Applied Logic - CS4860-2018-Lecture 7

Robert L. Constable

Abstract

We continue our study of *evidence semantics* covered in Lectures 5 and 6. We use the insights from the last lecture to motivate the rules for propositional logic that are *based on evidence* rather than on truth values [1].

We also look at the task of showing whether a propositional formula has evidence. This is a subtle issue, and a great deal of work has been done to make it clear. Some of the deepest insights are from Dr.Judith Underwood who received her Cornell CS PhD in this area [4]. The contrast with truth value semantics for propositional logic is stark. Smullyan's book makes the truth functional case completely clear and natural. We will soon build on his insights as we read part II of the course textbook, *First-Order Logic* and extend our knowledge of truth functional semantics.

A major advantage of truth functional semantics for propositional logic is that we use it to decide whether a propositional formula is valid. There are conceptually simple (but computationally expensive) algorithms to do this. There are also methods to decide whether an intuitionistic propositional formula has computational evidence. They are more expensive, requiring polynomial space. Smullyan's PhD student M.C. Fitting wrote a book extending the Tableaux style logic to *intuitionistic validity* [2]. We will mention these results from time to time, but we will not prove them in this course.

An interesting feature of evidence semantics is that if our decision procedure shows that a logical specification has no evidence, then we know it is *not programmable*, i.e. we cannot not find evidence, including writing a program, to show that the proposition is "constructively true."

1 Review of Evidence Semantics and Further Observations

Evidence semantics for intuitionistic propositional logic requires understanding the type of *ordered pairs*, $pair(a; b)$, sometime also written $< a, b >$. This is a basic concept that we assume all students understand. We use operators to decompose these pairs. The most general

one is $spread(p; a, b.exp(a, b))$. From this we can define operations such as $first(pair(a; b))$ as $spread(p; a, b.a)$ which reduces to a and $second(pair(a; b))$ as $spread(p; a, b.b)$ which reduces to b. The term $pair(a; b)$ is the *canonical form* for conjunction.

A somewhat less familiar logical operator is the *disjoint union* $A \vee B$. This corresponds to a constructive logical *disjunction*, as in "A or B." The canonical elements of this type are either $inl(a)$ where a is evidence for A or $inr(b)$ where b is evidence for B. The tags inl and inr specify which proposition the evidence supports. The operation for processing this evidence is $decide(d; l.left(l); r.right(r))$.

For example $decide(inl(a); l.a + 1; inr(b).if\ b = true\ then\ false\ else\ true)$. This expression makes sense, "type checks", only if the a expression in $inl(a)$ is a number, say $inl(7)$, and b in the $inr(b)$ term is a Boolean, say $inr(true)$. We know this because the addition operator in $a + 1$ requires a number, and the conditional test on b requires a Boolean value, either $true$ or $false$.

One of the most expressive logical operators is *implication* as in $A \Rightarrow B$. The formula $A \Rightarrow B$ is a constructive *implication*. It captures a computational understanding of implication and is quite intuitive. To know $A \Rightarrow B$ we need a computational method to transform evidence a for A into evidence for B. We are familiar with using functions for such tasks, they transform inputs into outputs. So the evidence for $A \Rightarrow B$ is a *computable function*. We need this function to be computable because we require concrete access to the evidence for B if we are given concrete evidence for A.

There are many notations for functions in mathematics and computer science. In mathematics it is not always necessary to provide a method of computation. In computer science it is natural and often necessary to provide computation rules for functions. There are several notations for functions depending on the programming language. In functional programming languages, it is common to follow Church and use some variant of the *lambda calculus*. One variant of Church's notation is $\lambda(x.b(x))$ where the variable x is called the *bound variable*, and $b(x)$ is the *body* of the function. We teach a lot about this notion of computable function in functional programming courses, where we currently use OCaml. In the OCaml programming language, functions are written as $fun\ x \rightarrow\ b(x)$. The bound variable x names the input, and the body $b(x)$ shows how to compute the output value. Here is a concrete example where the input is an ordered pair, $\lambda(x.first(x))$.

Associated with the above way of naming functions is the following notion of application, $ap(f; a)$. The *computation rule* (or reduction rule) is $ap(\lambda(x; first(x)); pair(a; b))$ reduces to a and $ap(\lambda(x; second(x)); pair(a; b))$ reduces to b.

2 Proof Rules Based on Propositional Evidence

For each proof rule we provide a name that is the *outermost operator* of a proof expression with slots to be filled in as the refinement style 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 the algorithmic bottom up pass once the downward pass is complete. Here is a simple proof of the intuitionistic tautology $A \Rightarrow (B \Rightarrow A)$.

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

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

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 decomposition rule for $A \Rightarrow B$ is more delicate.

$H, f : A \Rightarrow B, H' \ \vdash \ G \ by \ ImpL \ \ on \ f$
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. We need to indicate that the value v is $ap(f; a)$, but at the point where the rule is applied, we only have 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, and when the subterms are created, we can evaluate the term further as we show below. We thus express the rule as follows.

$H, f : A \Rightarrow B, H' \vdash G$ *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 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.

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* $slot_b$

- **Implication Construction**

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

- **Or Construction**

  $H \vdash A \vee B$ *by* $inl(slot_l)$
  $H \vdash A$ *by* $slot_l$

  $H \vdash A \vee B$ *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* $slot_g(l, r)$

- **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* $slot_g(v)$

- **Simple Implication Decomposition**

  $H, f : A \Rightarrow B, H' \vdash G$ *by* $ap(f; slot_a)$

  $H, f : A \Rightarrow B, H' \vdash A$ *by* $slot_a$

  $H, f : A \Rightarrow B, H', v : B \vdash G$ *by* $slot_g(v)$

  The value a provided by filling in $slot_a$ is used by the function f to compute the value v of type B used to compute the value $g(v)$ that proves the goal. So the goal is implemented by $g(f(a))$.

- **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**

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

[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.

$H, x : A, H' \vdash\ A\ by\ hyp(x)$
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's especially important to be clear about the difference, so we use both notations.

Structural Rules

- **Cut rule**

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

Classical Rule

- **Law of Excluded Middle (LEM)**

  Define $\sim A$ as $(A \Rightarrow False)$

  $H \vdash (A \vee \sim A)\ by\ \mathbf{magic}(A)$

  Note that this is the only rule that mentions a formula in the rule name.

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, $decide_n(d; case_1.b_1; ...; case$
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 [3], as is

$\lambda(x.decide(inr(x); l.x; r.x))$.[2].

References

[1] Robert L. Constable. The semantics of evidence (also appeared as Assigning Meaning to Proofs). *Constructive Methods of Computing Science*, F55:63–91, 1989.

[2] M. Fitting. *Intuitionistic model theory and forcing.* North-Holland, Amsterdam, 1969.

[3] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[4] Judith L. Underwood. Aspects of the computational content of proofs. Department of Computer Science TR94-1460, Cornell University, Ithaca, NY, October 1994.

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