Linear functions and coverage checking stuff with holes in it
Request For Logic (RFL) #6
Robert J. Simmons
December 14, 2009

I describe the encoding of cut admissibility for rigid logic in order to motivate the problem of case analysis and coverage checking for contexts with holes in them - something that can be represented as linear functions. I describe several of the reasons this doesn’t work in Celf and Twelf, and also why it seems pretty cool despite not existing.

Rigid (i.e. non-commutative ordered) logic

Consider the non-associative Lambek calculus, i.e. non-associative ordered logic. We’ll call it Rigid Logic due to the rigid tree-like structure of contexts and because it gets really old referring to the non-associative Lambek calculus.

```
------ id
Q ⊢ Q

Δ ⊢ A    Γ[B] ⊢ C
------------------- ↠L
Γ[A ↠ B, Δ] ⊢ C

Γ, A ⊢ B
----------- ↠R
Γ ⊢ A ↠ B

Δ ⊢ A    Γ[B] ⊢ C
------------------- ↣L
Γ[Δ, A ↣ B] ⊢ C

A, Γ ⊢ B
----------- ↣R
Γ ⊢ A ↣ B

Γ[A, B] ⊢ C
---------------- •L
Γ[A • B] ⊢ C

Γ ⊢ A    Δ ⊢ B
---------------- •R
Γ , Δ ⊢ A • B
```

The rules for rigid logic look much like the ones for ordered logic, but the context is not associative as in ordered logic, so a , (a ↣ b, b ↣ c) ⊢ c is not provable but (a , a ↣ b) , b ↣ c ⊢ c is:

```
                 ------- init   -------- init
                  b ⊢ b          c ⊢ c
------ init      ------------------------ ↣ L
a ⊢ a             b, b ↣ c ⊢ c
---------------------------------- ↣L
 (a, a ↣ b), b ↣ c ⊢ c
```

Encoding rigid logic

We'll consider only the ↠ fragment of rigid logic, and both (non-adequate) Twelf encoding of the proof and (presumably adequate) Celf encoding.

Propositions

```
Twelf:
 prop : type.               %name prop A.
 atm : type.                %name atm Q q.
 a : atm -> prop.
 ↠ : prop -> prop -> prop. %infix right 9 ↠.
 %block vprop : block {q : atm}.

Celf (no infix or Unicode):
 prop : type.
 atm  : type.
 a    : atm -> prop.
 imp  : prop -> prop -> prop.
```

Contexts

```
Twelf:
ctx : type. %name ctx G γ.
hyp : prop -> ctx.
, : ctx -> ctx -> ctx.    %infix none 6 ,.

Celf:
 ctx  : type.
 hyp  : prop -> ctx.
 cons : ctx -> ctx -> ctx.
```

Rules

Here's where we're unable to keep a simple Twelf encoding adequate. The ↠L rule has a premise Γ[B] ⊢ C and a conclusion Γ[A ↠ B, Δ]. Γ[-] is usually described as a context with a single hole in it somewhere, which can be filled by any proposition. Therefore, it seems like it should be representable as a function from contexts to contexts:

```
Twelf:
 ⊢ : ctx -> prop -> type. %name ⊢ D. %infix none 3 ⊢.
 id : hyp (a Q) ⊢ a Q.
 ↠R : G , hyp A ⊢ B -> G ⊢ A ↠ B.
 ↠L : {G} GA ⊢ A -> G(hyp B) ⊢ C -> G(hyp(A ↠ B) , GA) ⊢ C.
```

However, this is not adequate, because the function G is a regular substitution function - the argument to a function can be present multiple times:

```
 ------ id          □
 a ⊢ a        b , b ⊢ b • b
 ----------------------------------- twelf-encoding-of-↠R (G = [γ] γ , γ)
 (a ↠ b , b) , (a ↠ b , b) ⊢ b • b
```

or not present at all:

```
 ------ id        ------- id
 a ⊢ a            c ⊢ c
 ----------------------------------- twelf-encoding-of-↠R (G = [γ] c)
 c ⊢ c
```

I've never seen any reference that indicates that either of these are legitimate instantiations of ↠R - in order to capture the intended meaning, G needs to be a linear function from contexts to contexts - a context with *exactly one* hole. Enter Celf (we're actually only using the LLF fragment), which gives what I believe to be an adequate encoding of the problematic ↠L rule.

```
Celf:
 seq  : ctx -> prop -> type.
 id   : seq (hyp (a Q)) (a Q).
 impR : seq (cons G (hyp A)) B -> seq G (imp A B).
 impL : Pi G: ctx -o ctx.
        seq GA A -> seq (G(hyp B)) C -> seq (G(cons (hyp(imp A B)) GA)) C.
```

Note that in both cases, in order for cut-elimination to typecheck at all we have to make the implicit argument G to ↠L/impL - the context with the hole in it - explicit.

Cut admissibility

Now we can consider Twelf and CLF proofs of cut elimination. We shouldn't really expect the Twelf proof to work - the failure of adequacy means are going to be Twelf proofs of "G ⊢ A" that don't correspond to any true sequent calculus proofs - but we should expect it to fail for interesting reasons. The Celf encoding fails for non-odd reasons. Celf has no meta-reasoning so I shouldn't expect it to check the proof, but I cannot get Celf to accept the computational content of the proof!

Twelf non-proof

Because our representation of derivations isn't adequate, we shouldn't expect this Twelf proof to work, but it is enlightening in its failures. We have to make the context-with-hole explicit as we did in ↠L, but beyond that, cut admissibility should look like this:

```
 cut : {A} GA ⊢ A -> {G} G(hyp A) ⊢ C -> G(GA) ⊢ C -> type.

 %% IDENTITY CUTS
 i1 : cut (a Q) id G E E.
 i2 : cut (a Q) D ([γ] γ) id D.

 %% LEFT COMMUTATIVE CUTS
 l1 : cut A (↠L GA₂ (D₁ : GA₁ ⊢ B₁) (D₂ : GA₂(hyp B₂) ⊢ A)) G E
      (↠L ([γ] G(GA₂ γ)) D₁ F₂)
      <- cut A D₂ G E F₂.

 %% RIGHT COMMUTATIVE CUTS
 r1 : cut A D G (↠R E)
      (↠R F)
      <- cut A D ([γ] G(γ) , hyp C₁) E F.
 r2 : cut A D ([γ] G (hyp(B₁ ↠ B₂) , G₁(γ))) (↠L G E₁ E₂)
      (↠L G F₁ E₂)
      <- cut A D ([γ] G₁(γ)) E₁ F₁.
 r3 : cut A D ([γ] G(γ)(hyp(B₁ ↠ B₂) , G₁)) (↠L (G(hyp A)) E₁ E₂)
      (↠L (G(GA)) E₁ F₂)
      <- cut A D ([γ] G(γ)(hyp B₂)) E₂ F₂.

 %% PRINCIPAL CUTS
 p1 : cut (A₁ ↠ A₂) (↠R D) ([γ] G(γ , G₁)) (↠L G E₁ E₂) F
      <- cut A₁ E₁ ([γ] GA , γ) D F₁
      <- cut A₂ F₁ G E₂ F.
```

```
%mode cut +A +D +G +E -F.
%worlds (vprop) (cut _ _ _ _ _).
%total {A [D E]} (cut A D G E F).
```

Failure 1: mode checking

So, the first failure is that rule r2 (and r3) don't mode check! We'll look at r2 first:

Occurrence of variable G_1 in input (+) argument not necessarily ground

This makes sense in light of the non-adequate encoding. In this case, we're thinking about the second derivation ↠L looking like this:

$$\frac{E_1 : \Gamma_1[A] \vdash B_1 \qquad E_2 : \Gamma[B_2] \vdash C}{\Gamma[B_1 \twoheadrightarrow B_2,\ \Gamma_1[A]] \vdash C} \twoheadrightarrow L$$

And then making a recursive call using the derivation $\Gamma_1[A] \vdash B$. But Twelf calls foul: $\Gamma[-]$ is encoded as a function $([\gamma]\ G\ (hyp(B_1 \twoheadrightarrow B_2)\ ,\ G_1(\gamma)))$. But what if G doesn't use it's argument, that is, what if $([\gamma]\ G\ \gamma = [\gamma]\ G')$ for some non-function G'? This would be impossible if G was a linear function, but it's possible here. In that case, then B_1, B_2, and G_1 are completely unconstrained, so we can't expect G_1 to be fully constrained when we use it in the recursive call!

Rule r3 gives the same error message, but for G, not G'. In that case, I have a premise $([\gamma]\ G(\gamma)(hyp(B_1 \twoheadrightarrow B_2)\ ,\ G_1))$ and I match it against an incoming input. What if it's this input?

$([\gamma]\ (hyp(A \twoheadrightarrow B)\ ,\ hyp\ C)\ ,\ (hyp(A \twoheadrightarrow B)\ ,\ hyp\ C))$

There are many possibilities - two of them are actually linear in δ!

$B_1 = A,\ B_2 = B,\ G_1 = hyp\ C,\ G = [\gamma][\delta]\ \delta\ ,\ \delta$
$B_1 = A,\ B_2 = B,\ G_1 = hyp\ C,\ G = [\gamma][\delta]\ (hyp(A \twoheadrightarrow B)\ ,\ hyp\ C)\ ,\ \delta$
$B_1 = A,\ B_2 = B,\ G_1 = hyp\ C,\ G = [\gamma][\delta]\ \delta\ ,\ (hyp(A \twoheadrightarrow B)\ ,\ hyp\ C)$
$B_1 = ?,\ B_2 = ?,\ G_1 = ?,\ G = [\gamma][\delta]\ (hyp(A \twoheadrightarrow B)\ ,\ hyp\ C)\ ,\ (hyp(A \twoheadrightarrow B)\ ,\ hyp\ C)$

Of course the fourth possibility is a problem, but even before then, Twelf does not deal with with the possibility of there being multiple successful ways of instantiating something. I hope that this is the same observation being made in section 5.2.3 of [JR].

Failure 2: coverage checking

Coverage checking also fails, but in each case this can be chalked up to a failure of the.

Celf proof

The computational content of cut should be adequately representable in Celf:

```
cut  : Pi A: prop. seq GA A -> Pi G: ctx -o ctx. seq (G(hyp A)) C
        -> seq (G(GA)) C -> type.

%% IDENTITY CUTS
i1 : cut (a Q) id (\g. G(g)) E E.
i1 : cut (a Q) D (\g. g) id D.

%% LEFT COMMUTATIVE CUTS
l1 : cut A (impL GA2 D1 D2) (\g. G(g)) E
      (impL (\g. G(GA2(g))) D1 F2)
      <- cut A D2 (\g. G(g)) E F2.
```

```
%% RIGHT COMMUTATIVE CUTS
r1 : cut A D (\g. G g)
      (impR E)
      (impR F)
      <- cut A D (\g. cons (G g) (hyp C1)) E F.
r2 : cut A D (\g. G(cons (hyp(imp B1 B2)) (G1(g))))
      (impL (\g. G(g)) E1 E2)
      (impL (\g. G(g)) F1 E2)
      <- cut A D (\g. G1(g)) E1 F1.
r3 : cut A D (\g. G(g)(cons (hyp(imp B1 B2)) G1))
      (impL (\g. G(hyp A)(g)) E1 E2)
      (impL (\g. G(GA)(g)) E1 F2)
      <- cut A D (\g. G(g)(hyp B2)) E2 F2.

%% PRINCIPAL CUTS
p1 : cut (imp A1 A2) (impR D) (\g. G(cons g G1)) (impL G E1 E2) F
      <- cut A1 E1 (\g. cons GA g) D F1
      <- cut A2 F1 G E2 F.
```

Failure 3: type reconstruction

Even after making the context argument explicit, Celf does not accept r1 or r2 in the above signature.

Functional programming with linear stuff

It seems like this would be a fun programming language, though - able to describe logic-programmey things like difference lists:

```
queue : list -o list
isEmpty : queue -> bool
isEmpty λ̂x.x = true
isEmpty _ = false

push : int -> queue -> queue
push N λ̂x.Q[x] = λ̂x.Q[N :: x]

pop : queue -> int * queue
pop λ̂x.N :: Q[x] = (N, λ̂x.Q[x])
pop λ̂x.x = raise EmptyQueue

append : queue -> queue -> queue
append λ̂x.Q₁[x] λ̂y.Q₂[y] = λ̂z.Q₁[Q₂[z]]

to_list : queue -> list
to_list λ̂x.Q₁[x] = Q₁[nil]

from_list : list -> queue
from_list Q₁[nil] = λ̂x.Q₁[x]
```

Quoth Dan Licata: so what is the relationship between linear functions and derivatives [CM]? I have no idea.

```
list(bool * tree)      =   tree ⊸ tree
[]*                    =   λ̂x.x
((true , r) :: S)*     =   λ̂x.node(S* x, r)  (or λ̂x.S*(node(x, r))
((false , l) :: S)*    =   λ̂x.node(l, S* x)  (or λ̂x.S*(node(l, x))
```

References

[CM] C. McBride, “The derivative of a regular type is its type of one-hole contexts,” 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8611

[JR] J. Reed, “A hybrid logical framework,” Ph.D. dissertation, Carnegie Mellon University, July 2009. [Online]. Available: http://reports-archive.adm.cs.cmu.edu/anon/2009/abstracts/09-155.html