CS1Bh Lecture Note 19

Predicate Logic

Uses of Propositional Logic

Pure propositional logic, where all propositions have at the leaves of their syntax trees just constants *t* and *f* and variables, turns out to be suprisingly useful. For example, consider the puzzle:

> Can 7 black or white beads be arranged on a bracelet such that no two black or two white beads are adjacent?

We can construct a proposition that is satisfiable exactly when the puzzle can be solved. Introduce propositional variables $B_1, \ldots, B_7$, where any value we might choose for B_i will specify a colour for bead i (say '*t*' is *black*, '*f*' is *white*). The puzzle is solvable just when the propositional formula

$$(B_1 \Leftrightarrow \neg B_2) \wedge (B_2 \Leftrightarrow \neg B_3) \wedge (B_3 \Leftrightarrow \neg B_4) \wedge (B_4 \Leftrightarrow \neg B_5) \wedge (B_5 \Leftrightarrow \neg B_6)$$
$$\wedge (B_6 \Leftrightarrow \neg B_7) \wedge (B_7 \Leftrightarrow \neg B_1)$$

is satisfiable, and, moreover, any satisfying assignment gives a colouring. Do you think this puzzle is solvable? For what numbers of beads is it solvable?

There is much interest in both academia and industry in being able to automatically check the satisfiability of large formulas with 10^5 or more variables. For example Intel can phrase statements about the correctness of their chip designs as propositional formulas. If a formula is satisfiable, the satisfying assignment usually indicates a bug in the chip's design. If the formula is unsatisfiable, they have some assurance of the design's correctness.

Nevertheless, there is much we want to be able to express in precise logical notation that we cannot express in propositional logic. This is where predicate logic comes in. Predicate logic allows one to make a definition such as:

> A number n is *prime* just when it is only divisible by itself and 1.

or make an assertion in a program such as

> Every element of the **`int`** array `a` is non-zero.

Introduction to Predicate Logic

Predicate logic extends propositional logic with *atomic relations*, *functions*, *individual variables* and *quantifiers*. We'll describe what we mean by each in turn.

Atomic Relations

Relations are true or false properties of one or more arguments. For example

- *Primeness* is a unary relation on numbers. We might use the syntax *prime(n)* to write the assertion that number n is prime.
- $\leq$ is a binary relation on integers. We usually use the infix syntax $i \leq j$ to writte the assertion that i is less than or equal to j.

Relations are also known as predicates, hence the name *predicate* logic.

Atomic relations are basic relations from which we can use propositional and predicate logic to build more complicated relations. For example, we might decide that $=$ and $<$ are atomic binary relations, and that $\leq$ is defined in terms of these:

$$i \leq j \Leftrightarrow i = j \vee i < j$$

Functions and Constants

Predicate logic allows the arguments to relations to be built up from functions and constants. For example, the arithmetic expression $2 + (4 \times -3)$ is built up from the binary functions $+$ and $\times$, the unary function $-$, and constants $2, 3, 4$.

Individual Variables

Individual variables are variables that can occur in the expressions that are arguments to relations. For example, the relation $x > y \Rightarrow x^2 > y^2$ contains the individual variables x and y.

Individual variables are to be contrasted with *propositional* variables in propositional logic. For example, A and B in the formula $A \Rightarrow B \wedge A$. The most common forms of predicate logic do *not* allow propositional variables. For this reason, if it is clear we are discussing predicate logic, we usually drop the adjective *individual*.

Predicate logics can allow relations that take no arguments, and these can largely take on the role that propositional variables play in propositional logic.

Quantifiers

The true power of predicate logic comes from the *quantifiers*

- $\forall$ – read as *for all*
- $\exists$ – read as *there exists*

For example, we can write

$$\forall x.\ x^2 \geq 0,$$

read as *for all x, x squared is greater than or equal or zero*, or, more succinctly *every square is greater than or equal to zero.* The binary relation on integers *divides*, as in *i divides j*, or written more formally *divides(i,j)*, can be defined by the predicate logic formula

$$\exists k.\ k \times i = j$$

read as *there exists a k such that* $k \times i = j$. The unary relation *prime(n)* can be defined by

$$\forall d.\ divides(d, n) \Rightarrow d = n \vee d = 1\ ,$$

read as *every divisor* d *of* n *is either equal to* n *or equal to* 1.

Syntactic classes

Predicate logic has two distinct syntactic classes of expressions

- *term* expressions, built from constants, functions and individual variables,
- *formula* expressions, built from atomic relations, the propositional logic connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, etc.), and the quantifiers $\forall$ and $\exists$.

In practice, just as with programming in Java, term expressions usually have associated types, and it's not legitimate to use an expression of one type in places expressions of another type are expected (ignoring for sake of simplicity issues of subtyping).

Quantifier Scope

Consider the predicate logic expression

$$\exists x.\ P \Rightarrow Q$$

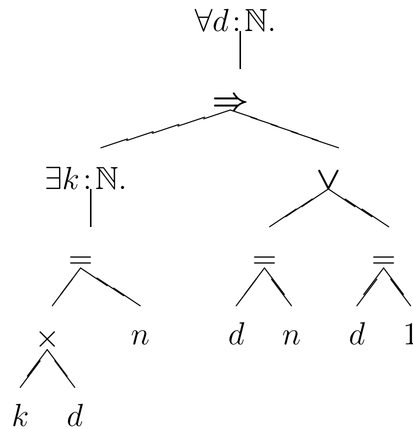
Do we mean by this $\exists x.\ (P \Rightarrow Q)$ or $(\exists x.\ P) \Rightarrow Q$? The most common convention is that the *scope* of a quantifier extends as far as possible to the right, subject to restrictions of parenthesisation. In this case, the scope of quantifier $\exists x$ is the whole expression $P \Rightarrow Q$, so the implied parenthesisation is $\exists x.\ (P \Rightarrow Q)$.

Syntax Trees

The structure of any predicate logic expression can be expressed using a syntax tree. This makes scoping particularly clear. For example, consider the characterisation above of a number n being prime, with the definition of *divides* expanded out and quantification ranges (see next section) given explicitly as $\mathbb{N}$:

$$\forall d{:}\mathbb{N}.\ (\exists k{:}\mathbb{N}.\ k \times d = n) \Rightarrow d = n \vee d = 1\ ,$$

The corresponding syntax tree is:



Ranges of Quantifiers

Apart from in the last example, we've been informal so far about the set of values quantified variables range over. We've said in English that, when we write $\forall x.\ x^2 \geq 0$, we mean that 'for all x, $x^2 \geq 0$', but we haven't said whether x is say a natural number, an integer or a real number. When we write

$$\forall x\!:\!\mathbb{N}.\ x^2 \geq 0\ ,$$

we are specifying that the range of quantification (sometimes called the *domain of quantification*) for x is the natural numbers $\mathbb{N}$ ($\{0, 1, 2, \ldots\}$). Likewise, if we write

$$\exists k\!:\!\mathbb{Z}.\ k \times i = j,$$

we are specifying that k ranges over the integers $\mathbb{Z}$.

Restricting Ranges

Say we have some sequence $a_1, \ldots, a_n$ of integers. We might want to assert that all the a_i are non-negative. We could write a predicate logic formula asserting this using the notation:

$$\forall i\!:\!\{1 \ldots n\}.\ a_i \geq 0\ \ .$$

Here we have used the common mathematical notation $\{1\ldots\text{n}\}$ for the set of values $\{k \in \mathbb{Z} | 1 \leq k \wedge k \leq n\}$. An alternative equivalent way of writing the above formula is

$$\forall i\!:\!\mathbb{Z}.\ 1 \leq k \wedge k \leq n \Rightarrow a_i \geq 0\ \ .$$

This idiom of including some information restricting a $\forall$ quantifier using the left-hand side of an implication is particularly useful when the flavour of predicate logic we might be using only permits quantification over simple sets such as $\mathbb{Z}$ and not subranges of integers such as $\{1\ldots\text{n}\}$.

Say instead we want to merely assert that *some* a_i is non-negative. This can be phrased as the formula

$$\exists i\!:\!\{1 \ldots n\}.\ a_i \geq 0\ \ .$$

With $\exists$, we use $\wedge$ to specify restrictions on the quantification outside of the quantifier type. For example, the above existential formula is equivalent to

$$\exists i\!:\!\mathbb{Z}.\ (1 \leq k \wedge k \leq n) \wedge a_i \geq 0\ \ .$$

Finite Ranges

The $\forall$ quantifier can be thought of as equivalent to the repeated use of $\wedge$ when the range of quantification is finite. For example, if we allow the notation:

$$\bigwedge_{i=1}^{n} P_i \equiv P_1 \wedge P_2 \wedge \ldots \wedge P_n \ ,$$

then the predicate logic formula

$$\forall i\!:\!\{1\ldots n\}.\ a_i \geq 0 \ \ .$$

could be written as

$$\bigwedge_{i=1}^{n} a_i \geq 0$$

Similarly, $\exists$ quantification over a finite range can be thought of as repeated $\vee$.

This equivalence doesn't hold when the range of quantification is infinite. There is no way to write down a propositional expression built of $\wedge$s or $\vee$s that is infinitely large.

Rules of Predicate Logic

We give here just one example of a pair of rules.

The De Morgan laws in propositional logic specify how negation can cause $\wedge$s and $\vee$s to swap:

$$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$$
$$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$$

There are similar laws in predicate logic

$$\neg(\forall x.P) \Leftrightarrow \exists x.\neg P$$
$$\neg(\exists x.P) \Leftrightarrow \forall x.\neg P$$

Predicate Logic in Programming

Predicate logic is very useful for writing program *assertions* as introduced in Note 6 on Induction and Invariants. Consider a class `A` with an integer array field `x` and a method `findMin()` for finding the minimum value stored in `x`.

```
class A {
  private int[] x;

  public int findMin() {
    int min = x[0];

    for (int i = 1; i != x.length ; i++) {
      if (x[i] < min) min = x[i];
    }
    return min;
  }
}
```

Let us assume that the array `x` will always have non-zero length. One condition that we wish to hold just before the `findMin()` method exits (the method's *post-condition*) could be written as:

$$\forall n\!:\!\texttt{int}.\ 0 \leq n \wedge n < \texttt{x.length} \ \Rightarrow\ \texttt{min} \leq \texttt{x[n]}$$

The condition that we wish always to hold when the `findMin` method starts (the method's *precondition*) is, to be precise,

$$\texttt{x} \neq \texttt{null} \wedge \texttt{x.length} \neq 0$$

Here we are at the edge of what normal predicate logic can express. The formula $\texttt{x.length} \neq 0$ doesn't even make sense unless we know that $\texttt{x} \neq \texttt{null}$. What is needed here is a conditional operator more like the Java `&&` where the right hand argument is only considered in the event the left hand argument is true.

If reasoning about the correctness of this program, it is useful to have an assertion about the state of affairs just before each and every `i != x.length` test in the `for` loop. A check in this position is called a *loop invariant*.

A correct loop invariant is:

$$\forall n\!:\!\texttt{int}.\ 0 \leq n \wedge n < \texttt{i} \Rightarrow \texttt{min} \leq \texttt{x[n]}$$

This asserts that `min` is a lower bound on the values in the array `x` at indices below the index `i`. Note how this is trivially true when the loop starts with $\texttt{i} = 1$, and is equivalent to the post-condition when `i` reaches `x.length` and the loop terminates. A full justification for why this method works would argue how this assertion remains true from one loop iteration to the next.

The JML and ESC/Java Assertion Languages

As you might already well appreciate, writing program assertions and checking them by hand can be a difficult business. Various research projects are underway to provide tool support for assertion checking. Two such projects are JML and ESC/Java:

- *JML*: JML provides a rich language for describing the interface behaviour of classes (e.g. the pre and post conditions of each method in a class) in a Java-friendly predicate logic. One JML tool can automatically add extra code to a class's implementation to check at *run-time* that these assertions are obeyed whenever methods in the class are called or return. Another JML tool can automatically generate test code based on a JML specification.

- *ESC/Java*: This provides a very similar but simpler language for formulas, but allows a richer range of assertions *within* methods (e.g. loop invariants). The beauty of the ESC/Java tool is that it checks assertions statically. One runs it much as one runs the `javac` compiler. The tool reasons logically about program structure and can be more thorough than the JML run-time checks which will only check assertions for data values you provide in your program tests.

The example class above, annotated liberally using the ESC/Java language looks like:

```
class A {
  private int[] x;

  //* requires x != null && x.length != 0;
  //* ensures (\forall int n; 0 <= n & n < x.length ==> \result <= x[n]);

  public int findMin() {
    int min = x[0];

    //* assert min == x[0];

    //* loop_invariant (\forall int n; 0 <= n & n < i ==> min <= x[n]);
    for (int i = 1; i != x.length ; i++) {
      if (x[i] < min) min = x[i];
    }
    //* assert (\forall int n; 0 <= n & n < x.length ==> min <= x[n]);
    return min;
  }
}
```

The various conditions are added in special comments (starting with `//*`) called *pragmas* to ensure that regular Java compilers don't pick them up. The `loop_invariant` and `assert` pragmas should be self explanatory. The `requires` pragma is used for the preconditions and `ensures` pragma for post-conditions. In the `ensures` pragma, `\result` stands for a method's return value.

The predicate logic used in these assertions is roughly the side-effect free subset of Java expressions (including method calls), with then a few extensions including the quantifiers `\forall` and `\exists` and logical connectives `<==>` and `==>`.

These languages are too advanced to be used in CS1, but we are considering adopting them in later years of the CS curriculum.

Writing Run-Time Assertions

You don't have to use ESC/Java or JML to add some run-time assertion checking to your code. Java versions 1.4 and later handle an assertion statement of form

`assert` *boolean-expression* `:` *Object-expression* `;`

If *boolean-expression* evaluates to `true`, then the statement has no effect, but if it evaluates to `false`, an exception is thrown with the value of *Object-expression*. No extensions are made to Java to support predicate-logic formulas, so quantifiers have to be coded using loops. For example, the pre- and post-condition checks for the `findMin()` method in the class `A` could be coded as:

```
class A {
  private int[] x;

  public int findMin() {
    assert x != null && x.length != 0 : "A.findMin: precondition" ;

    int min = x[0];

    for (int i = 1; i != x.length ; i++) {
      if (x[i] < min) min = x[i];
    }
    assert checkPostcondition(min) : "A.findMin: postcondition" ;
    return min;
  }
  private boolean checkPostcondition(int min) {
    boolean val = true;
    for (int i = 0; i != x.length; i++) {
      val = val & (x[i] <= min);
    }
    return val;
  }
}
```

Assertion checking can significantly slow code down. Java 1.4 allows checking of `asserts` to be selectively turned on or off at compile time, so no source code changes are necessary when going from code testing to code distribution.

Assertions across Time

Postcondition assertions for a method often need to refer to the values expressions had at the method start. In JML and ESC/Java one refers in a postcondition to the start value of an expression e with the syntax `\old(`e`)`. One can write `\old(this)` to refer to the initial value of the object the method is invoked on. If writing runtime assertions, one has to add extra variable declarations to the method start which can capture relevant expression values for later checking by a postcondition assertion.

References on Assertion Languages and Tools

- *Java 1.4 Assertions*:
  `http://java.sun.com/j2se/1.4.1/docs/guide/lang/assert.html`
- *JML*: `http://www.jmlspecs.org/`
- *ESC/Java*: `http://research.compaq.com/SRC/esc`

Paul Jackson,
16th April 2003,
.