Handout 7: Predicate calculus expressions

Review

1. Examples:

every cat is an animal	$\forall x[\mathrm{cat}(x) \rightarrow \mathrm{animal}(x)]$
some cat is not orange	$\exists y[\mathrm{cat}(y) \wedge \neg\mathrm{orange}(y)]$
Max is an orange cat	$\lambda x[\mathrm{orange}(x) \wedge \mathrm{cat}(x)](\mathrm{Max})$

2. It is assumed that we can divide the world into things. Technically: **individuals**.

- **a.** **Variables** x, y, x_1, x_2, etc., denote individuals.
- **b.** **Names** are constants that denote individuals. E.g. `Max`, `Fido`

3. **Terms** are expressions that denote individuals.

- **a.** Variables and names
- **b.** Function applications, e.g. `mother-of`(`Fido`), `mother-of`(`mother-of`(x))
- **c.** We include a function `the` that takes a singleton set and returns its sole member: `the`(`cat`)
- **d.** Usually represented using the inverted-iota operator: $\iota x \,.\, \mathtt{cat}(x)$

4. **Clauses** are expressions that are either true or false (`boolean`).

- **a.** An n-place **predicate symbol** names a relation. A given n-tuple of individuals either belongs to the relation or not.

$$\mathtt{dog}(\mathtt{Fido})$$
$$\mathtt{chases}(\mathtt{Fido}, y)$$

- **b.** A predicate applied to arguments is an **atomic clause**.
- **c.** A 1-place relation is equivalent to a **set**. Technically, `dog` is the **characteristic function** or **indicator function** for the set of dogs.
- **d.** **Boolean operators** take truth values as input and return truth values.

$$\mathtt{dog}(\mathtt{Fido}) \wedge \mathtt{cat}(x)$$
$$\mathtt{cat}(x) \rightarrow \mathtt{animal}(x)$$

- **e.** The operators are: $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$.
- **f.** $p \rightarrow q$ is false only if p is true but q is false.

5. **Quantifiers** also form complex clauses.

   a. **Universal** $\forall x \,.\, \phi$ is true just in case ϕ is true no matter what value we assign to x.

   $$\forall x \,.\, \underbrace{\texttt{cat}(x) \rightarrow \texttt{animal}(x)}_{\phi}$$

   b. **Existential** $\exists x \,.\, \phi$ is true just in case there is at least one value for x that makes ϕ true.

   $$\exists x \,.\, \texttt{cat}(x) \wedge \texttt{orange}(x)$$

6. **Lambda expressions** name functions

   a. "Maternal grandmother": $\lambda x \,.\, \texttt{mother-of}(\texttt{mother-of}(x))$

   b. A lambda expression returning an individual applies to arguments to create a term:

   $$\underbrace{\lambda x \,.\, \texttt{mother-of}(\texttt{mother-of}(x))}_{\text{function}}\,(\underbrace{\texttt{Max}}_{\text{arg}})$$

   c. A function returning a truth value represents a relation. A 1-place function returning a truth value represents a set.

orange cat:	$\lambda x \,.\, \texttt{orange(x)} \wedge \texttt{cat}(x)$
Max is an orange cat:	$\lambda x \,.\, \texttt{orange(x)} \wedge \texttt{cat}(x)\,(\texttt{Max})$

Implementation

7. Semantic representation: Polish prefix form ("Lisp" form).

   a. Replace special symbols with names. Operator comes first, parentheses around expression.

   b. Example:

   $\forall x \,.\, \texttt{cat}(x) \rightarrow \texttt{cat}(\texttt{mother-of}(x))$
   `(forall x (if (cat x) (cat (mother-of x))))`

8. Expressions

   a. Are just tuples

   ```
   (chases Fido (the cat))
   ```

   b. Internally: `('chases', 'Fido', ('the', 'cat'))`

   c. Use the same trick that we used for `Category`:

   ```
   class Expr (tuple):
       def __repr__ (self):
           ...
   ```

9. `parse_expr(s)`

- **a.** Pattern it after `parse_tree()`, except that it should produce `Expr` objects instead of `Tree` objects.
- **b.** Import `tokenize()` from `hw1`.
- **c.** `parse_expr(`s`)` basically calls `parse_subexpr(tokenize(`s`),0)`
- **d.** The inner loop for `parse_subexpr()` is:

```
while toks[i] != ')':
    (child, j) = parse_subexpr(toks, i)
    children.append(child)
    i = j
```

- **e.** Example

```
>>> from hw4 import *
>>> e = parse_expr('(and (dog x) (friendly x))')
>>> e
(and (dog x) (friendly x))
>>> e[0]
'and'
>>> e[1]
(dog x)
>>> e[1][0]
'dog'
```

10. Variables

- **a.** It will be convenient if variables are represented by objects of class `Variable`.
- **b.** But otherwise variables should be identical to strings:

```
class Variable (str):
    ...
```

- **c.** Because variables are strings, two variables that look alike are equal:

```
>>> x1 = Variable('x')
>>> x2 = Variable('x')
>>> x1 is x2
False
>>> x1 == x2
True
>>> d = {}
>>> d[x1] = 42
>>> d[x2]
42
```

d. Make a variable print out without quotes, so you can tell it apart from a string:

```
>>> x1
x
>>> str(x1)
'x'
>>> 'lo' + x1
'lox'
```

e. How to tell variables from constants: a variable name is **a single letter, followed optionally by digits**.

```
>>> expr = parse_expr('(hi x)')
>>> expr[0]
'hi'
>>> expr[1]
x
>>> expr[1].__class__
<class 'hw4.Variable'>
```

f. The function `fresh_variable()` returns variables with names `_1`, `_2`, etc. These variables are never produced by parsing an expression.

Lambda expressions

11. Beta-reduction

a. Suppose we translate "chases Max" as:

```
(lambda x (chases x Max))
```

b. Apply that to "Fido":

```
((lambda x (chases x Max)) Fido)
```

c. General form: ((`lambda` *params body*) *args*)

d. Bind parameters to args: `x` → `Fido`

e. Using those bindings, replace variables in body. Result:

```
(chases Fido Max)
```

12. Some more examples:

a. Before:

```
((lambda (x y) (knows (mother y) x))
 Fido
 (the cat))
```

b. After:

```
(knows (mother (the cat)) Fido)
```

c. Before:

```
((lambda x (and (friendly x) (slobberer x)))
 Fido)
```

d. After:

```
(and (friendly Fido) (slobberer Fido))
```

13. Simplification

a. A **lambda application** is of form `((lambda` *params body*`)` *args*`)`

b. An expression is **simplified** if it does not contain any lambda applications.

14. `simplify1(expr, env)`

a. [*To be revised.*] If `expr` is a variable that is bound in `env`, return `env[expr]`.

b. If `expr` is not an `Expr`, return it.

c. If `expr` is a lambda expression, simplify the body and return a new lambda expression.

d. Simplify each of the children (subexpressions) and set `expr =` the result.

e. If `expr[0]` is a lambda expression, return `beta_reduce(expr, env)`, else return `expr`.

15. Helper: `is_lambda_expr()`. Returns `True` for an expression whose first element is `'lambda'`.

16. `beta_reduce (expr, env)`

a. The `expr` is of form `((lambda` *params body*`)` *args*`)`.

b. For each *param* and its corresponding *arg*, set `env[`*param*`] =` *arg*. [*To do: what if there is already a value?*]

c. Set `result = simplify1(`*body*, `env)`.

d. For each *param*, do `del env[`*param*`]`.

e. Return `result`.

17. A problem: apply `(lambda y (lambda x y))` to free variable `x`

a. Applying beta-reduction yields: `(lambda x x)`

b. What should be a free variable `x` has been incorrectly bound

c. Solution:

```
>>> e = parse_expr('((lambda y (lambda x y)) x)')
>>> e = normalize(e)
>>> e
((lambda _23 (lambda _24 _23)) x)
>>> beta_reduce(e, {})
(lambda _24 x)
```

d. Replacing variables bound by a lambda expression is called **alpha-conversion**.

e. An expression is **normalized** if each lambda parameter is unique to that lambda expression.

f. If the expression is normalized, there can never be a previous entry for *param* in (16b).

18. Back to (14a)

a. Consider:

```
((lambda _1 (foo _1 _1))
 (lambda _2 _2))
```

b. We store `(lambda _2 _2)` as `env[_1]`.

c. Then we fetch it twice:

```
(foo (lambda _2 _2) (lambda _2 _2))
```

d. The resulting expression is no longer normalized! The fix: normalize each time we fetch.

```
(foo (lambda _3 _3) (lambda _4 _4))
```

e. Revised version of (14a): If `expr` is a variable that is bound in `env`, return `normalize(env[expr])`.

19. `normalize (expr, repl=None)` – do this first, before `simplify1()`

a. If `repl` is `None`, set `repl = {}`. The table `repl` maps old variables to new variables.

b. If `expr` is a variable with a replacement in `repl`, return `repl[expr]`.

c. If `expr` is not an `Expr`, return it.

d. If `expr` is not a lambda-expression, return a new copy in which each subexpression has been normalized (using `repl`).

e. Otherwise `expr` has form `(lambda` *params body*`)`.

f. If *params* is not a tuple, make it a 1-tuple. Check that all *params* are variables.

g. Let `old` be the old replacements for the `params`, in `repl`. Use `None` for unset variables.

h. Let `newparams` be a list of **fresh variables**, one for each old param. Set `repl[param] = newparam` for each.

i. Set *body* `=` `normalize(`*body,* `repl)`.

j. Restore the old replacements.

k. Return a new lambda-expression using `newparams` and the normalized body.

20. `simplify (expr)`: do `simplify1(normalize(expr),{})`

21. Some examples. The algorithm handles these correctly.

a. Identity function

```
((lambda x x) fido)
```

b. Higher-order variables

```
((lambda f (f fido))
 (lambda x (dog x)))
```

c. Double expansion

```
(((lambda f (lambda x (f x x)))
  (lambda (x y) (likes y x)))
 fido)
```

d. Warning: infinite recursion is possible

```
((lambda x (x x))
 (lambda x (x x)))
```

Quantifiers, defined symbols

22. Kinds of quantifiers

a. "Regular" quantifiers: ∀, ∃

$$\forall x[\text{cat}(x) \to \text{animal}(x)]$$

$$\exists x[\text{cat}(x) \wedge \text{orange}(x)]$$

b. Generalized quantifiers: every, some

$$\text{every}(\text{cat}, \text{animal})$$

$$\text{some}(\text{cat}, \text{orange})$$

c. Definitions:

$$\text{every} = \lambda P \lambda Q[\forall x[P(x) \to Q(x)]]$$

$$\text{some} = \lambda P \lambda Q[\exists x[P(x) \wedge Q(x)]]$$

d. Restricted quantifiers

$$\text{every}\, x : \text{cat}(x)\ [\text{animal}(x)]$$

$$\text{some}\, x : \text{cat}(x)\ [\text{orange}(x)]$$

e. In our notation

```
(every x (cat x) (animal x))
(some x (cat x) (orange x))
```

23. Defined operators

a. Example: restricted quantifier “`every`”

```
every x R S: (forall x (if R S))
```

b. A definition consists of an **operator**, a list of **parameters**, and a **body**.

c. Store definitions in a table indexed by operator.

d. Suppose we wish to expand

```
(every c (cat c) (animal c))
```

e. Look up the definition of `every`. Bind the parameters to the arguments:

```
x -> c
R -> (cat c)
S -> (animal c)
```

f. In the body of the definition, replace the parameters with their values.

```
(forall c (if (cat c) (animal c)))
```

g. Do expansion on the result, if `forall` is itself a defined term. Otherwise, recurse.

h. Caution: circular definitions will lead to an infinite loop.