Computational Logic

A Motivational Introduction
Computational Logic

- Logic of Computation
  - Program verification
  - Proving properties

- Declarative Programming
  - Direct use of logic as a programming tool

- Logic programming
- Functional programming
- Knowledge representation
- Logic and AI
- Lambda calculus
- Constraints
- Logic of programming
- Declarative programming
- Algorithms
- Verification
- Logic
Conventional models of using computers – not easy to determine correctness!

- Has become a very important issue, not just in safety-critical apps.
- Components with assured quality, being able to give a warranty, ...
- Being able to run untrusted code, certificate carrying code, ...
A Simple Imperative Program

- Example:

```c
#include <stdio.h>
main() {
    int Number, Square;
    Number = 0;
    while(Number <= 5)
        { Square = Number * Number;
          printf("%d\n",Square);
          Number = Number + 1; } }
```

- Is it correct? With respect to what?

- A suitable formalism:
  - to provide *specifications* (describe problems), and
  - to reason about the *correctness of programs* (their *implementation*).

is needed.
Natural Language

“Compute the squares of the natural numbers which are less or equal than 5.”

Ideal at first sight, but:

- verbose
- vague
- ambiguous
- needs context (assumed information)
- ...

Philosophers and Mathematicians already pointed this out a long time ago...
Logic

- A means of clarifying / formalizing the human thought process
- Logic for example tells us that (classical logic)
  Aristotle likes cookies, and
  Plato is a friend of anyone who likes cookies
  imply that
  Plato is a friend of Aristotle
- Symbolic logic:
  A shorthand for classical logic – plus many useful results:
  \[ a_1 : \text{likes}(\text{aristotle}, \text{cookies}) \]
  \[ a_2 : \forall X \text{ likes}(X, \text{cookies}) \rightarrow \text{friend}(\text{plato}, X) \]
  \[ t_1 : \text{friend}(\text{plato}, \text{aristotle}) \]
  \[ T[a_1, a_2] \vdash t_1 \]
- But, can logic be used:
  - To represent the problem (specifications)?
  - Even perhaps to solve the problem?
For expressing specifications and reasoning about the correctness of programs we need:

- Specification languages (assertions), modeling, ...
- Program semantics (models, axiomatic, fixpoint, ...).
- Proofs: program *verification* (and debugging, equivalence, ...).
Generating Squares: A Specification (I)

Numbers —we will use “Peano” representation for simplicity:
0 → 0 1 → s(0) 2 → s(s(0)) 3 → s(s(s(0))) …

- Defining the natural numbers:
  \(\text{nat}(0) \land \text{nat}(s(0)) \land \text{nat}(s(s(0))) \land \ldots\)

- A better solution:
  \(\text{nat}(0) \land \forall X \ (\text{nat}(X) \rightarrow \text{nat}(s(X))))\)

- Order on the naturals:
  \(\forall X \ (\text{le}(0, X)) \land \forall X \forall Y \ (\text{le}(X, Y) \rightarrow \text{le}(s(X), s(Y)))\)

- Addition of naturals:
  \(\forall X \ (\text{nat}(X) \rightarrow \text{add}(0, X, X)) \land \forall X \forall Y \forall Z \ (\text{add}(X, Y, Z) \rightarrow \text{add}(s(X), Y, s(Z))))\)
Generating Squares: A Specification (II)

- Multiplication of naturals:
  \[ \forall X \ (\text{nat}(X) \rightarrow \text{mult}(0, X, 0)) \land \]
  \[ \forall X \forall Y \forall Z \forall W \ (\text{mult}(X, Y, W) \land \text{add}(W, Y, Z) \rightarrow \text{mult}(s(X), Y, Z)) \]

- Squares of the naturals:
  \[ \forall X \forall Y \ (\text{nat}(X) \land \text{nat}(Y) \land \text{mult}(X, X, Y) \rightarrow \text{nat\_square}(X, Y)) \]

We can now write a specification of the (imperative) program, i.e., conditions that we want the program to meet:

- **Precondition:**
  empty.

- **Postcondition:**
  \[ \forall X (\text{output}(X) \leftarrow (\exists Y \ \text{nat}(Y) \land \text{le}(Y, s(s(s(s(s(0))))))) \land \text{nat\_square}(Y, X))) \]
Alternative Use of Logic?

• So, logic allows us to *represent problems* (program specifications).

i.e., the process of implementing solutions to problems.

• The importance of Programming Languages (and tools).

• Interesting question: can logic help here too?
Assuming the existence of a *mechanical proof method* (deduction procedure) a *new view of problem solving and computing is possible* [Greene]:

- program once and for all the deduction procedure in the computer,
- find a suitable *representation* for the problem (i.e., the *specification*),
- then, to obtain solutions, ask questions and let deduction procedure do rest:

- No correctness proofs needed!
Computing With Our Previous Description / Specification

<table>
<thead>
<tr>
<th>Query</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{nat}(s(0)) ) ? \text{true}</td>
<td>\text{(yes)}</td>
</tr>
<tr>
<td>( \exists X \ \text{add}(s(0), s(s(0)), X) ) ?</td>
<td>( X = s(s(s(0))) )</td>
</tr>
<tr>
<td>( \exists X \ \text{add}(s(0), X, s(s(s(0)))) ) ?</td>
<td>( X = s(s(0)) )</td>
</tr>
<tr>
<td>( \exists X \ \text{nat}(X) ) ? \text{true}</td>
<td>( X = 0 \lor X = s(0) \lor X = s(s(0)) \lor \ldots )</td>
</tr>
<tr>
<td>( \exists X \ \exists Y \ \text{add}(X, Y, s(0)) ) ?</td>
<td>( (X = 0 \land Y = s(0)) \lor (X = s(0) \land Y = 0) )</td>
</tr>
<tr>
<td>( \exists X \ \text{nat}_\text{square}(s(s(0)), X) ) ?</td>
<td>( X = s(s(s(0))) )</td>
</tr>
<tr>
<td>( \exists X \ \text{nat}_\text{square}(X, s(s(s(s(0))))) ) ?</td>
<td>( X = s(s(s(0))) )</td>
</tr>
<tr>
<td>( \exists X \ \exists Y \ \text{nat}_\text{square}(X, Y) ) ?</td>
<td>( (X = 0 \land Y = 0) \lor (X = s(0) \land Y = s(0)) \lor (X = s(s(0)) \land Y = s(s(s(s(0))))) \lor \ldots )</td>
</tr>
<tr>
<td>( \exists X \ \text{output}(X) ) ?</td>
<td>( X = 0 \lor X = s(0) \lor X = s(s(s(s(0)))) \lor X = s^9(0) \lor X = s^{16}(0) \lor X = s^{25}(0) )</td>
</tr>
</tbody>
</table>
Which Logic?

- We have already argued the convenience of representing the problem in logic, but

  - which logic?
    - propositional
    - predicate calculus (first order)
    - higher-order logics
    - modal logics
    - \( \lambda \)-calculus, ...

  - which reasoning procedure?
    - natural deduction, classical methods
    - resolution
    - Prawitz/Bibel, tableaux
    - bottom-up fixpoint
    - rewriting
    - narrowing, ...
Issues

• We try to maximize expressive power.
• But one of the main issues is whether we have an effective reasoning procedure.
• It is important to understand the underlying properties and the theoretical limits!
• Example: propositions vs. first-order formulas.
  ◦ Propositional logic:
    
    "spot is a dog" \( p \)
    "dogs have tail" \( q \)

    but how can we conclude that Spot has a tail?

  ◦ Predicate logic extends the expressive power of propositional logic:
    
    \[ \text{dog}(\text{spot}) \]
    \[ \forall X \text{dog}(X) \rightarrow \text{has\_tail}(X) \]

    now, using deduction we can conclude:

    \[ \text{has\_tail}(\text{spot}) \]
Comparison of Logics (I)

- Propositional logic:
  
  “spot is a dog” $p$
  
  + decidability/completeness
  
  - limited expressive power
  
  + practical deduction mechanism

  $\rightarrow$ circuit design, “answer set” programming, ...

- Predicate logic: (first order)
  
  “spot is a dog” $\text{dog}(\text{spot})$
  
  +/- decidability/completeness
  
  +/- good expressive power
  
  + practical deduction mechanism (e.g., SLD-resolution)

  $\rightarrow$ classical logic programming!
Comparison of Logics (II)

- Higher-order predicate logic:
  
  "There is a relationship for spot" \(X(spot)\)
  - decidability/completeness
  + good expressive power
  – practical deduction mechanism

  But interesting subsets \(\rightarrow\) HO logic programming, functional-logic prog., ...

- Other logics: decidability? Expressive power? Practical deduction mechanism?
  
  Often (very useful) variants of previous ones:
  
  ◊ Predicate logic + constraints (in place of unification)
    \(\rightarrow\) constraint programming!
  ◊ Propositional temporal logic, etc.

- Interesting case: \(\lambda\)-calculus
  
  + similar to predicate logic in results, allows higher order
  - does not support predicates (relations), only functions

  \(\rightarrow\) functional programming!
Generating squares by SLD-Resolution – Logic Programming (I)

- We code the problem as definite (Horn) clauses:
  \[
  \begin{align*}
  &\text{nat}(0) \\
  &\neg\text{nat}(X) \lor \text{nat}(s(X)) \\
  &\neg\text{nat}(X) \lor \text{add}(0, X, X) \\
  &\neg\text{add}(X, Y, Z) \lor \text{add}(s(X), Y, s(Z)) \\
  &\neg\text{nat}(X) \lor \text{mult}(0, X, 0) \\
  &\neg\text{mult}(X, Y, W) \lor \neg\text{add}(W, Y, Z) \lor \text{mult}(s(X), Y, Z) \\
  &\neg\text{nat}(X) \lor \neg\text{nat}(Y) \lor \neg\text{mult}(X, X, Y) \lor \text{nat\_square}(X, Y)
  \end{align*}
  \]

- **Query:** \(\text{nat}(s(0)) \, ?\)

- In order to refute: \(\neg\text{nat}(s(0))\)

- **Resolution:**
  \(\neg\text{nat}(s(0)) \text{ with } \neg\text{nat}(X) \lor \text{nat}(s(X)) \text{ gives } \neg\text{nat}(0)\)

- **Answer:** \((yes)\)
Generating squares by SLD-Resolution – Logic Programming (II)

\[ \text{nat}(0) \]
\[ \neg \text{nat}(X) \lor \text{nat}(s(X)) \]
\[ \neg \text{nat}(X) \lor \text{add}(0, X, X) \]
\[ \neg \text{add}(X, Y, Z) \lor \text{add}(s(X), Y, s(Z)) \]
\[ \neg \text{nat}(X) \lor \text{mult}(0, X, 0) \]
\[ \neg \text{mult}(X, Y, W) \lor \neg \text{add}(W, Y, Z) \lor \text{mult}(s(X), Y, Z) \]
\[ \neg \text{nat}(X) \lor \neg \text{nat}(Y) \lor \neg \text{mult}(X, X, Y) \lor \neg \text{square}(X, Y) \]

- **Query:** \( \exists X \exists Y \; \text{add}(X, Y, s(0)) \) ?

- **In order to refute:** \( \neg \text{add}(X, Y, s(0)) \)

- **Resolution:**
  - \( \neg \text{add}(X, Y, s(0)) \) with \( \neg \text{nat}(X) \lor \text{add}(0, X, X) \) gives \( \neg \text{nat}(s(0)) \)
  - \( \neg \text{nat}(s(0)) \) solved as before

- **Answer:** \( X = 0, Y = s(0) \)

- **Alternative:**
  - \( \neg \text{add}(X, Y, s(0)) \) with \( \neg \text{add}(X, Y, Z) \lor \text{add}(s(X), Y, s(Z)) \) gives \( \neg \text{add}(X, Y, 0) \)
Generating Squares in a Practical Logic Programming System (I)

:- module(_,_,['bf/bfall']).

nat(0).
nat(s(X)) :- nat(X).

le(0,X) :- nat(X).
le(s(X),s(Y)) :- le(X,Y).

add(0,Y,Y) :- nat(Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,Y,0) :- nat(Y).
mult(s(X),Y,Z) :- add(W,Y,Z), mult(X,Y,W).

nat_square(X,Y) :- nat(X), nat(Y), mult(X,X,Y).

output(X) :- nat(Y), le(Y,s(s(s(s(s(0)))))), nat_square(Y,X).
<table>
<thead>
<tr>
<th>Query</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>`- nat(s(0)).</td>
<td>yes</td>
</tr>
<tr>
<td>`- add(s(0), s(s(0)), X).</td>
<td>X = s(s(s(0)))</td>
</tr>
<tr>
<td>`- add(s(0), X, s(s(s(0)))).</td>
<td>X = s(s(s(0)))</td>
</tr>
<tr>
<td>`- nat(X).</td>
<td>X = 0 ; X = s(0) ; X = s(s(0)) ; ...</td>
</tr>
<tr>
<td>`- add(X, Y, s(0)).</td>
<td>(X = 0 , Y=s(0)) ; (X = s(0) , Y = 0)</td>
</tr>
<tr>
<td>`- nat_square(s(s(0)), X).</td>
<td>X = s(s(s(s(0))))</td>
</tr>
<tr>
<td>`- nat_square(X, s(s(s(s(0))))).</td>
<td>X = s(s(0))</td>
</tr>
<tr>
<td>`- nat_square(X, Y).</td>
<td>(X = 0 , Y=0) ; (X = s(0) , Y=s(0)) ; (X = s(s(0)) , Y=s(s(s(s(0)))))) ; ...</td>
</tr>
<tr>
<td>`- output(X).</td>
<td>X = 0 ; X = s(0) ; X = s(s(s(s(0)))) ; ...</td>
</tr>
</tbody>
</table>
father_of(john, peter)
father_of(john, mary)
father_of(peter, michael)
mother_of(mary, david)

∀X∀Y(∃Z(father_of(X, Z) ∧ father_of(Z, Y)) → grandfather_of(X, Y))
∀X∀Y(∃Z(father_of(X, Z) ∧ mother_of(Z, Y)) → grandfather_of(X, Y))

father_of(john, peter).
father_of(john, mary).
father_of(peter, michael).
mother_of(mary, david).

grandfather_of(L,M) :- father_of(L,K),
                   father_of(K,M).

grandfather_of(X,Y) :- father_of(X,Z),
                   mother_of(Z,Y).

• How can grandmother_of/2 be represented?

• What does grandfather_of(X, david) mean? And grandfather_of(john, X)?
Introductory example (II) - Testing membership in lists

- Declarative view:
  - Suppose there is a functor \( f/2 \) such that \( f(H, T) \) represents a list with head \( H \) and tail \( T \).
  - Membership definition: \( X \in L \leftrightarrow \begin{cases} X \text{ is the head of } L \\ \text{or } X \text{ is member of the tail of } L \end{cases} \)
  - Using logic:
    \[
    \forall X \forall L (\exists T (L = f(X, T) \rightarrow \text{member}(X, L))) \\
    \forall X \forall L (\exists Z \exists T (L = f(Z, T) \land \text{member}(X, T) \rightarrow \text{member}(X, L)))
    \]
  - Using Prolog:
    
    \[
    \text{member}(X, f(X, T)). \\
    \text{member}(X, f(Z, T)) :- \text{member}(X,T).
    \]

- Procedural view (but for checking membership only!):
  - Traverse the list comparing each element until \( X \) is found or list is finished

    /* Testing array membership in C */
    int member(int x, int list[LISTSIZE]) {
      for (int i = 0; i < LISTSIZE; i++)
        if (x == list[i]) return TRUE;
      return FALSE;
    }
A (very brief) History of Logic Programming (I)

- **60’s**
  - Greene: programming as problem solving.
  - Robinson: (linear) resolution.

- **70’s**
  - Kowalski: procedural interpretation of Horn clause logic. Read: 
    \[ A \text{ if } B_1 \text{ and } B_2 \text{ and } \cdots \text{ and } B_n \text{ as:} \]
    to solve (execute) \( A \), solve (execute) \( B_1 \) and \( B_2 \) and,..., \( B_n \)
  
    \[
    \text{Algorithm = logic + control.}
    \]
  - Colmerauer: specialized theorem prover (Fortran) embedding the procedural interpretation: Prolog (Programmation et Logique).
  - In the U.S.: “next-generation AI languages” of the time (i.e. planner) seen as inefficient and difficult to control.
A (very brief) History of Logic Programming (II)

- **Late 80’s, 90’s**
  - Major research in the basic paradigms and advanced implementation techniques: Japan (Fifth Generation Project), US (MCC), Europe (ECRC, ESPRIT projects, leading to the “framework programs”).
  - Numerous commercial Prolog implementations, programming books, and a *de facto* standard, the Edinburgh Prolog family.
  - Leading in 1995 to The ISO Prolog standard.
  - Parallel and concurrent logic programming systems.
  - **CLP** – Constraint Logic Programming: Major extension – opened many new applications areas.
    * Commercial CLP systems with fielded applications.
    * Concurrent constraint programming systems.

- **2000-...**
  - Many other extensions: full higher order, inclusion of functional programming, types, verification, partial evaluation, concurrency, distribution, objects, ...
  - Highly optimizing compilers, environments, automatic parallelism, automatic debugging.
A (very brief) History of Logic Programming (III)

- Many applications:
  - Natural language processing
  - Scheduling/Optimization problems
  - AI related problems
  - (Multi) agent systems programming
  - Program analyzers
  - ...

- Some examples:
  - The first C++ compiler was written in Prolog.
  - The java abstract machine is specified in Prolog.
  - The IBM Watson System (2011) has important parts written in Prolog.
    - https://www.youtube.com/watch?v=P18EdAKuClU
  - ...
  - ...