Some administrative stuff

- Class web page:
  - http://www.cs.ucsd.edu/classes/sp08/cse291e/
- Web page will have a section at the top called "For next class"
  - which states what you should read for next class
  - by default, from now on, assume that you should write a review (if you're taking the class for >= 2 credits)

Some administrative stuff

- Paper reviews
  - short sentence or two summarizing the paper
  - plus points (points in favor, things you liked, things you found interesting)
  - negative points (points against, things you didn't like, things you think can be improved)
- Email them directly to me, at the latest half-an-hour before class: lerner@cs.ucsd.edu
- There will be on the order of about 5 paper reviews

De Millo, Lipton, and Perlis

Math	Software verification
Proofs in math are a social process: intuitive, evolving, peer-reviewed	Software verification is the rigorous application of inference rules – no social process!
Important theorems in math are simple	Verification conditions are complicated, ugly and not humanly readable
Proofs in math are beautiful, full of ideas	Verification proofs are long, tedious and boring

De Millo, Lipton, and Perlis

- "Unfortunately, there is a wealth of evidence that fully automated verifying systems are out of the question. Even the strong adherents of program verification do no take seriously the possibility of totally automated verifiers."

Verification obstacles

- No social process
  - Acceptability of mathematical proofs depend on social processes
  - Who would review program verifications?
- Absence of continuity
  - One verification not useful for others
- Inevitability of change
  - Programs and spec change constantly
- Complexity of specification
  - Many programs are not specifiable
- Computational cost is high
  - A huge amount of formal details
- Might lead to overconfidence
  - titanic effect

But… scaling back a little

- Don't need to do full verification of correctness
  - verification of simple properties can be useful
- Don't need to verify the whole code
  - start with the most critical part
- Don't need to do it fully automatically
  - user hints and annotations
- Don't need to be fully precise -- false positives are ok, as long as:
  - they don't overwhelm the programmer
  - you can find bugs

Moore talk

- "Code in that language is executed to perform the computations the user cares about."
- "Executability is crucial. And remember, I do *not* mean just the theoretical burden of reducing ground terms to constants. I mean the practical burden of doing it efficiently. "

Moore talk

- ACL2 uses LISP as the language for expressing computation
- What if the code I care about is in a different language, say assemby?
  - Can model the code in LISP
  - Or "formalize the semantics operationally in your system and prove theorems about constants representing programs in the other language"
  - In other words: write an interpreter for assemby, and then prove properties about your interpreter

Grand challenges

- Automatic Invention of Lemmas and New Concepts
- How to use Examples and Counterexamples
- How to use Analogy, Learning, and Data Mining
- How to Architect an Open Verification Environment
- Parallel, Distributed and Collaborative Theorem Proving
- User Interface and Interactive Steering
- Education of the User Community -- and Their Managers
- How to Build a Verified Theorem Prover

Logics

Logics

- Standard logics
  - Propositional logic
  - First-order predicate logic
  - Higher-order predicate logic
- Non-standard logics
  - Categorical logic, Combinatory logic, Conditional logic, Constructive logic, Cumulative logic, Deontic logic, Dynamic logic, Epistemic logic, Erotetic logic, Free logic, Fuzzy logic, Infinitary logic, Intensional logic, Intuitionistic logic, Linear logic, Many-valued logic, Modal logic, Non-monotonic logic, Paraconsistent logic, Partial logic, Prohairetic logic, Quantum logic, Relevant logic, Stoic logic, Substance logic, Substructural logic, Temporal (tense) logic
  - In short: a lot!

In ATPs, logic has three purposes

- What are they?

In ATPs, logic has three purposes

- It is used to express the problem at hand
- It is used by the theorem prover for automated reasoning
- It is used to communicate with the end user of the theorem prover

This leads to three questions

- How expressive is the logic?
  - what problems can be expressed in the logic
- How automatable is the logic?
  - how much can we hope to automate reasoning in the logic
- How human-friendly is the logic?
  - for proofs and/or counter-examples
  - for interactive theorem proving
- We will look at some logics, with these three questions in mind

Propositional logic

$$\begin{array}{rcl} \phi & ::= & true \mid false \mid x \mid \\ & & \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \Rightarrow \phi \end{array}$$

- Simple and easy to understand
- Decidable, but NP complete
  - Very well studied; efficient SAT solvers
  - if you can reduce your problem to SAT …
- Drawback?

Propositional logic

$$\begin{array}{rcl} \phi & ::= & true \mid false \mid x \mid \\ & & \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \Rightarrow \phi \end{array}$$

- Simple and easy to understand
- Decidable, but NP complete
  - Very well studied; efficient SAT solvers
  - if you can reduce your problem to SAT …
- Drawback
  - can only model finite domains
  - How can we fix this?

First-order logic

$$\begin{array}{rcl} t & ::= & x \mid F(t, \ldots, t) \\ \phi & ::= & true \mid false \mid P(t, \ldots, t) \mid \\ & & \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \phi \Rightarrow \phi \mid \\ & & \forall x.\phi \mid \exists x.\phi \end{array}$$

- Example:

$$\forall x.HUMAN(x) \Rightarrow MAMMAL(x)$$
$$\forall x.HUMAN(x) \Rightarrow HUMAN(mom(x))$$
$$\exists x.mom(x) = x$$

First-order logic: sources of "infinities"

- ∀ (∃) can range over infinite sets: $\forall i \;.\; i + 1 > i$
- If ∀ (∃) ranges over a finite known set, can expand into conjunction (disjunction)
  - In a world with only 3 students (Bob, Alice, and a squid), we can expand: $\forall student \;.\; SMART(student)$
  - Into: $SMART(Bob) \wedge SMART(Alice) \wedge SMART(Squid)$
  - Assign each on of these to a propositional variable:

$$A = SMART(Bob)$$
$$B = SMART(Alice)$$
$$C = SMART(Squid)$$

  - We are now in propositional logic

First-order logic: sources of “infinities”

- ∀ (∃) can range over finite, but unbounded sets
- For example, suppose we have a world with a finite number of students
- But we don’t know how many students there are
- We can’t expand: $\forall student\ .\ SMART(student)$
- Even though the set of students is finite

First-order logic: sources of “infinities”

- Function symbols can be applied a finite but an unbounded number of times

  $mom(Alice), mom(mom(Alice)), mom(mom(mom(Alice))), \ldots$
- Is this a different source of infinity than we’ve seen so far?
- Answer: no. This infinity (unboundedness) is reflected through the infinite domains
  - In the above case, the Herbrand universe is infinite
  - May or may not

Many-sorted first-order logic

- First-order logic with many sorts (types)
- Not more expressive than first-order logic, but convenient for expressing formulas over domains that have complex structure

$$\begin{array}{l}\forall s : student\ .\ \exists c : course\ .\ TAKES(s,c)\\ \forall s : Stmt\ .\ \exists \sigma : ProgState\ .\ CAUSES\_EXPT(s,\sigma)\\ \forall s : Var, f : FieldName\ .\ select\_field(s,f) \neq null\\ \exists s : Var\ .\ select\_field(s,next) \neq null \wedge\\ \qquad select\_field(select\_field(s,next),next) = null\end{array}$$

- Types can be encoded in the untyped first-order logic

Summary of first-order logic

- Expressiveness
  - more expressive than propositional logic
  - because of infinite domains
- Automation
  - not decidable anymore
  - one of the main source of difficulty are quantifiers, not surpisingly
  - still, very well studied, and many theorem provers and theorem proving techniques available

Summary of first-order logic

- Human-friendliness
  - Intuitive for humans to understand. Also, types, if present, improve readability
  - One may choose FOL even if the problem is expressible in propositional logic, because of FOL’s notational conveniences and human friendliness.
  - For example, even if domain is finite, it may be more intuitive to write a formula using first-order quantifiers, rather than propositional logic
  - However, more expressive logics are generally less automatable: danger that a formula provable by a propositional engine will not be provable automatically when expressed using quantifiers

Limitations of first-order logic

- Let’s look at proof by induction
- To prove: $\forall i : Nat\ .\ P(i)$
- Show: $P(0)$
- And: $\forall i : Nat\ .\ P(i) \Rightarrow P(i+1)$

Let's express this as a theorem

$$(P(0) \wedge \forall i : Nat \;.\; (P(i) \Rightarrow P(i+1))) \Rightarrow (\forall i : Nat \;.\; P(i))$$

Let's express this as a theorem

$$\forall P : ??? \;.\; (P(0) \wedge \forall i : Nat \;.\; (P(i) \Rightarrow P(i+1))) \Rightarrow (\forall i : Nat \;.\; P(i))$$

Let's express this as a theorem

$$\forall P : ??? \;.\; (P(0) \wedge \forall i : Nat \;.\; (P(i) \Rightarrow P(i+1))) \Rightarrow (\forall i : Nat \;.\; P(i))$$

- P needs to range over predicates…
- In FOL, a variable was a term
  - could only be passed to function symbols, or to predicate symbols.
- A variable could not be applied as a predicate

Let's express this as a theorem

$$\forall P : ??? \;.\; (P(0) \wedge \forall i : Nat \;.\; (P(i) \Rightarrow P(i+1))) \Rightarrow (\forall i : Nat \;.\; P(i))$$

- Could use encoding tricks, for example:
  - use app(P, x) to represent P(x)
- However, this becomes cumbersome, and it also doesn't provide all the expressiveness that quantifying over predicates does

Higher-order logic

- In first-order logic
  - quantifiers range over ground terms
  - predicates only take terms as arguments
- Higher-order logic also allows
  - quantification over predicates
  - predicates that take predicates as arguments

Expressiveness of higher-order logic

- First-order logic + transitive closure
  - strictly more expressive than first-order logic
  - expressible in higher-order logic
- Suppose you have a linked list L of finite, but unbounded size. You want to express the theorem "5 is in L"

$$\exists n \in next_{tc}(L) \;.\; data(n) = 5$$

- Intuitively, $next_{tc}$ computes the transitive closure of the next relation
  - $next_{tc}(L)$ returns the set of nodes reachable from L

Expressiveness of higher-order logic

- Let's try doing this in first-order logic

$$\forall L \;.\; L = null \Rightarrow \neg five\_in\_list(L)$$
$$\forall L \;.\; L \neq null \Rightarrow$$
$$(five\_in\_list(L) \iff$$
$$data(L) = 5 \vee five\_in\_list(next(L)))$$

- This is a correct and complete encoding for finite-length lists

Expressiveness of higher-order logic

- However, for infinite-length lists (for example cyclic lists), this encoding does not work
- Consider a cyclic linked list with one node that points to itself, and suppose the data for that node is 0.
  - We have $next(L) = L$ and $L \neq null$
  - Since $data(L) = 0$, the second axiom from the previous slide gives us:
    $$five\_in\_list(L) \iff five\_in\_list(L)$$
  - Thus we gain no information about $five\_in\_list(L)$

Expressiveness of higher-order logic

- The transitive closure formulation works, even on infinite lists
- Transitive closure is also very convenient for expressing heap properties
  - for example: linked lists L_1 and L_2 are disjoint

Let's go back to induction

- Anxious student: Does this mean we can't use induction in first-order logic?
- No, we can still use induction, using an induciton inference rule
- We just can't reason about induction in first-order logic

Let's go back to induction

- Anxious student: Does this mean we can't use induction in first-order logic?

Summary of higher-order logics

- More expressive than first-order logic
- Even less automatable than first-order logic
  - No theorem prover that I know of handles higher-order logic fully automatically

Summary of standard logics

- prop $\rightarrow$ FOL $\rightarrow$ HOL
- less expressive to more expressive
- more automatable to less automatable
- sometimes may choose more expressive logic, even if not strictly required
  - LCF, HOL and Isabelle theorem prover

Non-standard logics

- Non-classical logics impose restrictions or variations on classical logics in order to get certain properties
- Linear logic: allowed to use each assumption only once
- Constructive logic: not allowed to use law of the excluded middle ($\alpha \wedge \neg \alpha$)
- Relevance logic: proof of a $\rightarrow$ b only true if a really is relevant to making b true
- Temporal logic: adds notion of time

Non-standard logics

- Usually less expressive
  - but can express certain concepts more succinctly (eg: temporal logic)
- Generally easier to reason about
  - sometimes full decision procedures exists
- Because of their non-standard nature, may be harder to understand for a human

For the rest of the course...

- For the rest of this course, we will mostly deal with standard logics
- and within standard logics, we will deal mostly with FOL
  - it is a good starting point, since it provides a good balance between expressiveness, automation, and human-friendliness
- We will talk about constructive logic, since they have important applications in the area of program verification

Next lecture

- Show you example of encoding problems in first order logic (in preparation for the mini-project)
- Start looking at two examples of uses of theorem provers:
  - ESC/Java
  - Rhodium