Problem Set 2 and iFOL Rules

September 29, 2018

1 Propositions

Please write simple realizers that provide the evidence for the following propositions. It might be that one or two of them are not constructively true and thus have no realizers. In any such case, explain why there is no realizer (program) for the task.

1. $(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$
2. $(P \Rightarrow (Q \Rightarrow (P \wedge Q)))$
3. $(P \wedge Q) \Rightarrow (P \vee Q)$
4. $(P \vee Q) \Rightarrow (Q \vee P)$
5. $P \Rightarrow \neg(\neg P)$
6. $\neg(P \wedge Q) \Rightarrow (\neg P \vee \neg Q)$
7. $\neg(P \vee Q) \Rightarrow \neg P \wedge \neg Q$
8. $(\neg P \vee \neg Q) \Rightarrow \neg(P \wedge Q)$

2 Proof Rules and Proof Expressions

Each rule has a name that is the outer operator of the proof expression with slots to be filled in as the proof is developed. The partial proofs are organized as a tree generated in two passes. The first pass is *top down*, driven by the user creating terms with slots to be filled in on an algorithmic bottom up pass.

$\vdash A \Rightarrow (B \Rightarrow A)$ *by* $\lambda(x.slot_1(x))$
$x : A \vdash (B \Rightarrow A)$ *by* $slot_1(x)$

In the next step, $slot_1(x)$ is replaced at the leaf of the tree by $\lambda(y.slot_2(x, y))$ to give:

$\vdash A \Rightarrow (B \Rightarrow A)$ *by* $\lambda(x.slot_1(x))$
$x : A \vdash (B \Rightarrow A)$ *by* $\lambda(y.slot_2(x, y))$ *for* $slot_1(x)$
$x : A, y : B \vdash A$ *by* $slot_2(x, y)$

When the proof is complete, we see the slots filled in at each inference step as in:

$\vdash A \Rightarrow (B \Rightarrow A)$ *by* $\lambda(x.\lambda(y.x))$
$x : A \vdash (B \Rightarrow A)$ *by* $\lambda(y.x)$

$$x : A, y : B \vdash A \ by \ x$$

The variable x fills $slot_2$ and the lambda term $\lambda(y.x)$ fills $slot_1$. The complete proof expression is thus the lambda term $\lambda(x.\lambda(y.x))$. It is easy to see intuitively that the meaning of this term is precisely the evidence needed to show that the formula is true. If we used *typed lambda terms*, the proof term would be $\lambda(a : A.(\lambda(y : B.x)))$. (We could make the proof expression more standard if we used a name such as *impin* instead of λ; we prefer to use notation that makes the semantic ideas clearer than the traditional rule names.

The rules are organized with the *construction rules* first, often called the *introduction rules* because they introduce the *canonical proof terms* also called the *right hand side* rules because they apply to terms on the right hand side of the *turnstile*, $\vdash$ .. Typical names seen in the literature are: for & say *AndIntro* or *AndR*; for $\Rightarrow$ say *ImpIntro* or *ImpR*; for $\vee$ say *OrIn-r* or *OrIn-l*, and for $\forall x$ say *AllIntro* or *AllR*, and for $\exists$ say *ExistsIntro* or *ExistsR*.

The decomposition rules for $A \Rightarrow B$ and $\forall x.B(x)$ are the most delicate to motivate and use intuitively. The evidence for $A \Rightarrow B$ is a function $\lambda(x.b(x))$, a reader might expect to see a decomposition rule name such as $apply(f; a)$ or abbreviated to $ap(f; a)$. However, a Gentzen sequent-style proof rule for decomposing an implication has this form:

$H, f : A \Rightarrow B, H' \ \vdash \ G \ by \ ImpL \ \ on \ f$
1. $H, f : A \Rightarrow B, H' \ \vdash \ A$
2. $H, f : A \Rightarrow B, v : B, H' \ \vdash \ G$

As the proof proceeds, the two subgoals 1 and 2 with conclusions A and G respectively will be refined, say with proof terms $g(f, v)$ and a respectively. The rules need to indicate that the value v is $ap(f; a)$, but at the point where the rule is applied, there are only slots for these subterms and a name v for the new hypothesis B. So the rule form is $apseq(f; slot_a; v.slot_g(v))$ where we know that v will be assigned the value $ap(f; slot_a)$ to "sequence" the two subgoals properly. So *apseq* is a sequencing operator as well as an application. When the subterms are created the proof term can be further evaluated as shown below.

$H, f : A \Rightarrow B, H' \ \vdash \ G \ by \ apseq(f; slot_a; v.slot_g(v))$
$H, f : A \Rightarrow B, v : B, H' \ \vdash \ G \ by \ slot_g(v)$
$H, f : A \Rightarrow B, H' \ \vdash \ A \ by \ slot_a$

We can evaluate the term $apseq(f; a; v.g(v))$ to $g(ap(f; a))$ or more succinctly to $g(f(a))$. This simplification can only be done on the final bottom up pass of creating a closed proof expression, one with no slots.

2.1 First-order refinement style proof rules over domain of discourse D

Minimal Logic

Construction rules

- **And Construction**
  $H \ \vdash \ A\&B \ by \ pair(slot_a; slot_b)$
  $H \ \vdash A \ by \ slot_a$
  $H \ \vdash B \ by \ slot_b$

- **Exists Construction**[a]
  $H \ \vdash \ \exists x.B(x) \ by \ pair(d; slot_b(d))$
  $H \ \vdash d \in D \ by \ obj(d)$
  $H \ \vdash B(d) \ by \ slot_b(d)$

[a] Also see Alternative Rules below.

- **Implication Construction**
  $H \ \vdash \ A \Rightarrow B$ *by* $\lambda(x.slot_b(x))$ *new* x
  $H,\ x : A \vdash B$ *by* $slot_b(x)$

- **All Construction**
  $H \ \vdash \ \forall x.B(x)$ *by* $\lambda(x.slot_b(x))$ *new* x
  $H,\ x : D \vdash B(x)$ *by* $slot_b(x)$

- **Or Construction - left**
  $H \ \vdash \ A \vee B$ *by* $inl(slot_l)$
  $H \ \vdash A$ *by* $slot_l$

- **Or Construction - right**
  $H \ \vdash \ A \vee B$ *by* $inr(slot_r)$
  $H \ \vdash B$ *by* $slot_r$

Decomposition rules

- **And Decomposition**

  $H, x : A\&B, H' \vdash G$ *by* $spread(x; l, r.slot_g(l, r))$ *new* l, r
  $H, l : A, r : B, H' \vdash G$ *by* $slot_g(l, r)$

- **Exists Decomposition**

  $H, x : \exists y.B(y), H' \vdash G$ *by* $spread(x; d, r.slot_g(d, r))$ *new* d, r
  $H, d : D, r : B(d), H' \vdash G$ *by* $slot_g(d, r)$

- **Implication Decomposition**

  $H, f : A \Rightarrow B, H' \vdash G$ *by* $apseq(f; slot_a; v.slot_g[ap(f; slot_a)/v])$ *new* v[1]
  $H, f : A \Rightarrow B, H' \vdash A$ *by* $slot_a$
  $H, f : A \Rightarrow B, H', v : B \vdash G$ *by* $slot_g(v)$

- **All Decomposition**

  $H, f : \forall x.B(x), H' \vdash G$ *by* $apseq(f; d; v.slot_g[ap(f; d)/v])$
  $H, f : \forall x.B(x), H' \vdash d \in D$ *by* $obj(d)$
  $H, f : \forall x.B(x), H', v : B(d) \vdash G$ *by* $slot_g(v)$[2]

- **Or Decomposition**

  $H, y : A \vee B, H' \vdash G$ *by* $decide(y; l.leftslot(l); r.rightslot(r))$
  1. $H, l : A, H' \vdash G$ *by* $leftslot(l)$
  2. $H, r : B, H' \vdash G$ *by* $rightslot(r)$

- **Hypothesis - domain (D)**
  $H, d : D, H' \vdash \ d \in D$ *by* $obj(d)$

- **Hypothesis - formula (A)**
  $H, x : A, H' \vdash \ A$ *by* $hyp(x)$
  We usually abbreviate the justifications to *by* d and *by* x respectively.

Intuitionistic Rules

- **False Decomposition**

  $H, f : False, H' \vdash G$ *by* $any(f)$

  This is the rule that distinguishes intuitionistic from minimal logic, called "ex falso quodlibet". We use the constant $False$ for intuitionistic formulas and $\bot$ for minimal ones to distinguish the logics. In practice, we would use only one constant, say $\bot$, and simply add the above rule with $\bot$ for $False$ to axiomatize iFOL. However, for our results it is especially important to be clear about the difference, so we use both notations.

[1] This notation shows that $ap(f; slot_a)$ is substituted for v in $g(v)$. In the CTT logic we stipulate in the rule that $v = ap(f; slot_a)$ *in* B.

[2] In the CTT logic, we use equality to stipulate that $v = ap(f; d)$ *in* $B(v)$ just before the hypothesis $v : B(d)$.

Note that we use the term d to denote objects in the domain of discourse D. In the classical evidence semantics, we assume that D is non-empty by postulating the existence of some d_0 in it. Also note that in the rule for $False$ Decomposition, it is important to use the $any(f)$ term which allows us to thread the explanation for how $False$ was derived into the justification for G.

Structural Rules

- **Cut rule**

  $H \vdash G$ *by* $CutC(x.slot_g(slot_c))$ *new* x
  1. $H, x : C \vdash G$ *by* $slot_g(x)$
  2. $H \vdash C$ *by* $slot_c$.

Classical Rules

- **Non-empty Domain of Discourse**

  $H \vdash d_0 \in D$ *by* $obj(d_0)$

- **Law of Excluded Middle (LEM)** Define $\sim A$ as $(A \Rightarrow False)$
  $H \vdash (A \vee \sim A)$ *by* **magic**(A)
  Note that this is the only rule that mentions a formula in the rule name.

Alterative Rule

- **Exists Construction**

  $H \vdash \exists x.B(x)$ *by* $pair(slot_d/X; slot_b[slot_d/X])$
  $H \vdash D$ *by* $slot_d$
  $H \vdash B(X)$ *by* $slot_b(X)$
  Note, the substitution of $slot_d$ propagates to $B(X)$ as soon as the first subgoal determines the value of the slot for the goal rule. The term X acts as a *logic variable*.

2.2 Computation rules

Each of the rule forms when completely filled in becomes a term in an applied lambda calculus. The standard *computation rules* specify how to reduce these terms. These rules are given in detail in papers about the lambda calculus. They are not repeated here. The reduction is to *head normal form*.

The reduction rules are simple. For $ap(f; a)$, first reduce f, if it becomes a function term, $\lambda(x.b)$, then reduce the function term to $b[a/x]$, that is, substitute the argument a for the variable x in the body of the function b and continue computing. If it does not reduce to a function, then no further reductions are possible and the computation aborts. It is possible that such a reduction will abort or continue indefinitely. But the terms arising from proofs will always reduce to normal form. This fact is discussed in the references.

To reduce $spread(p; x, y.g)$, reduce the principal argument p. If it does not reduce to $pair(a; b)$, then there are no further reductions, otherwise, reduce $g[a/x, b/y]$.

To reduce $decide(d; l.left; r.right)$, reduce the principal argument d until it becomes either $inl(a)$ or $inr(b)$ or aborts or fails to terminate.[3]. In the first case, continue by reducing $left[a/l]$ and in the other case, continue by reducing $right[b/r]$.

[3]The computation systems of CTT and ITT include diverging terms such as $fix(\lambda(x.x))$. We sometimes let $\uparrow$ denote such terms.

It is thus easy to see that all evidence terms terminate on all inputs from all models. This is a fact about valid evidence structures.

Additional notations It is useful to generalize the semantic operators to n-ary versions. For example, we will write λ terms of the form $\lambda(x_1, ..., x_n.b)$ and a corresponding n-ary application, $f(x_1, ..., x_n)$. We allow n-ary conjunctions and n-tuples which we decompose using
$spread_n(p; x_1, ..., x_n.b)$. More rarely we use n-ary disjunction and the decider,
$decide_n(d; case_1.b_1; ...; case_n.b_n)$. It is clear how to extend the computation rules and how to define these operators in terms of the primitive ones.

It is also useful to define $True$ to be the type $\bot \Rightarrow \bot$ with element $id = \lambda(x.x)$. Note that $\lambda(x.spread(pair(x; x); x_1, x_2.x_1))$ is computationally equivalent to id, as is
$\lambda(x.decide(inr(x); l.x; r.x))$[4].

[4]We could also use the term $\lambda(x.decide(inr(x); l.div; r.r))$ and normalization would reduce it to $\lambda(x.x)$.