CS 5860 Introduction to Formal Methods

Lecture 5 Thur. Sept. 8, 2011* repeated Tue Sept 13

Topic: Formal Logics

PLAN We will discuss two kinds of logical system
Boolean Logics and Computational Logics

We have seen how types in ML describe or specify the behavior of functions. For example, the Euclidean algorithm analyzed by Gries has inputs x, y of type int and produces outputs q, r of type int. So its type is

$$\text{euclid}: \text{int} * \text{int} \longrightarrow \text{int} * \text{int}$$

$$\text{euclid}(x,y) = (q,r)$$

With assertions we can say more about x, y, q, r. Namely

$$\text{euclid}: \{(x,y): \text{int}*\text{int} \mid x \geq 0 \;\&\; y > 0\} \longrightarrow$$
$$\{(q,r): \text{int}*\text{int} \mid x = q*y + r \;\&\; 0 \leq r < y\}$$

This means that if integer inputs x, y satisfy $x \geq 0, y > 0$, then the output of euclid, (q,r), satisfies $x = q*y + r$ & $0 \leq r < y$.

We need logical expressions and richer types to say this.

First we look at the logical expressions.

* University officially closed due to weather emergency.

CS 5860

Lecture 5 Thur Sept 8, 2011 repeated Tue Sept 13 since Cornell was officially closed until 11am on Sept 8.

We will study an increasingly rich progression of logical systems:

Propositional Logic, First-Order Logic (also called Predicate Logic), Higher-Order Logic (HOL), Type Theory.

Each system can be given in two forms based on different semantics (meanings). The forms have several names.

truth-functional semantics	evidence semantics
Boolean logics	computational logics
Tarski semantics	Brouwer semantics
classical logics ("Aristotelian") ("Platonic")	constructive logics ("non Aristotelian") ("non Platonic" Euclidean)
ambiguous semantics	explicit semantics

Some of these terms are based on a philosophical analysis of how we know (epistemology) and the relationship among things, thoughts about them and words for expressing thoughts.

In philosophy they also say objects, concepts and expressions. In our semantics for computational logics, the expressions will refer directly to the objects.

We look first at Boolean (truth functional) Logics. The basic formulas are Boolean expressions, say of the kind we see in ML built from (not e), $(e_1 \& e_2)$, $(e_1$ or $e_2)$, and $(e_1 \Rightarrow e_2 | e_3)$ where e, e_1, e_2, e_3 are Boolean expressions as are any variables x_i, ~~declared~~ where x_i are declared to be Booleans (bool).

The meaning or semantics of these expressions is given relative to assignments of Boolean values, true and false or more succinctly t and f. We write $\mathbb{B} = \{t, f\}$. We assume the standard functions:

band : $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ where band(x,y) = t iff both x and y have been assigned t.

bor : $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ where bor(x,y) = t iff one of x or y is assigned the value t

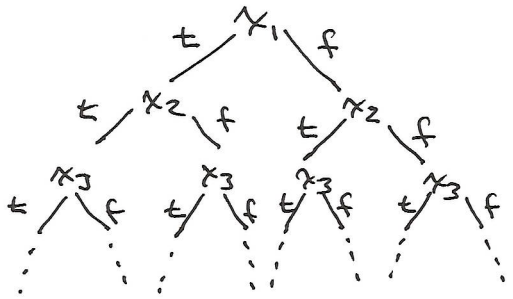
bnot : $\mathbb{B} \to \mathbb{B}$ where bnot(x) = t iff x is assigned f.

Let a : Var $\to \mathbb{B}$ where Var = $\{x_1, x_2, \cdots x_n, \cdots\}$, then given a Boolean expression e we can define its value given a recursively bval : (Bexp × Var $\to \mathbb{B}$) $\to \mathbb{B}$ is defined by cases on the expression e. The cases are that e is a variable x, an &, ~~or~~ $(e_1 \& e_2)$ an or, $(e_1$ or $e_2)$ or (not e).

bval(e,a) = case e
- var(v) → a(v)
- and (e_1, e_2) → band(bval(e_1,a), bval(e_2,a))
- or (e_1, e_2) → bor (bval(e_1,a), bval(e_2,a))
- not(e_1) → bnot (bval(e_1,a))

esac

More economical and realistic typing: the type Var → 𝔹 consists of a large (uncountable) collection of functions if we assign a value to every variable $x_1, x_2, \ldots, x_n, \ldots$. We can imagine the type as an infinite tree



Given a Boolean expression such as (not (x_1 & (x_3 v (not(x_1 & x_2))))) we only need to know the assignment to x_1, x_2, x_3.
Given e : Bexp, let Var(e) be the variables occurring in e. Then a better type for bval is

$$bval : (e : Bexp \times Var(e)) \to \mathbb{B}.$$

But this type is not available in ordinary programming languages (such as ML, Java, C#, Erlang, F#, etc.). It is available in Coq, Agda, Nuprl, the theorem provers and their programming languages, and Microsoft is designing a language F-star that will have these dependent types, e.g. the type Var(e) depends on the value of e an element of Bexp.

We will use dependent types in lecture and in the readings. Students who experiment with Nuprl will use them in programming. There is an experimental programming language from Microsoft with dependent types called F* (FStar), see the course Web page.

Using Bexp we can define some well known combinatorial problems from the theory of algorithms, the well known NP-complete problems such as Bexp-SAT, SAT, 3SAT, Graph-Coloring, Hamiltonian-Circuits, Quadratic Congruence, etc.

Bexp-SAT is the problem of deciding whether a Boolean expression e has value t for some assignment to its variables. Symbolically we say

Given $e \in Bexp$ can we find $a : Vac(e) \to \mathbb{B}$
such that $bval(e, a) = t$

more symbolically

Given $e : Bexp$ $\quad \exists\, a : Vac(e) \to \mathbb{B}\,(bval(e,a) = t)$

SAT is the problem of deciding whether a Bexp in conjunctive normal form, CNF, is satisfiable. A CNF formula is of the form $(x_1 \vee \overline{x}_2 \vee x_3) \,\&\, (\overline{x}_1 \vee x_2 \vee x_3 \vee \overline{x}_4) \,\&\, (x_2 \vee \overline{x}_1 \vee \overline{x}_3 \vee x_4) \,\&\, (x_2 \vee \overline{x}_3 \vee x_4)$ where $\overline{x}_i$ is (not x_i). We say that $\overline{x}_i$ is a literal. A CNF formula consists of clauses $C = (\ell_{i_1} \vee \cdots \vee \ell_{i_n})$ where the ℓ_i are literals. The CNF formula is a conjunction $C_1 \,\&\, C_2 \,\&\, \cdots \,\&\, C_n$ of clauses. CNF means Conjunctive Normal Form.

We say that a combinatorial problem is solvable in Polynomial time iff there is an algorithm that solves it in a polynomial number of steps in the size of the problem. For Bexp-Sat and SAT the size is the length of the Boolean expression e.

See the Web page for a longer discussion (NP Problems).

Logic is mainly concerned with expressions that are logically true. In the case of Bexp these are called tautologies.

Definition We say that Bexp e is a tautology iff

for all $a : Var(e) \to \mathbb{B}$, the value of e is t,

that is

$$\forall a : Var(e) \to \mathbb{B}.\ (bval(e,a) = t).$$

Here are simple examples of tautologies. Some of them have names, or rather their proofs have names associated with the expression. Define $(e_1 \supset e_2)$ iff $((not\ e_1) \vee e_2)$

1. $X \supset X$ called I
2. $X \supset (Y \supset X)$ called K
3. $X \& (Y \vee Z) \supset ((X \& Y) \vee (X \& Z))$
4. $\sim(X \vee Y) \supset (\sim X \& \sim Y)$ where $\sim X$ is $(not\ X)$
5. $\sim(X \& Y) \supset (\sim X \vee \sim Y)$

6. $(X \supset Y) \supset (X \supset (Y \supset Z)) \supset (X \supset Z)$ called S
7. $(X \supset Z) \supset ((Y \supset Z) \supset (X \vee Y) \supset Z)$

Proofs Truth tables express the truth-functional semantics very clearly, and they could be used to prove that an expression e is a tautology. But these "proofs" are guaranteed to be very long, exponential size in the number of variables. So logicians look for more compact ways of showing that an expression is a tautology. One method are the tableau proofs that we teach in CS 4860. These proofs are systematic searches for a counter example. You can read about them in the supplemental material posted at the Web page under Tableau Proofs.