Computational category theory – a tutorial

generic.intelligence@gmail.com

October 20, 2018

After a few years of bad mathematics, I have read a lot of books recently. Computational category theory is Rydeheard & Burstall's 1988 book, but there are no newer textbooks on this topic.

I think this book is very important because it involves **unification algorithm**. The most critical issue in artificial intelligence seems to be how to transition from propositional logic to relational or first-order logic, especially how to find efficient learning algorithms. The basics of logic-based AI can be found in the book "AI — a modern approach".

The Unification algorithm is the core algorithm for predicate logic derivation. By adding this algorithm to the derivation algorithm of the propositional logic (which is called resolution), the derivation algorithm of the predicate logic can be obtained. In other words: unification + resolution = first-order deduction.

Unify means: Turn two **logical items** into "same" through **variable substitution**. Variable substitution is the essence of predicate logic, and **quantifiers** $\forall$ and $\exists$ are applied to these variables. All junior high school students know how to do "variable substitution", but it is actually a very troublesome action. Without it, the predicate logic becomes propositional logic. Although all mathematicians know what variable substitution is, its precise description did not begin until the 1920s and 30s. For example, Schönfinkel and Curry created combinatory logic, and Church created λ-calculus. One of their purposes is to reveal the mechanism of "substitution".

Example of Unification:

$$\text{loves}(X, Y) = \text{loves}(\text{john}, \text{mary}) \quad \text{with } \{X/\text{john}, Y/\text{mary}\} \tag{1}$$

(If the function symbols in the logic can be more complicated)

The Unifcation algorithm was originally proposed by Jacques Herbrand (1930), and the resolution algorithm was invented by J A Robinson (1965), which was then combined and applied to automatic reasoning.

In classic logic AI, unification is one of the core algorithms, but its time review is not a bottleneck. I have developed higher-order unification and have open source code on GitHub.

The following is a simple unification algorithm in Lisp by Peter Norvig:

```
;;; -*- Mode: Lisp; Syntax: Common-Lisp; -*-
;;; Code from Paradigms of Artificial Intelligence Programming
;;; Copyright (c) 1991 Peter Norvig

;;;; File unify.lisp: Unification functions
```

```lisp
(requires "patmatch")

(defparameter *occurs-check* t "Should we do the occurs check?")

(defun unify (x y &optional (bindings no-bindings))
  "See if x and y match with given bindings."
  (cond ((eq bindings fail) fail)
        ((eql x y) bindings)
        ((variable-p x) (unify-variable x y bindings))
        ((variable-p y) (unify-variable y x bindings))
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
                (unify (first x) (first y) bindings)))
        (t fail)))

(defun unify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  (cond ((get-binding var bindings)
         (unify (lookup var bindings) x bindings))
        ((and (variable-p x) (get-binding x bindings))
         (unify var (lookup x bindings) bindings))
        ((and *occurs-check* (occurs-check var x bindings))
         fail)
        (t (extend-bindings var x bindings))))

(defun occurs-check (var x bindings)
  "Does var occur anywhere inside x?"
  (cond ((eq var x) t)
        ((and (variable-p x) (get-binding x bindings))
         (occurs-check var (lookup x bindings) bindings))
        ((consp x) (or (occurs-check var (first x) bindings)
                       (occurs-check var (rest x) bindings)))
        (t nil)))

(defun subst-bindings (bindings x)
  "Substitute the value of variables in bindings into x,
  taking recursively bound variables into account."
  (cond ((eq bindings fail) fail)
        ((eq bindings no-bindings) x)
        ((and (variable-p x) (get-binding x bindings))
         (subst-bindings bindings (lookup x bindings)))
        ((atom x) x)
        (t (reuse-cons (subst-bindings bindings (car x))
                       (subst-bindings bindings (cdr x))
```

```
                        x))))

(defun unifier (x y)
 "Return something that unifies with both x and y (or fail)."
 (subst-bindings (unify x y) x))
```

From the perspective of deep learning, the question is how to integrate neural network algorithms into logic algorithms? On the surface, this is two very different things. The author has been thinking about this issue for many years and has proposed some programs, but it is not particularly successful. In order to better understand the mechanism of unification, I have seen some theories dealing with unification from the perspective of category theory.

The first person to study unification from the perspective of category theory was Joseph Goguen (1941-2006), who discovered that unification corresponds to the concept of **co-equalizer** in category theory:

He wrote a very detailed paper "What is unification? — a categorical view of substitution, equation and solution" (1989).

First introduce what is the **calculation** category theory. The category theory is abstract (it focuses on the **map** between collections, not the **element** in the collection), but the calculation is concrete. The key is: Use **types** to represent objects in category theory, and **functions** to represent morphisms in category theory. In other words, using **functional programming** to simulate category theory, the result of the calculation is some functions.

As we all know, Haskell's predecessor is ML, ML = Lisp + type system. ML also derives OCaml = Objective Caml, and Caml's CAM = categorical abstract machine. CAM is a combination of Cartesian-closed category and combinatory logic. (I don't know the theory of CAM for the time being)

In the book Rydeheard & Burstall, the language used is ML. E.g:

type `‘o` objects
type `‘a` arrows

Then the type of the source and target functions is

`‘a -> ‘o.`

In category theory, the definition of **co-equalizer** is:

$$\begin{tikzcd} X \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & Y \arrow[r, "q"] \arrow[dr, "q'"'] & Q \arrow[d, dashed, "!"] \\ & & Q' \end{tikzcd} \tag{2}$$

In other words, q is a universal arrow, making $qf = qg$.

Its **dual**, the definition of **equalizer**, "cone" is on the left:

$$\begin{tikzcd} Q \arrow[r, "q"] & X \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & Y \\ Q' \arrow[u, dashed, "!"] \arrow[ur, "q'"] & & \end{tikzcd} \tag{3}$$

Where $fq = gq$.

Their meanings are different (Awodey 2006) p.54-57:

An **equalizer** is a generalization of the idea of the **kernel** of a homomorphism, or an equationally defined "variety", like the zero-set of a real-valued function. In other words, sets of elements x for which $f(x) = g(x)$ for $f, g : X \to Y$.

A **co-equalizer** is a generalization of a **quotient** by an equivalence relation.

Here we only explain the co-equalizer: Define a relation on Y by $y_1 \rightsquigarrow y_2$ iff $\exists x \in X$ such that $f(x) = y_1$ and $g(x) = y_2$. Let $\simeq$ be the equivalence closure of $\rightsquigarrow$, and Q be the set of $\simeq$-equivalence classes. The quotient function $q : Y \to Q$ maps an element y to its equivalence class $[y]$ so that $qf = qg$.

In the R & B book §3.4.2 describes the scope of the logical **terms** $\mathcal{T}_\Omega(X)$, where X is the set of variables. A **term substitution** $f : X \to Y$ is a function that acts on this category:

$$f : X \to \mathcal{T}_\Omega(Y) \tag{4}$$

Unification works on **equations** in the form $s = t$.

Suppose there is a set of equations with index set I indicator: $\{s_i = t_i : i \in I\}$. This set of equations can be expressed like this:

$$\begin{tikzcd} I \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & X \end{tikzcd} \tag{5}$$

Where $f(i) = s_i, g(i) = t_i$. In other words, f and g point to the left and right ends of the equation, respectively.

The characteristics of the category-based algorithm are: put the function recursively **split** into a more subtle function to calculate.

Theorem 1. *If $q : X \to Q$ is the co-equalizer of the parallel pair:*

$$\begin{tikzcd} I \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & X \arrow[r, "q"] & Q \end{tikzcd} \tag{6}$$

and $r : Q \to R$ is the co-equalizer:

$$\begin{tikzcd} & X \arrow[dr, "q"] & & \\ I' \arrow[ur, "f'"] \arrow[dr, "g'"'] & & Q \arrow[r, "r"] & R \\ & X \arrow[ur, "q"'] & & \end{tikzcd} \tag{7}$$

then $rq : X \to R$ is the co-equalizer:

$$\begin{tikzcd} I + I' \arrow[r, shift left, "{[f,f']}"] \arrow[r, shift right, "{[g,g']}"'] & X \arrow[r, "rq"] & R \end{tikzcd} \tag{8}$$

The meaning of this spin-off is, for example:

$$\text{loves}(X, Y) = \text{loves}(\text{john}, \text{mary}) \tag{9}$$

Then split into two equations:

$$\begin{aligned} X &= \text{john} \\ Y &= \text{mary} \end{aligned} \tag{10}$$

There is also a theorem to split the term out of **sub-term**, which is explained in the book.

Conclusion

It is found that the original category theory's expression of unification is basically the same as the simplest naïve's algorithm (such as Lisp)! The difference is only from a more abstract point of view, but that's it. But It may be instructive to link the unification to other mathematical structures.

Knowing the structure of unification may make it easier to apply neural networks to logic, although there are still no specific ideas. In fact, if the syntax of first-order logic is used, the whole system is completely different from the classic symbolic AI, and the neural network seems to be more than one. Therefore, in order to truly play the role of the neural network, it is necessary to use the so-called "**distributed representations**". This is the direction I am thinking about now.

References

Awodey (2006). *Category theory*.

Goguen (1989). “What is unification - a categorical view of substitution, equation and solution”. In: *Resolution of equations in algebraic structures 1: algebraic techniques*, pp. 217–261.

Robinson (1965). “A machine-oriented logic based on the resolution principle”. In: *Communications of the ACM* 5, pp. 23–41.

Rydeheard and Burstall (1988). *Computational category theory*.

Simmons (2011). *An introduction to category theory*.