Ling130 – Lecture Notes for 2/12/10

❖ Goals for today

- Start over with λ-calculus
- Abstraction and Application Rules
- Type Derivations
- More interesting types
- Interpreting typed expressions
- Problem Set 2

❖ What's the deal with λ-calculus?

- It's used to define functions and we can think of everything in FOL as a function
  - Consider *∀x(sleepy(x))*. This could mean "everyone is sleepy" and it is either true or false (it has type *t*).
  - Now consider *sleepy(x)*. This is like a "holey" preposition. It doesn't really mean anything unless we know what *x* stands for.
  - The FOL quantifiers tell us what the free variable could stand for, but what if we just want to define the preposition?
  - *λx[sleepy(x)]* = a function that takes in a value for *x* and outputs a truth value.

❖ The Syntax of λ-Expressions

- Variables – *x, y, z*, etc.
  - Just like the variables we have in FOL
  - Can be bound or free
- Abstractions – *λV.E*
  - *V* is a bound variable over the body *E*.

- Example: $\lambda x[P(x)]$ means that the variable x is bound in P.
- This is how functions are defined in λ-calculus.
- What is the type of a λ-abstraction?
  - The type is composed by the type of the argument variable V and the type of the expression E.
  - $x{:}e,\ \phi{:}t\ \vdash\ \lambda x[\phi]{:}e \rightarrow t$

➢ Applications – (E_1E_2)
- Function application where we apply E_1 to E_2.
- Example: $\lambda x[sleepy(x)](j)$
  - The argument j must be the same type as x.

➢ Basic Normal Form
- $E ::= V \mid \lambda V.E \mid (E_1E_2)$
- $V ::= x \mid y \mid z \mid \ldots$

➢ Note that expressions in λ-calculus can also have quantifiers in them.
- Example: $\lambda x[\exists y[loves(y,x)]](m)$ is wellformed.

❖ Conversion and Reduction Rules

➢ There are 3 rules you can use in λ-calculus, but we will be mostly interested in β-conversion.
- *α-conversion*: Any abstraction $\lambda V.E$ can be converted to $\lambda V.E[V'/V]$ iff $[V'/V]$ in E is valid.
  - Read $[V'/V]$ as "replace V with V'".
  - This just means you can change all of the bound occurrences of V in E with a different variable.
- *η-conversion*: Any abstraction $(\lambda V.E)$ where V has no free occurrences in E can be converted to E.
- *β-conversion*: Any application $(\lambda V.E_1)E_2$ can be converted to $E_1[E_2/V]$ iff $[E_2/V]$ in E_1 is valid.

- This is the basic rule of function application for λ-calculus and we'll use it a lot!
- Example: *λx[sleepy(x)](j) ▷ sleepy(j)*
- Example: *λx[∃y[loves(y,x)]](m) ▷ ∃y[loves(y,m)*

❖ **Back to Types**

- We've already talked about types a little bit for λ-calculus.
- Let's look at another example: "John loves Mary."
  - λyλx[loves(x,y)] ≡ Love':e→e→t
  - Type Tree

- It's pretty easy to see how the basic types work in λ-calculus, but what about other elements of language such as adverbs and conjunctions?
  - We can figure out the types of different categories by figuring out what they input and output.
  - Example: "John walks slowly."
    - What we already know: *j:e, λx[walks(x)]:e→t*
    - Type Tree for *John walks.*

- We still need to get type *t* at the top of the tree. Applying $\lambda x[walks(x)](j)$ does that, so we don't want the adverb to change the type of the verb.
- So, adverbs take in the type of the verb phrase *e→t* and output the same type: *(e→t)→(e→t)*
- Type Tree for *John walks slowly.*

➢ Let's figure out the types of different syntactic categories:

- Clausal Verb: *Jess believes that <u>Elana is the cutest baby</u>.*
  - What is the type of the clause? *t*
  - Again, we want to end up with a *t* at the end, so the type of *believes* is *t→(e→t)*.
- Auxiliary Verb: *Jess may be wrong.* (but probably not)
  - This is similar to adverbs.
  - *be wrong* is type *e→t*; *may* is type *(e→t)→(e→t)*
- Negations: *It is not the case that Jess is wrong*.
  - Negations take in sentences and output sentences so they have type *t→t*.
- Conjunctions (of sentences)
  - t→(t→t)
  - Conjunctions take in things of the same type and then output that same type. We will return to conjunctions later.

- The ability to talk about all of these (complex) categories is what makes type theory (and therefore λ-calculus) much more powerful than FOL.
  - Type assignments that are more than just *e* are called <u>higher-order</u>. Predicates like *e→t* can be arguments as well as individuals.
  - Example: *John didn't willingly love Mary.*

Constituent	**Translation**	**Type**
love	*λyλx[loves(x,y)] = Love'*	*e→(e→t)*
Mary	*m*	*e*
love Mary	*Love'(m)*	*e→t*
willingly	*Willingly'*	*(e→t)→(e→t)*
willingly love Mary	*Willingly'(Love'(m))*	*e→t*
didn't	*λPλz[¬P(z)]*	*(e→t)→(e→t)*
didn't willingly love Mary	*λPλz[¬P(z)](Willingly'(Love'(m)))*	*e→t*
John	*j*	*e*
John didn't willingly love Mary	*λz[¬(Willingly'(Love'(m)))(z)](j)* ≡ *¬(Willingly'(Love'(m)))(j)*	*t*

  - Type Tree

- The main thing to notice during this exercise was the use of the capital letter *P* in the translation for *didn't*.
  - As always, capital letters mean predicates, so this is an example of a predicate being used in an abstraction.
  - Look again at the type for *didn't:* $(e \rightarrow t) \rightarrow (e \rightarrow t)$
    - The type tells us that *didn't* requires a predicate and an entity, and that's just what the corresponding λ-expressions says: $\lambda P \lambda z[\neg P(z)]$
  - We'll use predicates in λ-expressions more when we get to the next chapter.

Interpreting Typed Expressions

- Typed expressions that are just FOL can use the same interpretation as FOL does, described in section 3.3.
- Obviously, given what we just saw, that would be an incomplete interpretation for all typed expressions though!
- Our plan is to try to stick with set theory, but modify things a bit to account for the extra power of type theory.
- Denotations we already know:
  - The denotation of an expression of type *e* is an entity from the domain of individuals. (written D_e)
  - The denotation of an expression of type *t* is a truth value from the domain {0,1}. (written D_t)
  - The denotation of an expression of type $e \rightarrow t$ is a set of entities.
  - The denotation of an expression of type $a \rightarrow t$ where *a* is a type is a set of objects of type *a*.

- Unfortunately, we need to take a step back to talk about some more "mathy" stuff before we can figure out what all types denote.
  - Relations: a set of ordered pairs mapping elements in the domain to elements in the range.
    - Ex. {<a,2>,<b,2>, <b,4>, <b,5>, <d,5>}
  - Functions: a special kind of relation; A relation is a function iff every element in the domain is assigned one and only one value in the range.
    - Ex. {<a,1>, <b,2>, <c,2>, <d,4>, <e,4>}
    - Functions are unambiguous and fully specified.
  - Characteristic Function: a special kind of function that directly defines a set; it maps the elements of its domain onto 1 if that element is in the set or 0 if it is not.
- Back to the denotations of types
  - Recall that 1-place predicates denote sets and are expressions of type $e \rightarrow t$
  - Characteristic functions are functions from entities to truth values. (sounds familiar!)

- Now we can figure out the denotations of all typed expressions recursively, just as we defined the definitions of types recursively.
- Let D_τ be the denotation domain for a type τ. Then we have:
  $D_{e \to t} = D_e \mapsto D_t$
- This is basically what the denotations of typed expressions look like:
  - *A function from the domain of things denoted by their antecedent type to the range of things denoted by the consequent type*.
- The general denotation of type τ is symbolized as D_τ and is defined as:
  - a) $D_e = A$ (the set of entities)
  - b) $D_t = \{1,0\}$ (the set of truth values)
  - c) If **a** and **b** are types, then $D_{a \to b}$ is $D_a \mapsto D_b$, a set of functions from elements of type **a** to elements of type **b**.
- Figuring out the denotation of a particular type involves unpacking the denotation of its parts:
  - Ex. $D_{e \to (e \to t)} = D_e \mapsto D_{e \to t} =$
    $D_e \mapsto (D_e \mapsto D_t) = A \mapsto (A \mapsto \{1,0\})$

    (functions from entities to functions from entities to truth values)
  - Ex. $D_{(e \to t) \to (e \to t)} = D_{e \to t} \mapsto D_{e \to t} =$
    $(D_e \mapsto D_t) \mapsto (D_e \mapsto D_t) = (A \mapsto \{0,1\}) \mapsto (A \mapsto \{0,1\})$

  (functions from functions from entities to truth values to functions from entities to truth values

- **Chapter 3 Problem Set**
  - Exercise 3.1(a,c) – truth tables for propositional logic
  - Exercise 3.2(a-d) – FOL models
  - Exercise 3.3(a-c) – More FOL stuff (translations, truth conditions, evaluations)
  - Exercise 3.5(b,c) – Basic λ-calculus
  - Exercise 3.6('detest', 'or', b) – Types for expressions, building up types for sentences
  - Exercise 3.7('detest', 'or') – Denotations for typed expressions
  - <u>New Due Date: March 2</u>
- **Something to think about for next time:**
  - What happens when the types don't fit nicely together?