Applied Logic	Lecture 3: Evidence Semantics for (Intuitionistic) Propositional Logic
CS 4860 Fall 2012	Thursday, August 30, 2012

So far we have used propositional logic only as vehicle for writing down statements in a more formal fashion. If we want to use it to analyze the validity of a given argument or to prove a mathematical theorem, we have to be more accurate about the precise meaning or *semantics* of logical formulas.

Most logic textbooks introduce truth tables or valuations and set-theoretic interpretations for this purpose. But such an approach links logical formulas to the metaphysical concept of truth and would, if done properly, require an explanation what truth is (textbooks usually don't do that). We will avoid dealing with such philosophical issues and explain the meaning of formulas by describing the evidence that can be given for them. This may be a bit unusual for mathematicians but we look at logic from the perspective of computer science. We want to be able to *construct* justifications for the validity of a claim and use justifications in a computational fashion when analyzing complex logical formulas.

Since we're starting from scratch the laws of Boolean algebra that most of you are probably familiar with cannot be used until they have been justified by evidence. We can only use what we have established. This is the same as in programming languages. At the beginning you can only use the features built into the language itself. If you want to use more complex operations than that, you first have to program them and add them to a program library.

Before defining the language for expressing evidence we investigate what we need and how this can be formalized. Much of our discussion will involve placeholders for logical formulas again. If A is a proposition we often use the corresponding small letter a for a specific evidence for A and write $[A]$ for the type of evidences for A. This allows us to abbreviate the phrase "a *is evidence for* A" by "$a \in [A]$" (mathematical notation) or "$a : [A]$" in programming language notation.

- Evidence for propositional variables

  A propositional variable A is a placeholder for some arbitrary unknown proposition. As such, they cannot have a fixed evidence and the type of its evidences $[A]$ remains unspecified.

- Evidence for $A \Rightarrow B$

  To know $A \Rightarrow B$ we must know how to generate some evidence b for B if we have evidence a for A. Thus the evidence for $A \Rightarrow B$ must be a *function* f that on input $a : [A]$ generates some $b : [B]$ and the type of evidences $[A \Rightarrow B]$ is the type of functions from $[A]$ to $[B]$.

  We write $[A \Rightarrow B] = [A] \to [B]$

  Let's look at two examples that we briefly discussed in the previous lecture

  - $P \Rightarrow P$: The evidence must be a function $f : [P] \to [P]$ that takes some evidence $p : [P]$ and produces evidence for P. The simplest function that will do that is the identity function, thus we can choose $f(p) = p$.

    How do we describe such a function as a closed expression, that is without using the symbol f? That symbol was only introduced as a reference to the evidence that we wanted to describe. We could have equally well called this function g or h, so the

chosen name should not occur in the evidence term. We just need a way to describe the function's behavior and that is taking and input p and returning p as result.

So the evidence is a mapping $p \mapsto p$ and in that mapping, even the name p is just an abstract placeholder for the input – or the input variable, if we were to use programming terminology. To mark p as such, mathematicians in the 1930's have used a variety of symbols, among others a backslash $\backslash$, and used the notation $\backslash p.\, p$ to describe the mapping. Later the backslash was replaced by the greek letter λ (*lambda*), which looks similar but was easier to pronounce, and the mapping was written as $\lambda p.\, p$. This notation has later become known as *abstraction* of the *lambda-calculus*, which has become the basis for many functional languages.

- $P \Rightarrow (Q \Rightarrow P)$: The evidence must be a function that takes some evidence $p : [P]$ and produces a function $g_p : [Q] \rightarrow [P]$, which takes some evidence $q : [Q]$ and produces evidence for P. The simplest solution for this specification of g_p is $\lambda q.\, p$. Thus the overall evidence will have the form $\lambda p.\, (\lambda q.\, p)$.

Here is a slightly more difficult example

- $P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)$: The evidence must be a function that takes some evidence $p : [P]$ and produces a function g_p, which takes as input a function $h : [P] \rightarrow [Q]$ and produces evidence for Q, that is an element of $[Q]$.

  Now since h produces such an element of $[Q]$ if given an element of $[P]$, we only need to apply h to the element $p : [P]$ that we already have. We write this in the usual fashion as $h(p)$, the *application* of h to p. g_p would thus be $\lambda h.\, h(p)$ and the overall evidence is $\lambda p.\, (\lambda h.\, h(p))$.[1]

- Evidence for $A \wedge B$

  To know $A \wedge B$ we must know A and we must know B. That is evidence for $A \wedge B$ must consist of both evidence a for A and evidence b for B, i.e. of a *pair* (a, b) where $a : [A]$ and $b : [B]$. So the type of evidences $[A \wedge B]$ is the *product type* of $[A]$ and $[B]$.

  We write $[A \wedge B] = [A] \times [B]$

  Again let us look at a few concrete examples

  - $P \Rightarrow (Q \Rightarrow (P \wedge Q))$: The evidence must be a function that takes some evidence $p : [P]$ and produces a function $g_p : [Q] \rightarrow ([P] \times [Q])$, which takes some evidence $q : [Q]$ and produces an element of $[P] \times [Q]$. The simplest solution for this specification of g_p is $\lambda q.\, (p, q)$ und the overall evidence is $\lambda p. (\lambda q.\, (p, q))$.
  - $P \Rightarrow (Q \Rightarrow (Q \wedge P))$: Using the same argument we get the evidence $\lambda p. (\lambda q.\, (q, p))$.
  - $(P \wedge Q) \Rightarrow P$: The evidence must be a function that takes an element x of $[P] \times [Q]$ and produces an element of $[P]$. Now x must be a pair whose *first component* x_1 is an element of $[P]$ and whose *second component* x_2 is an element of $[Q]$. Thus the evidence we need is $\lambda x.\, x_1$.

[1] Some people may get confused by the fact that we permit *higher-order functions*, i.e. functions that take other functions as input. Some of the older programming languages like C don't support this concept directly, but it is actually quite natural. We use it in calculus when we integrate or differentiate functions and many modern programming languages support it now, because it is very convenient to have. Higher-order constructs only cause complications when it comes to theoretical investigations or building fully automated proof engines.

- Evidence for $A \vee B$

  To know $A \vee B$ we must know A or we must know B. That is evidence for $A \vee B$ must consist either of evidence a for A or of evidence b for B. That alone, however wouldn't be sufficient. why?

  Imagine A is $P \Rightarrow P$ and B is $P \Rightarrow Q$. Then providing the evidence $\lambda p.\, p$ would indicate that we know $(P \Rightarrow P) \vee P \Rightarrow Q)$ but not which of the two we know to be true.

  Thus in addition to providing the evidence for one of the disjuncts we must denote whether the evidence was given for the left or for the right one. There are many notations that we can use for this purpose. We follow the standard programming notation and write $\texttt{inl}(a)$ where $a : [A]$ to show that we give evidence for the left disjunct and $\texttt{inr}(b)$ where $b : [B]$ to show that we give evidence for right disjunct. So the type of evidences $[A \vee B]$ is the *(direct) sum type* (or *disjoint union*) of $[A]$ and $[B]$.

  We write $[A \vee B] = [A] + [B]$

  Again let us look at concrete examples

  - $P \Rightarrow (P \vee Q)$: We need a function that takes some evidence $p : [P]$ and produces an element of $[P] + [Q]$. The simplest solution for this specification is $\lambda p.\, \texttt{inl}(p)$.
  - $(P \vee Q) \Rightarrow (Q \vee P)$: We need a function that takes some element x of $[P] + [Q]$ and produces an element of $[Q] + [P]$. As an element of the direct sum $[P] + [Q]$ x is either $\texttt{inl}(p)$ for some $p : [P]$ or $\texttt{inr}(q)$ for some $q : [Q]$. In the first case the solution is $\texttt{inr}(p)$ and in the second it is $\texttt{inl}(q)$.

    A closed expression for such a case analysis is a slightly more complex version of a boolean conditional. We have to decide if x belongs to the left part or to the right part of the disjoint union $[P] + [Q]$ and we have to forward the term that is inside the `inl` or the `inr` expression. We use a simplified form of pattern matching for this purpose and write $\texttt{case}\ x\ \texttt{of}\ \texttt{inl}(a) \rightarrow s \mid \texttt{inr}(b) \rightarrow t$ to denote the following behavior: if an element e of a direct sum has the form $\texttt{inl}(a)$ then we return the expression s (which may contain a), and if it has the form $\texttt{inr}(b)$ then we return t (which may contain b).

    Thus the evidence is $\lambda x.\, (\texttt{case}\ x\ \texttt{of}\ \texttt{inl}(p) \rightarrow \texttt{inr}(p) \mid \texttt{inr}(q) \rightarrow \texttt{inl}(q))$.

- Evidence for $\neg A$

  Negation is one of the trickiest issues in logic. What does it mean to know the negation of a proposition A? We could simply say that it means that A *is not true* but then we still do not know what that is supposed to mean. Is it sufficient to say that there is no evidence for A? That would mean that we have to show that $[A]$ is empty and for that we would have to start using informal arguments again.

  If we want to avoid informal reasoning we have to move the argument that A cannot have evidence into the logic itself and show that assuming A leads to a contradiction. Thus the evidence for $\neg A$ would be a function that on input $a : [A]$ gives us evidence for some contradictory proposition, e.g. evidence for `0=1`.

  Since basic propositional logic doesn't include arithmetic, we need to add a fundamental proposition to the language of logic that does not have evidence. To avoid any confusion

with truth values we denote this proposition by f and postulate that the type of its evidences is the empty type: $[\mathsf{f}] = \{\}$[2].

Using this proposition, we can view $\neg A$ as abbreviation for $A \Rightarrow \mathsf{f}$, which means that evidences for $\neg A$ are functions from $[A]$ into the empty type. Such evidence can only exist if $[A]$ is empty as well. why?

We write $[\neg A] = [A] \to \{\}$

Again let us look at concrete examples

- $P \Rightarrow \neg\neg P$: We need a function that takes some evidence $p : [P]$ and produces a function $g_p : ([P] \to \{\}) \to \{\}$, which takes as input a function $h : [P] \to \{\}$ and produces evidence for Q, that is an element of $\{\}$.

  Now since h produces such an element of $\{\}$ if given an element of $[P]$, we only need to apply h to p. g_p would thus be $\lambda h.\, h(p)$ and the overall evidence is $\lambda p.\, (\lambda h.\, h(p))$.

  Note that the argument is the same as for $P \Rightarrow ((P \Rightarrow Q) \Rightarrow Q)$ but Q is replaced by f.
- $\neg(P \vee Q) \Rightarrow \neg P$: We need a function that takes a function $h : ([P] + [Q]) \to \{\}$ and produces a function $g_h : [P] \to \{\}$ that produces an element of $\{\}$ if given a $p : [P]$.

  To produce the element of $\{\}$ we can utilize the function h but we must provide an element of $[P] + [Q]$. For this purpose we use $\mathtt{inl}(p)$ where $p : [P]$ is the input of g_h. Thus g_h is $\lambda p.\, h(\mathtt{inl}(p))$[3] and the overall evidence is $\lambda h.\, (\lambda p.\, h(\mathtt{inl}(p)))$.

The above examples have introduced all the necessary components of the language of evidence terms. The following table summarizes all these components ad groups them according to the logical connective to which they belong. Notice that there are two types of terms for each connective: terms that construct elements of the corresponding evidence type and terms that show how to make use of (or decompose) terms of that form when constructing evidence for other formulas.

proposition A	evidence type $[A]$	evidence term	evidence decomposition
$A \Rightarrow B$	$[A] \to [B]$	$\lambda a.b$	$f(a)$
$A \wedge B$	$[A] \times [B]$	(a, b)	x_1, x_2
$A \vee B$	$[A] + [B]$	$\mathtt{inl}(a)$, $\mathtt{inr}(b)$	$\mathtt{case}\ x\ \mathtt{of}\ \mathtt{inl}(a) \to s \mid \mathtt{inr}(b) \to t$
$\neg A$	$[A] \to \{\}$	$\lambda a.b$	$f(a)$

We could now give a formal definition of the syntax of evidence terms in the style of definition 2.4. We would have to specify the symbols that we may use and then recursively define the construction of evidence terms from evidence variables. Since this does not lead to any new insights we leave it as an exercise to the reader.

Note that the above examples also illustrate the fact that logic could be used as language for specifying programs and that providing evidence for a logical formula is the same as programming: the propositional formula provides a typing of the desired program and the task is to find a program

[2]For the formal language itself, the proposition f is not strictly needed, as it could be viewed as abbreviation for $A \wedge \neg A$. However, it is easier to explain the semantics of negation in terms of f than the other way around, because f is more primitive. Most logic books, however, use negation as primitive and we will follow that tradition.

[3]It may be confusing at first that we are able to describe functions that construct elements of the empty type. However, the function g_h will only produce an element of the empty type $\{\}$ if we are able to give it an element of $[P]$ as input. If $[P]$ does not have any elements, g_h will not produce any elements. In the example g_h is a well-defined function and therefore $[P]$ must be empty, which is exactly what $\neg P$ means.

that satisfies the given typing. While propositional logic can only give a very coarse specification of a program, more expressive logics like first-order logic or type theory are capable of providing specifications at an almost arbitrary level of detail.

Viewing evidence terms as programs also raises the question whether we may compute evidence terms. This is in fact the case, as evidence construction and evidence decomposition are inverses of each other. In the lambda calculus there are *reduction rules* for computing the value of matching combination of decomposition and construction. For instance the first component of a pair (a, b) must obviously be a, so the term $(a, b)_1$ can be *reduced* to a and $(a, b)_2$ to b. The term $(\lambda a.b)(c)$ can be to a version of the term b where every occurrence of a is replaced by c (denoted by $b[c/a]$) and $\texttt{case inl}(c)\ \texttt{of inl}(a) \to s \mid \texttt{inr}(b) \to t$ can be reduced to $s[c/a]$. The details of such a computation system require dealing with *substitution*, *free* and *bound* variables, *termination*, and *confluence* and would be beyond the scope of this course. We refer the interested reader to the literature about lambda calculus and type theory.

Here is an example that shows how evidence is constructed for slightly more complex formulas.

Example 3.1 The formula $((P \vee Q) \wedge ((P \Rightarrow R) \wedge (Q \Rightarrow R))) \Rightarrow R$ states that we can make use of a disjunction $P \vee Q$ to prove some proposition R if we also know $P \Rightarrow R$ and $Q \Rightarrow R$. Intuitively this seems obvious: if we know P then we can use $P \Rightarrow R$ to show R. Otherwise we know Q and we can use $P \Rightarrow R$ to show R. But how do we construct the (formal) evidence for that?

Since evidence construction for a logical proposition A is essentially the same writing a program of type $[A]$ all we have to do is find some program of type $[((P \vee Q) \wedge ((P \Rightarrow R) \wedge (Q \Rightarrow R))) \Rightarrow R]$. Using the table above we first determine the type of the program that we need to construct. We have

$$\begin{array}{ll} & [((P \vee Q) \wedge ((P \Rightarrow R) \wedge (Q \Rightarrow R))) \Rightarrow R] \\ = & [(P \vee Q) \wedge ((P \Rightarrow R) \wedge (Q \Rightarrow R))] \to [R] \\ = & ([(P \vee Q)] \times [(P \Rightarrow R) \wedge (Q \Rightarrow R)]) \to [R] \\ = & (([P]+[Q]) \times ([P \Rightarrow R] \times [Q \Rightarrow R])) \to [R] \\ = & (([P]+[Q]) \times (([P] \to [R]) \times ([Q] \to [R]))) \to [R] \end{array}$$

This means we need to construct a function that takes an input of type

$$([P]+[Q]) \times (([P] \to [R]) \times ([Q] \to [R]))$$

and produces an element of type $[R]$. Apart from the type constraints there are no requirements on that function. Lacking a good memnonic name for the input we simply call it x. We notice that the input comes in as one block of data, so x is actually a tuple consisting of an element of $[P]+[Q]$ and two functions that belong to the types $[P] \to [R]$ and $[Q] \to [R]$. We have to make use of these components to build an element of $[R]$.

The crucial part is the first component. As an element of $[P]+[Q]$ it has either the form $\texttt{inl}(p)$ for some $p : [P]$ or the form $\texttt{inr}(q)$ for some $q : [Q]$[4]. In the first case we could apply the function in $[P] \to [R]$ to p to construct an element of $[R]$ and in the second case could apply the function in $[Q] \to [R]$ to q.

Now we have an idea how to proceed but we also have to find a way to express this properly in the language of evidence terms. If we had names for the element of $[P]+[Q]$ and the two functions, say y, f, and g, we could write the case analysis as

[4]Note that the names p and q are just placeholders. We could equally well have called them y and z but the names p and q help us keep in mind what their role is.

$$\texttt{case}\, y\, \texttt{of}\ \ \texttt{inl}(p) \to f(p) \mid \texttt{inr}(q) \to g(q)$$

However, the language of evidence terms is very simple and does not provide syntactic sugar like general pattern matching, so we can *not* write the desired function as

$$\lambda(y, f, g).\ \ \texttt{case}\, y\, \texttt{of}\ \ \texttt{inl}(p) \to f(p) \mid \texttt{inr}(q) \to g(q)$$

Instead we have to use the operations for accessing the components of a pair to describe the elements y, f, and g in terms of the input x. Because x is a nested tuple $(y, (f, g))$, we know that y is the first component, i.e. x_1, that f is the first subcomponent of the second one, i.e. x_{21}, and that g is x_{22}. If we rephrase the above description using the component notation we get the following evidence term

$$\lambda x.(\texttt{case}\, x_1\, \texttt{of}\ \ \texttt{inl}(p) \to x_{21}(p) \mid \texttt{inr}(q) \to x_{22}(q)$$

This is the evidence term we need. □

The above example shows that combining all the assumptions of a logical inference by conjunctions may appear quite intuitive but causes complications when it comes to expressing evidence, as the evidence for the assumptions first has to be decomposed into smaller components before it can be used. From the perspective of computer science it is better to use iterated implications instead, as these provide the evidences for the assumptions one at a time instead of a big block. The process of converting conjunctions on the left side of an implication into iterated implications is often calle *currying*.

Example 3.2 The formula $(P \vee Q) \Rightarrow ((P \Rightarrow R) \Rightarrow ((Q \Rightarrow R) \Rightarrow R))$ is the curried version of the formula $((P \vee Q) \wedge ((P \Rightarrow R) \wedge (Q \Rightarrow R))) \Rightarrow R$ and should have a similar evidence. The main difference is that the type of the program that we need to construct is now

$$([P]+[Q]) \to (([P]\to[R]) \to (([Q]\to[R]) \to [R]))$$

That is we need to construct a function that takes an input y of type $[P]+[Q]$, an input f of type $[P]\to[R]$, and an input g of type $[Q]\to[R]$ and constructs produces an element of type $[R]$. From the above discussion we already know that this element can be constructed as

$$\texttt{case}\, y\, \texttt{of}\ \ \texttt{inl}(p) \to f(p) \mid \texttt{inr}(q) \to g(q)$$

But in contrast to the above example, we can use the inputs sequentially, so the desired evidence is

$$\lambda y.\lambda f.\lambda g.\ \ \texttt{case}\, y\, \texttt{of}\ \ \texttt{inl}(p) \to f(p) \mid \texttt{inr}(q) \to g(q)$$[5]. □

We conclude this lecture by listing a few example propositions and the evidence that validates them. Note that there are formulas that have no evidence, either because there are counterexamples or because they are unsolvable, i.e. there no way to construct evidence. We will discuss the issue of unsolvability in one of the later lectures. For now we use informal arguments to point out the issue.

- $(P \wedge Q) \Rightarrow (P \vee Q)$

  There are two different evidences: $\lambda x.\, \texttt{inl}(x_1)$ or $\lambda x.\, \texttt{inr}(x_2)$

[5]Note that the case analysis can be performed any time after we have read the input y. In our solution we have delayed it until we have read all three arguments. But we could also have berformed it early on. This would give us the evidence $\lambda y.\ \texttt{case}\ y\ \texttt{of}\ \texttt{inl}(p) \to (\,\lambda f.\lambda g.\, f(p)\,)\ \mid\ \texttt{inr}(q) \to (\,\lambda f.\lambda g.\, g(q)\,)$

- $(P \Rightarrow Q) \Rightarrow ((R \Rightarrow Q) \Rightarrow (R \Rightarrow P))$:

  There cannot be any evidence. A possible counterexample is R being `0=0` and P being `0=1`

  An alternative argument would be the following:
  Any evidence for this formula must have the form $\lambda f.\lambda g.\,(\lambda r.\,p)$ for some $p : [P]$, where r is a variable of type $[R]$. Neither f nor g offer a way to construct an element of P.

- $(P \vee Q) \Rightarrow (P \vee Q)$:

  The evidence is simply $\lambda x.\,x$. There is no need to decompose the evidence x for $P \vee Q$

- $(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$:

  The evidence is $\lambda f.\,(\lambda g.\,(\lambda p.\,g(f(p))))$

- $(P \Rightarrow Q) \Rightarrow (\neg P \Rightarrow \neg Q)$:

  There cannot be any evidence. A possible counterexample is Q being `0=0` and P being `0=1`

  Again there is an alternative argument.
  Any evidence for this formula must have the form $\lambda f.\lambda g.\,(\lambda q.\,x)$ for some $x : \{\}$, where q is a variable of type $[Q]$. The only way to construct an element of $\{\}$ would be using g but for that we need an element of $[P]$ as input. There is no way to construct that element.

- $\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)$:

  The evidence is $\lambda f.\,(\lambda p.\,f(\texttt{inl}(p)),\ \lambda q.\,f(\texttt{inr}(q)))$

- $(\neg P \wedge \neg Q) \Rightarrow \neg(P \vee Q)$:

  The evidence is $\lambda x.\,(\lambda y.\,((\text{case } y \text{ of inl}(p) \to x_1(p) \mid \text{inr}(q) \to x_2(q))))$

- $\neg(\neg P \wedge \neg Q) \Rightarrow (P \vee Q)$:

  Any evidence for this formula must have the form $\lambda f.\,\texttt{inl}(p)$ for some $p : [P]$ or $\lambda f.\,\texttt{inr}(q)$ for some $q : [Q]$ where f is a function in $([\neg P]\times[\neg Q])\to\{\}$. There is no way to construct either p or q from that function, since f can only construct elements of $\{\}$.

- $\neg\neg P \Rightarrow P$:

  Any evidence for this formula must have the form $\lambda f.\,p$ for some $p : [P]$ where f is a function in $[\neg P]\to\{\}$. There is no way to construct p.

- $\neg P \vee P$:

  Any evidence for this formula must have the form $\texttt{inl}(\lambda p.\,x)$ for some $x : \{\}$ or $\texttt{inr}(p)$ for some $p : [P]$. We have nothing in our hands to construct either of the two.

- $(P \vee (Q \wedge R)) \Rightarrow (P \vee Q) \wedge (P \vee R)$:

  The evidence is $\lambda x.(\texttt{case}\ x\ \texttt{of}\ \ \texttt{inl}(p) \to (\texttt{inl}(p), \texttt{inl}(p)) \mid \texttt{inr}(y) \to (\texttt{inr}(y_1), \texttt{inr}(y_2))$