+ n-1 j))), in other words (= Nat (add1 (add1 (+ n-1 j)) (add1 (add1 (+ n-1 j))) where old is (add1 (+ n-1 j))
in-here is a function that shows you where you want to replace old (lambda (here) (.. here ..)), this could be (= Nat (add1 (add1 (+ n-1 j)) (add1 here))
with-new is a proof that old is the same as new (= X old new), this could be (= Nat (add1 (+ n-1 j)) (+ n-1 (add1 j))), where old is (add1 (+ n-1 j)) and new is (+ n-1 (add1 j))
which would make the resulting type
(= Nat (add1 (add1 (+ n-1 j)) (add1 (+ n-1 (add1 j))))

So if I have a proof that old is the same as new
Then I can take any type containing old
Simply show replace, where old is
And get a type where old is replaced by new

What do you think?

mot, base and target seem to be confusing me at the moment.
[Quoted text hidden]

Paul Cadman <pcadman@gmail.com> 25 February 2019 at 22:13
To: The Little Typer Study Group London <the-little-typer-study-group-london@googlegroups.com>

That makes sense to me. Here's how I think of it (we'll all have our own way to understand it).

Let `target` be a proof that `from` is equal to `to` - i.e any expression with type (= X from to).

Let `mot` be a family of propositions (-> X U) - i.e for any x: X we get a proposition (lambda x). (NB: This proposition may or may not have proofs).

If `base` is a proof of the proposition (lambda from) then using the proof `target` we can **replace** the `from` in (lambda from) with `to` to get a proof of (lambda to).

[Quoted text hidden]

[Quoted text hidden]

To view this discussion on the web, visit https://groups.google.com/d/msgid/the-little-typer-study-group-london/02069818-64ef-4c06-aeb9-cb2d06ed07bd%40googlegroups.com.

[Quoted text hidden]

Ayman Osman <aymano.osman@gmail.com> 25 February 2019 at 23:39
To: Paul Cadman <pcadman@gmail.com>
Cc: The Little Typer Study Group London <the-little-typer-study-group-london@googlegroups.com>

It took me a while to understand what replace was doing so I think I'm in a good position to explain it. Here goes.

Consider this 'inference rule':

```
a=b  (... a ...)
----------------
  (... b ...)
```

In prose: given a proof that `a` equals `b` and a proof involving `a`, produce a proof involving `b`.

Think of the `(... a/b ...)` as a template. We can write a template as a lambda.
Namely, `(lambda (a/b) (... a/b ...)`.

A `replace` expression is then:

```
(replace a=b
  (lambda (a/b) (... a/b ...))
  (... a ...))
```

Which produces:

```
(... b ...)
```

Example: defining trans
==========
See https://docs.racket-lang.org/pie/index.html?q=trans#%28def._%28%28lib._pie%2Fmain..rkt%29._trans%29%29

```
#lang pie

(claim _trans
  (Pi ((X U)
       (from X)
       (middle X)
       (to X))
      (-> (= X from middle) (= X middle to)
        (= X from to))))

(define _trans
  (lambda (X from middle to from=middle middle=to)
    (replace middle=to
      (lambda (middle/to) (= X from middle/to))
      from=middle)))
```

(= X middle to) (= X from middle)

(= X from to)

So you can see why our template is of the form `(lambda (a/b) (= X from a/b))`.

Ayman

[Quoted text hidden]
[Quoted text hidden]
To view this discussion on the web, visit https://groups.google.com/d/msgid/the-little-typer-study-group-london/CALfQKLNeRc%2B_T3t1ZZUg4q%3D5MZv5jfUeF_Q_B_rrUfVyu6OSmQ%40mail.gmail.com.
[Quoted text hidden]

Walter Schulze <awalterschulze@gmail.com> 26 February 2019 at 15:58
To: Ayman Osman <aymano.osman@gmail.com>
Cc: Paul Cadman <pcadman@gmail.com>, The Little Typer Study Group London <the-little-typer-study-group-london@googlegroups.com>

Thank you both of you.

That was very helpful.

I particularly like that you both think differently.

a=b (... a ...)

(... b ...)

and

Let `mot` be a family of propositions (-> X U)

wow, these are both very helpful ways of thinking about it.

Thank you
[Quoted text hidden]