Learning to Learn OCaml

Alexander Berenbeim

2016-10-10 Monday

Outline

Turing Completeness

A programming language is Turing Complete when ::

- it can map every Turing machine to a program
- Turing machines can be emulated in a program
- It is a superset of a known Turing-complete language

One such example is the λ-calculus, which is the basis of Lisp, ML, Haskell, and OCaml (oh, and Scheme too).
A Correspondence between ALGOL 60 and Church's λ-notation (Landin) proves that sequential procedural programming languages can be understood using λ-calculus.

What is the λ-calculus?

- Simply put, higher-order functions; the functions are first-class values.
- Syntactically, an expression (or term) e is inductively defined as follows: $e ::= x \mid \lambda\, x.e_1 \mid (e_1\ e_2)$
- A λ-abstraction $\lambda x.e$ sets up an application; in OCaml this is simply

```
(fun x -> e)
```

- An abstraction binds the formal parameter x to the entirety of the expression to the right;
- An application calls the the first expression with the

second term as input; applications are left associative, e.g. $x\ (\ y\ z\ )\ !{\equiv}(\ x\ y\ )\ z$.

Question:

- What is $\lambda x.x$? What is $(\ x\ x\ )$? What is $\lambda x.x\ (\ x\ )$?

Basic Relations on λ terms: α,β,η

- The basic equivalence relation on λ-terms is convertibility. Given an arbitrary λ expression M:
- α-conversion::
- *change of bound variables* in M replaces a part of λ x.N of M with λ y.(N[x:=y]), where y does not occur in N;
- M $\equiv_\alpha$ N if N results from M by a series of changes of bound variable; this is captured by the α- *conversion* scheme: λ x.M=λ y.M[x:=y]
- β-reduction :: apply functions to arguments
- $(\lambda x.M)N = M[x := N]$
- η-conversion :: extensionality in λ-calculus, when two functions are the *same*
- If x is not free in f, then $\lambda x.(fx) \equiv_\eta f$

A Fixed Point Theorem & Some More Syntactic Notions

Theorem: ∀ F ∃ X FX = X.

PROOF Given expression F, let W ≡ λ x.F(xx). Further, let X ≡ WW. Then, syntactically

$$X \equiv WW \equiv_\alpha (\lambda x.F(xx)))W =_\beta F(WW) =_\alpha FX$$

Free Variables (Inductively)

1. FV(x)={x}, FV(λ x.M)= FV(M)-{x}, FV(MN)=FV(M)∪ FV(N);
2. M is closed or a combinator if FV(M)=∅;
3. Λ^0={M∈ Λ| M is closed};
4. $\Lambda^0(\bar{x}) = \{M \in \Lambda | FV(M) \subsetneq \bar{x}\}$;
5. A closure of M is $\lambda\bar{x}.M$, where $\bar{x} = FV(M)$.

The (simply) Typed λ calculus

We define Typ, the set of types inductively as

1. $0 \in \mathtt{Typ}$;
2. $\sigma,\tau \in \mathtt{Typ} \Rightarrow (\sigma \to \tau) \in \mathtt{Typ}$

λ^τ, the typed λ-calculus is defined as follows:

1. λ^τ has alphabet (,), λ, and variables v_i^σ for each type σ;
2. the set of `terms of type` σ, Λ_σ inductively defined by:

$v_i^\sigma \in \Lambda_\sigma$; M$\in \Lambda_{\sigma\to\ \tau}$,N$\in \Lambda_\sigma \Rightarrow$ (M N)$\in \Lambda_\tau$; M$\in \Lambda_\tau$, x$\in \Lambda_\sigma \Rightarrow$ (λ x.M)$\in$ $\Lambda_{\sigma\to\ \tau}$, with x ranging over the variables.

1. Formulae of λ^τ consists of equations M=N, with $M, N \in \Lambda_\sigma$;
2. Free,bound,closed, and substitutions are naturally defined;
3. λ^τ is axiomated by equality axiom and rules and the (β) scheme :(λ x .M)N=M[x:=N];
4. $\lambda\eta^\tau$ extends λ^τ by (η) scheme: if x$\notin$ FV(M), then λ x.Mx=M.

Motivation: The Propositions as Types paradigm

- The underlying idea is that proofs-are-programs, by relating syntactical derivations via an appropriately well-specified syntax-semantics adjunction (a Curry-Howard isomorphism)
- Let Ω denote a subobject classifier in an (intuitionstic) logic, and suppose S is some set S. A predicate of S is an object in the function space Ω^S.
- A proposition `M:*` holds when there is a witness to M
- `P: S-> *` is the type of the predicate and `P[x]` is the claim that P holds for $x \in S$
- $\forall x \in S, \varphi(x)$ is equivalent to $\Pi_{x:S}\varphi[x]$

Proposition as Types example

For example $\forall x \forall y P(x, y) \to \forall x P(x, x)$ is equivalent to

$$(\prod_{x:S} \prod_{y:S} P[x, y]) \to (\prod_{x:S} P[x, x])$$

and is 'true' if

$$\lambda H : (\prod_{x:S} {}_{y:S} P[x, y]).\lambda x : S.H[x, x]$$

is a valid construction from the *context*.

Dependent Sum Type Formers

- In the previous slides, we used a type former Π, that was identified with universal quantification.
- There is a similar notion generalizing coproducts and existential

quantification: the dependent sum type

- The proposition ($\exists$ x$\in$ A, φ(x)) is equivalent to $\Sigma_{x:A}\varphi[x]$.
- A witness of this type is an ordered pair $\langle$ a, b$\rangle$; such a witness is syntactically derived if

$$\Gamma \vdash a : A$$

and

$$\Gamma, a : A \vdash b : B$$

Barendregt's Λ-cube

/Users/logos/Math For The Real World/In OCaml Lectures/lamb

Barendregt's Λ-cube unpacked

- $\lambda_{\rightarrow}$ may be viewed as a single sorted theory with the sort type denoted by *, and with vertex defined by ordered pair ⟨*,*⟩
- If we add a second sort, □, denoting kinds, we can understand each vertice as corresponding to an element of the power set of the following ordered pairs:

$$\{\langle *, \Box\rangle, \langle \Box, \Box\rangle, \langle \Box, *\rangle\}$$

1. Polymorphic Types :: $\lambda 2$ :: ⟨ *,□⟩ :: kinds dependent on types
2. Type Operations :: $\lambda \underline{\omega}$ :: $\langle \Box, \Box\rangle$ :: kinds dependent on kinds
3. Dependent Types :: λ Π :: $\langle \Box, *\rangle$ :: types dependent on kinds

Dependencies of types and terms

In the literature one often sees a different labeling of these two sorts: $*$ as terms and $\square$ as types. Either way we still identify the following:

Normal functions :: $\lambda_{\rightarrow}$ terms dependent on terms

Polymorphism :: $\lambda 2$ Terms dependent on Types

Type operators :: $\lambda \underline{\omega}$ Types dependent on Types

Dependent types :: λ Types depending on terms

While the simply typed lambda calculus isn't very expressive (it's basically PL), in the polymorphic λ-calculus, we can do second order logic and express the parametric identity function: $\vdash \Lambda\alpha.\lambda\ x : \alpha.x : \forall\ \alpha.\alpha \rightarrow \alpha$

Why does this matter?

- Well, $\lambda\Pi\omega$ describes the typed system with polymorphic

types, type operations and dependent types. This is known as the Calculus of Constructions.

- CoC extends the Curry-Howard isomorphism that relates each natural-deduction proof to a term in the simply typed λ calculus.
- Coq, the interactive theorem prover written in OCaml, is a dependently typed functional programming language expressing a further extension of CoC, the Calculus of Inductive Constructions, by adding inductive type former rules.

What Does This Have To Do With OCaml?

- `Caml`, Categorical Abstract Machine Language, was developed by `INRIA` in part to help develop the `Coq` system in the 80s.
- `OCaml` was developed in part due to the rise of type systems and type inference in object-oriented programming in the 90s, by extending `Caml` to support type-parametric classes, binary methods, and other object oriented paradigms in a statically type-safe way while avoiding unsoundness or the run-time type checks that occur in classical Object Oriented languages like `C++` and `Java`.

So, what exactly does this mean for us?

- Well, putting this all together, an *expression* or *term* is any

valid OCaml program. Every valid *expression* has a type, and to produce an *answer*, OCaml *evaluates* the expression.

- OCaml's type syntax is explicit as the interpreter relies on it to tell you the type of every value; by type inference, we often will not need to specify the type ourselves.
- OCaml allows us to define new types using the `type` keyword. Formally, most user-defined types are formed with with or constructors, i.e. they are either sum types or product types.
- We can also define types with nullary constructors, i.e. constructors without arguments:

```
# type day = Monday | Tuesday | Wednesday | Thursday |
Friday | Saturday | Sunday;;
```

Sum Types

- nullary constructors are used to describe monomorphic types; we can define polymorphic types as well, like

```
# type 'a list =
Nil
| Cons of 'a * 'a list;;
```

- Lists are one example of sum types

```
# [true; false; true; true] : bool list;;
# [1;2;3;4;5] : int list;;
# [1,2,3] : int * int * int list;;
```

Product Types

- Product types are finite labeled products of types. They are the generalization of *cartesian products*, whose witnesses are called records.

```
# type ta =
# {name : string; email : string ;
 schedule : string * day * int list};;
```

- If Alex : ta, then Alex is a *record* of the type *ta*, and this means

```
# Alex : string * string * (string * day * int list) =
 "Alexander Berenbeim", "aberen2@uic.edu",
[("Model Theory", "Monday", 11); ("AI", "Monday", 13);
("Model Theory", "Wednesday", 11); ("AI", "Wednesday", 13);
("Model Theory", "Friday", 11);("AI", "Friday", 13)];;
```

If we want to extract information from Alex, we might want to use pattern matching. . .

Pattern Matching

- A pattern is not an OCaml expression, rather it is an arrangement of elements of our alphabet (constants of primitive type, variables, constructors, and the symbol denoting the wildcard pattern). Pattern matching applies to values by recognizing the form of a value and guides the computation accordingly by associating each pattern to an expression to be computed.
- We can use pattern matching to define functions

```
# let negation b = match b with
true -> false
| false -> true;;
val negation : bool -> bool = <fun>
```

Another Example

```
# let tomorrow d = match d with
Monday -> Tuesday
| Tuesday -> Wednesday
| Wednesday -> Thursday
| Thursday -> Friday
| Friday -> Saturday
| Saturday -> Sunday
| Sunday -> Monday ;;
val tomorrow : day -> day = <fun>
```

Yet Another Example

So far we've looked at defining functions solely by pattern matching by cases. Let's do something more interesting:

```
# let f = fun (n,m) -> 2 * n * n + 3 * m * m - n * m;;
val f : int * int -> int = <fun>
```