Relations

Klaus Draeger

Lecture 5: Logic

- Logic gives us a general way of talking about **properties** of a system
- The most basic version is **Propositional** Logic
  - Propositional (Boolean) **variables** with **true** / **false** values
  - Boolean **operators** $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$
    (&, or, not, =>, <=> in the ascii notation)
  - Values defined by **truth tables**
- Extended in two directions:
  - **Predicate** Logic
    - Statements about numbers, sets, ... replace Boolean variables
  - **Temporal** Logic
    - Adds operators to deal with values changing over time
- We will focus on Predicate Logic for now.

Propositional Logic

- Propositional Logic formulas are built from:
  - A set **Var** of Boolean variables like **x, itIsRaining, bit7, . . .**
    – Each taking a value in **true, false**
  - Boolean operators to build more complex formulas
- Most basic question: given
  - A formula **F**
  - A valuation **v : Var -> {true, false}**

  is F **satisfied** by v?
- More complicated: given a formula **F**, is it
  - **Satisfiable**, i.e. is there **at least one** v that satisfies it?
  - **Valid**, i.e. always satisfied?
    Note that this is equivalent to ¬F being unsatisfiable

Evaluating Formulas

- Generally evaluation of **P** is **recursive**:
  - If P is just a variable, use its value from v directly
  - If P has the form **P1 op P2** where op is a Boolean operator,
    - First evaluate **P1** and **P2**
    - Then apply **op** to their values

    using the usual truth table below
  - Note that formulas like **A & B or C** can be ambiguous:
    - Could mean **(A & B) => C** or **A & (B => C)**
    - Resolved using **precedence** (**&** before **or**, . . .)
    - If in doubt, use explicit parentheses

P1	P2	not P1	P1 & P2	P1 or P2	P1 => P2	P1 <=> P2
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Boolean Satisfiability

- To check if **P** is **satisfiable**:
  - We could try all combinations of variable values
    - But for **n** variables there are 2^n combinations
    - Realistic applications can have thousands of variables
    - Example: Modelling the behaviour of a CPU, checking if invalid states are possible
  - Or transform it into CNF
    e.g. (A or not B or C) & (not A or D) & B & (C or not D)
    Then use a SAT algorithm like DPLL:
    - B appears on its own, so its value is forced to be true
    - C appears only un-negated, so set its value to true
    - Repeat these steps, always simplifying the formula
    - Only if none apply, guess a value for one variable;
      May need to backtrack if this does not lead to a solution

DPLL example

- Formula:
  (A or not B or C) & (not A or D) & B & (C or not D)
- B appears on its own and must be **true**
  Remaining formula: (A or C) & (not A or D) & (C or not D)
- C only occurs un-negated, so set it to **true**
  Remaining formula: (not A or D)
- D only occurs un-negated, so set it to **true**
  Nothing left to do, the formula is satisfied

Predicate Logic

- Usually our formulas don't just contain Boolean variables
  - Example: **x : A & y : B & x+y : A**$/\backslash$ **B**
  - Code: conditions in **while** loops and conditionals
  - Specifications: Pre-conditions, invariants
- Instead we have **atomic** formulas
  - Comparisons: **s = t, s < t, . . .**
  - Set membership: **s : A**

  where s, t, A can be
  - variables of integer/set/. . . type
  - more complicated terms using arithmetic/set-theoretic/. . . operations

Predicate Logic

- Predicate Logic deals with **Predicates** involving the data – Hence the name
- This includes given ones like comparisons or set membership
- Can define new ones in the DEFINITIONS section
  Example: isEven(nn) == (nn : NAT & nn mod 2 = 0)
- This becomes more relevant for more complex cases
  Examples: partialOrder, equivalence from last week

Satisfiability in Predicate Logic

- Satisfiability is more complex in Predicate Logic.
- Suppose we have this formula:
  (x<y or x<z) & (y<x or y<z) & (z<x or z<y)
- A straightforward approach would be to:
  - Introduce a Boolean variable for each atomic formula:
    P1,. . . ,P6 for x<y, x<z, y<x, y<z, z<x, z<y
  - Solve the resulting propositional formula
    (P1 or P2) & (P3 or P4) & (P5 or P6)
- The problem is that the atomic formulas are **not independent**
- For example:
  - x<y and y<x cannot both be true
  - x<y, y<z, and z<x cannot all be true

Satisfiability Module Theories

- Tools to solve this satisfiability problem are SMT solvers (for "Satisfiability Module Theories")
- They combine
  - A general (Boolean) SAT solver
  - Specialised solvers for theories (arithmetic, set theory, ...)
- In a nutshell, they
  - Solve the propositional version as on the previous slide
  - Ask the theory solvers if the combination of truth values is "realisable"
    e.g. ask the arithmetic solver if x<y and y<x can be true
  - If **yes**, we have a solution;
    If **no**, add a constraint to capture this conflict, and retry.
    For example, it would add **(not P1 or not P3)** after finding out that x<y and y<x are incompatible

Quantifiers

- So far our formulas have used
  - **variables**, e.g. x or S
  - **specific values**, e.g. 2 or {}
- We often need to state that a condition is true for:
  - **at least one** value of the variables, i.e. **satisfiable**, or
  - **all** values of the variables, i.e. **valid**.

  This is achieved by using **quantifiers.**
- There are two main quantifiers:
  - **Existential** (written $\exists$ or "#" in ascii) meaning "there exists"
  - **Universal** (written $\forall$ or "!" in ascii) meaning "for all"

Quantifier Examples

- To express that **xx** is a square we write
  #(yy).(yy:NAT & xx=yy*yy)
  i.e. there is a natural number yy such that xx = yy*yy
- To express that all natural numbers are at least 0 we write
  !(xx).(xx:NAT => xx>=0)
- Note the structure of these: we always write
  - #(xx).(xx:SS & PP)
    i.e. "There is an xx which is in SS and satisfies PP"

  or
  - !(xx).(xx:SS => PP)
    i.e. "For all xx, if xx is in SS then it satisfies PP"

More Examples

- We can combine multiple quantifiers for more complex statements.
- Suppose we have a function **ff : AA +-> BB.**
  - To express that ff is **total**, we could write:
    "For all xx:AA there is yy:BB with (xx,yy):ff", in logic:
    !(xx).(xx:AA => #(yy).(yy:BB & (xx,yy):ff))
  - To express that ff is **surjective**, we could write:
    "For all yy:BB there is xx:AA with (xx,yy):ff", in logic:
    !(yy).(yy:BB => #(xx).(xx:AA & (xx,yy):ff))

More Examples

- We can combine multiple quantifiers for more complex statements.
- Suppose we have a function **ff : AA +-> BB.**
  - To express that ff is **total**, we could write:
    "For all xx:AA there is yy:BB with (xx,yy):ff", in logic:
    !(xx).(xx:AA => #(yy).(yy:BB & (xx,yy):ff))
  - To express that ff is **surjective**, we could write:
    "For all yy:BB there is xx:AA with (xx,yy):ff", in logic:
    !(yy).(yy:BB => #(xx).(xx:AA & (xx,yy):ff))

Quantification Order

- Note that the order of quantifiers matters:
- Suppose we have the sets **AA = {4,6,10}** and **BB = {2,3,5}.**
- !(xx)**.**(xx:AA => #(yy)**.**(yy:BB & (xx mod yy = 0))) means: "For each xx:AA there is a yy:BB which divides it"
- #(yy)**.**(yy:BB & !(xx)**.**(xx:AA => (xx mod yy = 0))) means: "There is a yy:BB which divides every xx:AA"
- The first is true (we can choose a different yy for each xx) but the second is false (we would have to use the same yy for all xx)

More Examples

- Quantifiers of the same kind can be combined.
  - To express that a function ff is **injective**, we could write: "For all xx:AA and yy:AA, if xx and yy are different then so are ff(xx) and ff(yy)":
    !(xx,yy).(xx:AA & yy:AA => (xx/=yy => ff(xx)/=ff(yy)))
  - To express that 221 is a sum of two squares, we could write: "There are xx:NAT and yy:NAT such that xx*xx+yy*yy=221":
    #(xx,yy).(xx:NAT & yy:NAT & xx*xx+yy*yy=221)

More Examples

- Quantifiers of the same kind can be combined.
  - To express that a function ff is **injective**, we could write: "For all xx:AA and yy:AA, if xx and yy are different then so are ff(xx) and ff(yy)":
    !(xx,yy).(xx:AA & yy:AA => (xx/=yy => ff(xx)/=ff(yy)))
  - To express that 221 is a sum of two squares, we could write: "There are xx:NAT and yy:NAT such that xx*xx+yy*yy=221":
    #(xx,yy).(xx:NAT & yy:NAT & xx*xx+yy*yy=221)

Logic and Sets

- Predicate Logic can also be used in set comprehensions:
- Suppose we want to define the composition **RR;SS** of relations **RR,SS : NAT<->NAT** without using the actual composition operator
- Can you figure out how?
- We want the set of all pairs **pp = (xx,zz)** such that **there exists yy** with (xx,yy):RR and (yy,zz):SS

$$
\begin{aligned}
RR; SS = \{pp | pp : NAT * NAT \ \& \\
\#(yy).((prj1(pp), yy) : RR \ \& \\
(yy, prj2(pp)) : SS)\}
\end{aligned}
$$

Logic and Sets

- Predicate Logic can also be used in set comprehensions:
- Suppose we want to define the composition **RR;SS** of relations **RR,SS : NAT<->NAT** without using the actual composition operator
- Can you figure out how?
- We want the set of all pairs **pp = (xx,zz)** such that **there exists yy** with (xx,yy):RR and (yy,zz):SS

$$
\begin{aligned}
RR; SS = \{pp | pp : NAT * NAT \ \& \\
\#(yy).((prj1(pp), yy) : RR \ \& \\
(yy, prj2(pp)) : SS)\}
\end{aligned}
$$

Exercises

Try to express the following predicates:

- pp:NAT is a **prime number**
- ff:NAT–>NAT is **strictly increasing**
- The set of sets SS<:POW(NAT) is a **partition**,
  i.e. none of the sets in SS have any elements in common
  (for example, {{1,2,3},{4,7},{5,9}} is a partition)
- ff:NAT*NAT–>NAT has **inverses**,
  i.e. for any xx:NAT there is some yy:NAT with ff(xx,yy) = 0