Where we’re at

• We explored the three standard logics, and compared them along three dimensions
  – expressiveness
  – automation
  – human-friendliness
• Next 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

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:
  \( \text{int } f(\text{int } x) \{ \text{return } x + 1 \} \)
  \( \forall x. f(x) = x + 1 \)
• In Simplify syntax:
  \( \text{(BG_PUSH (FORALL (x) (EQ (f x) (+ x 1)))}) \)

Another example

• Factorial function in C:
  ```c
  int fact(int n) {
    if (n <= 0)
      return 1;
    else
      return n * fact(n-1);
  }
  ```
• In Simplify:
  \( \text{(BG_PUSH (FORALL (n) (EQ (fact n) ???)))} \)

Ideally, would like an IF stmt

• Factorial function in C:
  ```c
  int fact(int n) {
    if (n <= 0)
      return 1;
    else
      return n * fact(n-1);
  }
  ```
• In Simplify:
  \( \text{(BG_PUSH (FORALL (n) (EQ (fact n) ???)))} \)

No built-in IF in Simplify

• Simplify has \( (\text{IMPLIES } P_1 \ P_2) \), 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 \( (\text{IF } (\leq 0 \ 1) \ A \ B) \) is in fact mal-formed.
• Can get around this by thinking of predicates as being function symbols that return special terms \( \text{@true} \) or \( \text{@false} \).
  – Simplify does this in fact, but only for user defined predicates
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

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
  – 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:

\[
\text{BG_PUSH}
\begin{align*}
\forall (a \ b) & \exists (a \ b) \\
\left( \forall (a \ b) \ (\text{IMPLIES} \ (\leq a \ b) \ (\text{EQ} \ (f \ a \ b) \ (+ \ a \ b))) \right) \\
\text{BG_PUSH}
\end{align*}
\]

Overlapping: good or bad?
• What happens if the overlapping definitions don’t agree?
• For example:

\[
\text{BG_PUSH}
\begin{align*}
\forall (a \ b) & \exists (a \ b) \\
\left( \forall (a \ b) \ (\text{IMPLIES} \ (\leq a \ b) \ (\text{EQ} \ (f \ a \ b) \ (+ \ a \ b))) \right) \\
\end{align*}
\]

• 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 and Coq, have safeguards
  – In Coq/ACL2, functions are defined in CIC/LISP.
    Thus, there are no ill-defined functions, such as \( f \) in the previous slide
  – Theorems are proved about CIC/LISP code
  – The CIC/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:

\[
\forall x \ y \ (\exists (x \ y) \ (\text{first} \ (\text{pair} \ x \ y) \ x))
\]

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:

\[
\text{BG_PUSH}
\begin{align*}
\forall (a \ b) & \exists (a \ b) \\
\left( \forall (a \ b) \ (\text{EQ} \ (\text{first} \ (\text{pair} \ a \ b)) \ a) \right) \\
\text{BG_PUSH}
\end{align*}
\]

\[
\text{BG_PUSH}
\begin{align*}
\forall (a \ b) & \exists (a \ b) \\
\left( \forall (a \ b) \ (\text{EQ} \ (\text{second} \ (\text{pair} \ a \ b)) \ b) \right) \\
\end{align*}
\]
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:

\[
\begin{align*}
&(\forall x \ a \ b) \ (\text{first}(x) \ a) \\
&(\forall x \ a \ b) \ (\text{second}(x) \ b)
\end{align*}
\]

Example: pairs

• Do we need an axiom defining “pair”?

\[
(\forall a \ b \ldots) \ (\text{pair}(a \ b \ldots))
\]

Example: pairs

• 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

Some subtleties

• In \((\text{Assgn} \ x \ (\text{FunCall} f \ y))\), 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 \(x = \text{result}, f = \text{fact}\) and \(y = \text{myvar}\), then \((\text{Assgn} \ x \ (\text{FunCall} f \ y))\) represents the C statement \(\text{result} = \text{fact}(\text{myvar})\)
• 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 \(\langle x \rangle, \langle f \rangle\) and \(\langle y \rangle\)
• Little difference between a constant and a global variable

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 \((\text{Assgn} \ LHS \ RHS)\) represents \(LHS = RHS\)
  – the term \((\text{FunCall} f \ y)\) represents \(f(y)\)
• So the assignment would be:
  – \((\text{Assgn} \ x \ (\text{FunCall} f \ y))\)
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

\[(\forall (\text{map} \ idx \ \text{value}) \ (\text{eq} (\text{select} (\text{store} \ \text{map} \ \text{idx} \ \text{value}) \ \text{idx}) \ \text{value}))\]
\[(\forall (\text{idx}1 \ \text{idx}2 \ \text{map} \ \text{value}) \ (\text{implies} (\text{neq} \ \text{idx}1 \ \text{idx}2) \ (\text{eq} (\text{select} (\text{store} \ \text{map} \ \text{idx1} \ \text{value}) \ \text{idx2}) \ (\text{select} \ \text{map} \ \text{idx2}))))\]
\[(\forall (\text{value} \ \text{idx}) \ (\text{eq} (\text{select} (\text{mapFill} \ \text{value}) \ \text{idx}) \ \text{value}))\]
- Let’s try some examples out in Simplify

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)
  - (Assign x (FunCall f (store (store (mapFill uninit) 0 a) 1 b)))

Encoding predicates

- In the simple case:
  - (DEFPRED (Pred A B C))
  - (SEQ_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:

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