Some administrative stuff

- Reminder: Mini-project is up on the web page
- It's due April 19
- Get started early!
- Project web page is up too
- Take a look at the milestones

Where we're at

- Last time we explored the three standard logics, and compared them along three dimensions
  - expressiveness
  - automation
  - human-friendliness
- Today we'll look in more detail at FOL
  - we will explore how to encode computations in FOL in an axiomatic way
  - we will use Simplify as the running theorem prover for today's lecture

Encoding functions in FOL

- Suppose I have a pure function f. How can this function be encoded in FOL?

$$\forall x. f(x) = \text{body of } f$$

- For example: `int f(int x) { return x + 1 }`

$$\forall x. f(x) = x + 1$$

- In Simplify syntax:

```
(BG_PUSH (FORALL (x) (EQ (f x) (+ x 1))))
```

Another example

- Factorial function in C:

```
int fact(int n) {
   if (n <= 0)
      return 1;
   else
      return n * fact(n-1);
}
```

- In Simplify:

```
(BG_PUSH (FORALL (n)
   (EQ (fact n)
       ???)))
```

Ideally, would like an IF stmt

- Factorial function in C:

```
int fact(int n) {
   if (n <= 0)
      return 1;
   else
      return n * fact(n-1);
}
```

- In Simplify:

```
(BG_PUSH (FORALL (n)
   (EQ (fact n)
       (IF (<= n 0)
           1
           (* n (fact (- n 1)))))))
```

No built-in IF in Simplify

- Simplify has (IMPLIES P_1 P_2), which stands for $P_1 \Rightarrow P_2$
- However, it does not have the IF statement we want
- How do we encode factorial?

Solution 1: write your own IF

- Write your own IF function – is this even possible?
- IF is a function symbol, so it takes terms as arguments. This means that (IF (<= 0 1) A B) is in fact mal-formed.
- Can get around this by thinking of predicates as being function symbols that return special terms |@true| or |@false|.
  - Simplify does this in fact, but only for user defined predicates

Solution 1: write your own IF

```
(BG_PUSH
   (FORALL (P A B)
      (IMPLIES (EQ P |@true|)
               (EQ (IF P A B) A))))
(BG_PUSH
   (FORALL (P A B)
      (IMPLIES (NEQ P |@true|)
               (EQ (IF P A B) B))))

(DEFPRED (leq a b) (<= a b))

(BG_PUSH (FORALL (n)
   (EQ (fact n)
       (IF (leq n 0)
           1
           (* n (fact (- n 1)))))))
```

Solution 1: write your own IF

- The IF function symbol makes it easy to write functions
- However, in Simplify it doesn't work for primitive predicates
  - although there is no theoretical reason it shouldn't
- Also, the extra indirection of using predicates like leq in the previous slide, instead of <=, can confuse Simplify

Solution 2: break the function cases

- Break the function into cases, and pull the case analysis to the top level

```
(BG_PUSH
   (FORALL (n) (IMPLIES (<= n 0)
                        (EQ (fact n) 1))))

(BG_PUSH
   (FORALL (n) (IMPLIES (NOT (<= n 0))
                        (EQ (fact n)
                            (* n (fact (- n 1)))))))
```

Solution 2: break the function cases

- Break the function into cases, and pull the case analysis to the top level

```
(BG_PUSH
   (FORALL (n) (IMPLIES (<= n 0)
                        (EQ (fact n) 1))))

(BG_PUSH
   (FORALL (n) (IMPLIES (NOT (<= n 0))
                        (EQ (fact n)
                            (* n (fact (- n 1)))))))
```

- Let's try it out in Simplify!

Solution 2: break the function cases

- In general, the cases may overlap
  - Simple example: can add the following to the previous definition of factorial

```
(BG_PUSH (EQ (fact 0) 1))))
```

  - A slightly more complicated example:

```
(BG_PUSH
   (FORALL (a b) (IMPLIES (<= a b)
                          (EQ (min a b) a))))
(BG_PUSH
   (FORALL (a b) (IMPLIES (>= a b)
                          (EQ (min a b) b))))
```

Overlapping: good or bad?

- Some functions may be easier to define if we allow overlap
- Some of the overlap provides redundancies that may actually help the theorem prover

```
(BG_PUSH (EQ (fact 0) 1))))
```

  - The above axiom is *completely* redundant, since it doesn't add anything to the definition of factorial
  - However, it may help the theorem prover

Overlapping: good or bad?

- What happens if the overlapping definitions don't agree?
- For example:

```
(BG_PUSH
   (FORALL (a b) (IMPLIES (<= a b)
                          (EQ (f a b) (+ a b)))))
(BG_PUSH
   (FORALL (a b) (IMPLIES (>= a b)
                          (EQ (f a b) (* a b)))))
```

Overlapping: good or bad?

- What happens if the overlapping definitions don't agree?
- For example:

```
(BG_PUSH
   (FORALL (a b) (IMPLIES (<= a b)
                          (EQ (f a b) (+ a b)))))
(BG_PUSH
   (FORALL (a b) (IMPLIES (>= a b)
                          (EQ (f a b) (* a b)))))
```

- Can prove false!
  - which means can prove anything
  - let's try it out

Inconsistencies in axioms

- In Simplify, there are no safeguards against user-defined inconsistencies
  - What can we do about this?
- Other theorem provers, for example ACL2, have safeguards
  - In ACL2, functions are defined in LISP. Thus, there are no ill-defined functions, such as f in the previous slide
  - Theorems are proved about LISP code
  - The LISP code is therefore a model for the formulas you are proving

Data structures

- Our functions so far have only operated on integers
- Function symbols can be used to encode more complicated data structures
  - function symbols for constructors
  - function symbols for accessors (fields)

Example: pairs

- Suppose we want to encode pairs:
- We will have:
  - a function symbol "pair" to construct a pair
  - a function symbol "first" to extract the first element
  - a function symbol "second" to extract the second element
- Axioms:

Example: pairs

- Suppose we want to encode pairs:
- We will have:
  - a function symbol “pair” to construct a pair
  - a function symbol “first” to extract the first element
  - a function symbol “second” to extract the second element
- Axioms:

```
(BG_PUSH
   (FORALL (a b) (EQ (first (pair a b)) a)))
(BG_PUSH
   (FORALL (a b) (EQ (second (pair a b)) b)))
```

Example: pairs

- Note that the above two axioms can be seen as definitions of “first” and “second”
- The following more explicit version of the axioms makes this clear:

```
(BG_PUSH
   (FORALL (X a b)
      (IMPLIES (EQ X (pair a b))
               (EQ (first X) a))

(BG_PUSH
   (FORALL (X a b)
      (IMPLIES (EQ X (pair a b))
               (EQ (second X) b))
```

Example: pairs

- Do we need an axiom defining “pair”?

```
(BG_PUSH
   (FORALL (a b ...) (EQ (pair a b) ...)))
```

Example: pairs

- Do we need an axiom defining “pair”?

```
(BG_PUSH
   (FORALL (a b ...) (EQ (pair a b) ...)))
```

- No, the “pair” constructor really only has meaning with respect to the “first” and “second” selectors
- So the two axioms for first and second completely specify the behavior of pairs

A simple AST example

- Suppose we want to represent call in C
  - For now, let’s assume only one parameter
  - x = f(y)
- We’ll use the two constructors, Assgn and FunCall:
  - the term (Assgn LHS RHS) represents LHS = RHS
  - the term (FunCall f y) represents f(y)
- So the assignment would be:
  - (Assgn x (FunCall f y))

Some subtleties

- In (Assgn x (FunCall f y)), what are x,f, and y? Are they variables? Constants?
  - recall: a constant is a nullary function symbol
- If x, f, and y are variables, then they store the strings that represent the statements we are encoding
  - for example, if x = “result”, f = “fact” and y = “myvar”, then (Assgn x (FunCall f y)) represents the C statement result = fact(myvar)
- If x, f, and y are constants, then they actually *are* the strings in the statement
  - in this case, if we were to follow Simplify syntax, we should write then as (x), (f) and (y)
- Little difference between a constant and a global variable

A simple AST example

- Suppose now that we wanted to have function calls with 0 or more parameters
  - How would we do this?

A simple AST example

- Suppose now that we wanted to have function calls with 0 or more parameters
  - How would we do this?
- We need a data structure that can store many elements
  - could do this with a linked list
  - could also do it with an array
- Let's do it with arrays, since Simplify has support for them

Maps in Simplify

- Simplify has built-in support for maps (arrays):
  - (select map idx) returns the value at index idx in map
    - think: map[idx]
  - (store map idx value) returns the input map, but with idx updated to the given value
    - think: map' = copy map; map'[idx] = value; return map'
  - (mapFill value) returns a map that has all indices set to value
- Simplify has background axioms defining these function symbols

Simplify axioms about maps

```
(FORALL (map idx value)
   (EQ (select (store map idx value) idx) value))

(FORALL (idx1 idx2 map value)
   (IMPLIES (NEQ idx1 idx2)
            (EQ (select (store map idx1 value) idx2)
                (select map idx2))))

(FORALL (value idx)
   (EQ (select (mapFill value) idx) value)
```

Back to function calls in C

- The second parameter to FunCall is now a map, rather than a string
  - (FunCall f map)
- x := f(a,b)

```
(Assgn x (FunCall f
      (store (store (mapFill uninit) 0 a) 1 b)))
```

Encoding predicates

- In the simple case:

```
(DEFPRED (Pred A B C))
(BG_PUSH (FORALL (A B C) (IFF (Pred A B C)
                              body)))
```

- As before, we can split the definition over multiple possibly overlapping cases

Special case

- When there is only one case, as in the previous slide, Simplify allows the body of the predicate to be given with the declaration:

```
(DEFPRED (Pred A B C ) definition))
```

- Logically equivalent, but…
  - the above declaration causes (Pred A B C) to immediately be rewritten to its body
  - whereas the declaration on the previous slide gets expanded using quantifier instantiation heuristics

Incomplete functions or predicates

- Some functions may not be defined on all of their inputs. Such functions are said to be partial
- Easy to model in an axiomatic setting such as Simplify
  - incomplete axioms ⇒ incomplete functions
  - for example, here is fact, only defined on naturals:

```
(BG_PUSH (EQ (fact 0) 1))
(BG_PUSH
   (FORALL (n) (IMPLIES (> n 1)
                        (EQ (fact n)
                            (* n (fact (- n 1)))))))
```

Two examples of how theorem provers are used

overview today, more details on Tuesday

ESC/Java: Key to making it work

- Give up on total correctness
- Require user annotations
- Give up on completeness (may report bugs that don't exists)
- Give up on soundness (may miss some bugs it was supposed to find)
- Most of the unsoundness arises from loops

Rhodium: Key to making it work

- Use a domain specific language to restrict the programs you have to reason about
- Separate "profitability heuristics" from "correctness"
- Split the proof into two parts:
  - the first part is optimization independent, and is done by hand once and for all
  - the other part is optimization dependent, and this is what the theorem prover discharges