Very Elementary domain theory

The Origins of the subject via Lambda calculus

- We see a lot of domain theory in this workshop. It is rigorous work as it is mathematical.

- The structures are rather involved and complicated.

- But what inspired researchers to study domains in the first place?

- Today we will have a look at what motivated domain theory in the first place.

- In this talk we will consider (un-typed) lambda calculus and its semantics which inspired Dana Scott to invent what we call The Scott Domain.

- Lambda calculus is set in a syntactic framework. Domains where introduced by Dana Scott for the first time to give a model of Lambda calculus without types.

- We will try to give an intuitive tour of the subject.

- Lambda calculus was introduced by A. Church in the 1903's to serve as a foundation of computation.

- Church's work was parallel to his students' A.Turing and S.Kleene who each had their own formalism of computation!!!

- It could be said that most of what laid the foundation of computation was done in Church's school. Alan Turing, Stephen Kleene, Dana Scott were all Church's students. So Church was in a sense the spiritual father of an entire field of study.

- We will not look at Kleene's work but let us remind ourselves of Turing's formalism, the very well known Turing machine.

- Church's plan was to give a precise foundation of what it means for a function to be computable.

- He decided that a way of doing this is to consider a system in which everything is viewed as a function.

-Church's highly syntactical lambda calculus leads to the same class of functions as Turing's, so does Kleene's notion of computation.

– We can assume that everyone is familiar with Turing's formulation of the notion of computability via Turing machines.

- In the standard case in a Turing machine we have an infinite tape, a moving head and a very simple alphabet.

- All the actions of a Turing machine on its tape are comprised of moving symbols around. What happens if we just do that? That is, just move symbols around?

- Church's idea was to study manipulation of syntactic terms and to interpret computable functions in terms of such manipulations.

- Subsequently Turing proved that the functions that are computable in Church's view are the same as Turing's own computable functions.

-Now it is time to see what un-typed lambda calculus is. We do this via example and start with some examples from programming that have a lambda calculus like features. Here is the syntax

$$\text{function } x \longrightarrow *(2\ x) \qquad\qquad \text{Notation: } \lambda x.\ 2 * x$$

- The left side is in a format you'd expect to see in a programming language the right side is in Church's lambda notation.

$\lambda x.2x$ is the function that assigns $2 * x$ to x or the function that returns $2 * x$ when the input is x.

What does this function do when applied to 4 for instance?

$$(\text{function } x \longrightarrow *(2\ x))\ 4 \longrightarrow *(2\ 4) \longrightarrow 2 * 4 \longrightarrow 8$$

Likewise:

$$(\lambda x. * (2\ x))\ 4 \longrightarrow *(2\ 4) \longrightarrow 2 * 4 \longrightarrow 8$$

What does this expression refer to?

$$\text{function } x \longrightarrow (\text{function } y \longrightarrow * \ (\ (\ +\ (x\ y))\ \ 2))$$

This is a function of x that (given x) returns a function of y that given y returns the result of multiplying 2 and the sum of x and y.

In simpler terms this is the function $(x, y) \longrightarrow 2\ *(x + y)$.

In λ notation this is :

$$\lambda x.\ (\lambda y.\ (*\ (\ +\ \ (x\ y))\ 2)$$

Looks like an inconvenient notation but it has the advantage of dealing with functions of one variable only!

Before formal definitions let us see how expressions are ”calculated” in our informal programming language.

$$+\ ((*\ 8\ 3)\ (*\ 2\ \ 5)) \longrightarrow +\ ((8 * 3)\ \ *(2\ \ 5)) \longrightarrow +\ (24\ \ *(2\ \ 5)) \longrightarrow +\ (24\ \ (2 * 5))$$
$$\longrightarrow +\ (24\ \ 10) \longrightarrow 24 + 10 \longrightarrow 34$$

We could have done the calculation differently:

$$+\ (*\ (8\ \ 3)\ \ *(2\ \ 5)) \longrightarrow +\ (*(8\ \ 3)\ \ (2 * 5)) \longrightarrow +\ (*(8\ \ 3)\ \ 10) \longrightarrow +\ ((8 * 3)\ \ 10)$$
$$\longrightarrow +\ (24\ \ 10) \longrightarrow 24 + 10 \longrightarrow 34$$

We would expect the two different ways of evaluating a term to yield the same result. Keep that in mind for later.

Lambda terms are terms that are evaluated/reduced much like the example above. Except we do not have any built in functions like addition and multiplication. We have to define those.

Church's notation for functions (that will have reduction properties like above) is completely reductionist. This means one says all that a function does when introducing a function.

In other words $\lambda x.f(x)$ is THE function that returns $f(x)$ when given x as input. That is the function that we normally call f. With this idea in mind we will see what lambda terms are next.

Lambda terms.

Lambda terms are either variables or terms applies to terms or made by abstraction:

$$\text{Term:}\quad \text{Variable} \mid \text{Term Term} \mid \lambda\ \text{Variable.}\ \ \text{Term}$$

Examples:

$$xxy$$
$$\lambda x.x$$
$$\lambda y.yxy$$
$$(\lambda x.xy)z$$

We make some shorthand writing conventions.

$FM_1M_2\cdots M_n \equiv (((FM_1)M_2)\cdots M_n)$ and $\lambda x_1x_2\cdots x_n.M \equiv (\lambda x_1.(\lambda x_2(\cdots(\lambda x_n.M))))$

We only consider one notion of reduction here (there are others):

$$(\lambda x.M)N \longrightarrow M[x/N]$$

What does this mean?

Well, $\lambda x.M$ is THE function that assigns M to x. Presumably M has something to do with x, like a function of x. So when we apply this to a term N, all we are doing is to replace N for x.

This is what the notation $M[x/N]$ signifies. Well, x should appear free in M. This brings us to our next definition.

$$FV(x) = x$$
$$FV(T_1T_2) = FV(T_1) \cup FV(T_2)$$
$$FV(\lambda x.M) = FV(M)\backslash\{x\}$$

Some examples of reduction (β-reduction).

$$(\lambda xyz.yyzx)abc \longrightarrow (\lambda yz.yyza)bc \longrightarrow (\lambda z.bbza)c \longrightarrow bbca$$

We use equality for the terms that are reduced to one another, the same term, or are reduction of the same term in one or several steps. This is the informal definition of equality.

To see some reductions in more details how this work let:

$$I \equiv \lambda x.x \quad \text{and} \quad K \equiv \lambda xy.x \quad \text{and} \quad S \equiv \lambda xyz.xz(yz)$$

then:

$$KMN \longrightarrow M$$

because

$$KMN \equiv \lambda xy.x(MN) \equiv (((\lambda x.(\lambda y.x))M)N) \longrightarrow ((\lambda y.x)M)N) \longrightarrow (\lambda y.M)N \rightarrow M$$

Next,

$$SMNL \equiv ML(NL)$$

because

$$SMNL \equiv ((((\lambda xyz.xz(yz))M)N)L) \longrightarrow ((\lambda yz.Mz(yz))N)L \longrightarrow (\lambda z.Mz(Nz))L \longrightarrow ML(NL)$$

Finally

$$SKK \longrightarrow I$$

because

$$SKK \equiv ((((\lambda xyz.xz(yz))K)K) \longrightarrow (\lambda yz.Kz(yz))K \longrightarrow (\lambda yz.z)K \longrightarrow \lambda z.z \equiv I$$

Note that we have used the fact that

$$Kz(yz) \equiv ((\lambda xy.x)z)(yz) \longrightarrow (\lambda y.z)(yz) \longrightarrow z$$

One could have distinct reductions if the reductions do not end. If they do end however all reductions end with the same result.That is we have the property we were seeking at the very beginning. That is evaluation of terms independent of the path taken.

Here is an example when we have two distinct reduction paths.

$$(\lambda x.x)((\lambda y.yy)(\lambda y.yy)) \longrightarrow ((\lambda y.yy)(\lambda y.yy)) \longrightarrow ((\lambda y.yy)(\lambda y.yy)) \longrightarrow \cdots$$

or we could have

$$(\lambda x.x)((\lambda y.yy)(\lambda y.yy)) \longrightarrow (\lambda x.x)((\lambda y.yy)(\lambda y.yy)) \longrightarrow \cdots$$

where in the first line we first reduce the leftmost term and in the second one we keep reducing by applying the second term to the third one.

There are many implicit assumptions about what it means for a term to reduce to another. Let's go to the board for that.

There are explanations on the board that are no included in the slides.

So what does all this have to do with computation? How do we calculate by sliding variables around?

Church's idea was to see everything as functions. This stands in contrast to a set theorist's point of view that sees everything as a set.

So, just like in set theory that we see a number, say 2, as a set with TWO elements here we see the number 2, as applying a function twice.

Church defined what we know as Church numerals by saying, n is the term that applies a term n times. Here is the exact definition. First,

$$F^0(M) \equiv M$$
$$F^{n+1}[M] = F(F^n(M))$$

So the Church numeral C_n which stands for n is defined to be the term that once given two variables f and x it applies f to x, n times.

$$C_0 \equiv \lambda fx.x$$
$$C_n \equiv \lambda fx.f^n x$$

Now we can define a computable function as a lambda definable function.

That is, we say that a function on natural numbers is definable if its translation to Church's numerals has a lambda term.

More precisely $f : \mathbb{N} \longrightarrow \mathbb{N}$ is λ-definable (computable)if and only if there is λ-term $\bar{f}$ such that

$$\bar{f}C_n = C_{f(n)}$$

Turing showed that the functions that are λ-definable are exactly the functions that are Turing definable/recursive. We will see a few example. You can check out the details later.

The following are the lambda terms for the basic arithmetical operations. These are due to Rosser (yet another student of Church). It could be said that the entire literature on computability was mainly coming out of Church's school back in the days it was first being developed.

$$A_+ \equiv \lambda xypq.xp(ypq).$$

Note that this means that A_+ applies to C_nC_m reduces to C_{m+n}. Try it out.

Next:

$$A_* \equiv \lambda xyz.x(yz).$$

Finally

$$A_{exp} \equiv \lambda xy.yx.$$

Note that the lambda terms for more complex operations are simpler!! Also note that

$$(C_nx)^m(y) = x^{n*m}(y) \quad \text{and} \quad (C_n)^m(x) = C_{(n^m)}(x)$$

Let us now have a look at features of lambda calculus that are programming like.

We define $true \equiv \lambda xy.x$ and $false \equiv \lambda xy.y$.

A term is called Boolean if it is either true or false. Now what do you guess is the term that stands for

If B is true then P and if B is false then Q?

The answer is very simple

If B then P else $Q \equiv BPQ$.

If B is true then (the term true) picks the first argument and if B is false it picks the second term.

What you see here is characteristic of lambda calculus. The definitions are chosen in a smart way that lets the operators to be defined in a very simple manner.

Here is another example. To create a "pair" of terms we define

$$[M, N] \equiv \lambda z. \text{ if} z \text{ then } M \text{ else } N$$

This works like a pairing operation. The terms "true" and "false " work like projections:

$$[M, N]true \longrightarrow M \quad \text{and} \quad [M, N]false \longrightarrow N.$$

From what we have seen we can tell Lambda calculus relies on the idea that terms are like functions.

Every term can apply to any other term.

So every term is both a function and an input for a function.

This is why finding a set theoretical model for lambda calculus was a challenge.

The class of all functions on a set is strictly bigger than the set itself.

So a set that represents all terms, cannot have enough elements to account for all functions.

Scott's idea to solve this issue was to consider special partial orders (he started with lattices) whose class of monotone functions could also be seen as continuous functions under a special topology (Scott's topology).

So instead of looking at the class of all functions on a "domain" (whose elements represent terms) one looks at the class of all continuous functions on that domain.

Scott was looking for a domain D whose class of continuous functions $[D \longrightarrow D]$ could be embedded back in D. This was called a reflective domain (lattice).

We will have a brief look at why such a domain resolves the issue of size for the class of all (continuous) functions. Scott actually built a domain that does have the required embeddings. We will not get to do this here.

What matters here is what Scott wanted. HOW he did it is what led to domain theory and all the wonderful mathematics that came out of it.

He created something he first called T^{∞} and was essentially a reflective partial order in the sense we explain next.

Here are the main points.

Elements of the lattice/domain/partial orders under the study represent data.

All computable functions are monotone (order preserving).

All computable functions are continuous under a special topology that is called Scott topology.

We can therefore focus on continuous functions as computable functions.

Scott proved that a domain D with the following properties exist.

1. The class of continuous functions on D can be ordered (topologized) to be a continuous lattice (in the sense we don't get into here). This class is denoted by $[D \longrightarrow D]$

2. There are a pair of functions $F : D \longrightarrow [D \longrightarrow D]$ and $G : [D \longrightarrow D] \longrightarrow D$ that are both continuous.

3. One goes back and force between $[D \longrightarrow D]$ and D via F and G. $F \circ G = \mathrm{Id}_D$

By using the functions F and G one can identify an element x of the domain with a function on the domain $F(x)$ and a conversely a function f in $[D \longrightarrow D]$ is identified with $G(f)$ which is an element of the domain.

This part includes item explained on the board not included in the slides.

Once this back and forth structure is established, modelling of lambda calculus becomes a natural consequence.

The conclusion that lambda calculus is consistent (that is true and false are distinct) is only one of the consequences of Scott's model construction.

One could prove Lambda calculus's consistency in purely syntactical terms too.

Nonetheless Scott's work provided a neat proof of consistency and laid the work for denotational semantics.

Our goal was not to give details of Scott's construction. Rather we wished to show the main example that motivated the work.

Questions and comments are welcome: Saliyari@umail.iu.edu