Lambda calcul

1 Codificări Church

Deși aparent simplu, și cu o singură regulă de calcul (aplicarea unei funcții), lambda-calculul este un limbaj la fel de puternic ca orice alt limbaj. Din punct de vedere matematic, limbajul este Turing-complet, adică poate calcula orice funcție care poate fi calculată de o mașină Turing.

Pentru a arăta acest lucru, vom folosi un set de codări (encodings) inteligent proiectate, numite codări Church (Alonzo Church a creat calculul lambda).

1.1 Valori Booleene

Iată codările pentru valori booleene:

$$\begin{aligned} TRUE &= \lambda x.\lambda y.x \\ FALSE &= \lambda x.\lambda y.y \end{aligned}$$

Valoarea "adevărat" va fi reprezentată de o funcție care primește două argumente și întoarce primul dintre ele, iar valoarea "fals" de o funcție cu două argumente care întoarce al doilea argument.

Operațiile booleene obișnuite sunt codate după cum urmează:

$$\begin{aligned} AND &= \lambda u.\lambda v.u\ v\ u \\ OR &= \lambda u.\lambda v.u\ u\ v \\ NOT &= \lambda u.u\ FALSE\ TRUE \end{aligned}$$

Putem verifica că într-adevăr termenii de mai sus se comportă conform așteptărilor:

$$\begin{aligned} & AND\ TRUE\ FALSE = \\ & (\lambda u.\lambda v.u\ v\ u)\ TRUE\ FALSE \to_\beta \\ & (\lambda v.TRUE\ v\ TRUE)\ FALSE \to_\beta \\ & TRUE\ FALSE\ TRUE = \\ & (\lambda x.\lambda y.x)\ FALSE\ TRUE \to_\beta \\ & (\lambda y.FALSE)\ TRUE \to_\beta \\ & FALSE. \end{aligned}$$

$$\begin{aligned} & AND\ TRUE\ TRUE = \\ & (\lambda u.\lambda v.u\ v\ u)\ TRUE\ TRUE \to_\beta \\ & (\lambda v.TRUE\ v\ TRUE)\ TRUE \to_\beta \\ & TRUE\ TRUE\ TRUE = \\ & (\lambda x.\lambda y.x)\ TRUE\ TRUE \to_\beta \\ & (\lambda y.TRUE)\ TRUE \to_\beta \\ & TRUE. \end{aligned}$$

$$\begin{aligned}
&\mathit{AND\ FALSE\ TRUE} = \\
&(\lambda u.\lambda v.u\ v\ u)\ \mathit{FALSE\ TRUE} \to_\beta \\
&(\lambda v.\mathit{FALSE}\ v\ \mathit{FALSE})\ \mathit{TRUE} \to_\beta \\
&\mathit{FALSE\ TRUE\ FALSE} = \\
&(\lambda x.\lambda y.y)\ \mathit{TRUE\ FALSE} \to_\beta \\
&(\lambda y.y)\ \mathit{FALSE} \to_\beta \\
&\mathit{FALSE}.
\end{aligned}$$

$$\begin{aligned}
&\mathit{AND\ FALSE\ FALSE} = \\
&(\lambda u.\lambda v.u\ v\ u)\ \mathit{FALSE\ FALSE} \to_\beta \\
&(\lambda v.\mathit{FALSE}\ v\ \mathit{FALSE})\ \mathit{FALSE} \to_\beta \\
&\mathit{FALSE\ FALSE\ FALSE} = \\
&(\lambda x.\lambda y.y)\ \mathit{FALSE\ FALSE} \to_\beta \\
&(\lambda y.y)\ \mathit{FALSE} \to_\beta \\
&\mathit{FALSE}.
\end{aligned}$$

Exercițiu: verificați comportamentul funcțiilor *OR* și *NOT*.

1.2 Numere Naturale

Numele naturale pot fi codate după cum urmează:

$$\begin{aligned}
\mathit{0} &= \lambda f.\lambda x.x \\
\mathit{1} &= \lambda f.\lambda x.f\ x \\
\mathit{2} &= \lambda f.\lambda x.f\ f\ x \\
\mathit{3} &= \lambda f.\lambda x.f\ f\ f\ x \\
\ldots
\end{aligned}$$

Cu alte cuvinte, un număr natural n va fi reprezentat printr-o funcție care primește două argumente: f și x și aplică f de n ori pe x.

Funcția succesor este foarte simplu de scris ca lambda-termen:

$$\mathit{SUCC} = \lambda n.\lambda f.\lambda x.((n\ f)\ (f\ x)).$$

Să calculăm *SUCC* 2:

$$\begin{aligned}
&\mathit{SUCC}\ 2 = \\
&\lambda n.\lambda f.\lambda x.((n\ f)\ (f\ x))\ 2 \to_\beta \\
&\lambda f.\lambda x.((2\ f)\ (f\ x)) = \\
&\lambda f.\lambda x.(((\lambda f.\lambda x.f\ f\ x)\ f)\ (f\ x)) \to_\beta \\
&\lambda f.\lambda x.((\lambda x.f\ f\ x)\ (f\ x)) \to_\beta \\
&\lambda f.\lambda x.(f\ f\ f\ x) = \\
&3.
\end{aligned}$$

Funcția de adunare a două numere poate fi reprezentată ca lambda-termen după cum urmează:

$$\mathit{PLUS} = \lambda n.\lambda m.\lambda f.\lambda x.m\ f\ (n\ f\ x).$$

Exercițiu: calculați *PLUS* 2 3.

Funcția de înmulțire poate fi reprezentată astfel:

$$MULT = \lambda n.\lambda m.\lambda f.\lambda x.m\ (n\ f)\ x.$$

Exercițiu: găsiți alte definiții pentru *MULT*.
Exercițiu: găsiți o definiție pentru *EXP* (funcția de exponențiere).
Putem defini *ISZERO* astfel:

$$ISZERO = \lambda n.n\ (\lambda x.FALSE)\ TRUE$$

Pentru funcția *predecesor* avem nevoie de câteva preliminarii.

1.3 Perechi

Putem forma perechi și accesa componentele folosind următorii lambda-termeni (perechile fiind reprezentate ca o funcție care primește o valoare de adevăr și întoarce sau prima componentă, dacă valoarea este TRUE, sau a doua, dacă valoarea este FALSE):

$$\begin{aligned} PAIR &= \lambda u.\lambda v.\lambda b.b\ u\ v \\ FST &= \lambda p.p\ TRUE \\ SND &= \lambda p.p\ FALSE \end{aligned}$$

1.4 Predecesor

Fie perechea $(0, 0)$.
Aplicăm următoarea transformare de n ori:

- copiem a doua componentă a perechii vechi pe prima poziție în noua pereche;
- incrementăm a doua componentă a perechii vechi.

Obținem următoarea secvență de perechi: $(0, 0) \Rightarrow (0, 1) \Rightarrow (1, 2) \Rightarrow (2, 3) \Rightarrow \ldots$.
Pe prima componentă se găsește predecesorul.

$$\begin{aligned} PRED &= \lambda u.FST\ (u\ AUX\ (PAIR\ 0\ 0)) \\ AUX &= \lambda p.PAIR\ (SND\ p)\ (SUCC\ (SND\ p)) \end{aligned}$$

1.5 Observații

1. Numărul 0, valoarea *fals* și lista vidă au aceeași reprezentare;
2. Limbajul nu este puternic tipizat. De exemplu se poate calcula *AND 2 5*.

1.6 Combinator de punct fix

$$Y = \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y))$$

Un combinator *FIX* este de punct fix dacă *FIX* t are același comportament cu t (*FIX* t).
Putem folosi un combinator de punct fix după cum urmează:

$$\begin{aligned} ISEVEN &= Y\ ISEVEN' \\ ISEVEN' &= \lambda f.\lambda n.ITE\ (ISZERO\ n)\ TRUE\ (NOT\ (f\ (PRED\ n))) \end{aligned}$$