Lambda Cube Part 2

Lambda Cube Crew

June 28, 2023

Contents

1 Polymorphic types, $\lambda 2$

The expressiveness of $\lambda \to$ is limited since a function is always bound to its type. I.e. we cannot reuse e.g. the identity function integers for other types.

$\lambda 2$ adds parametric polymorphism to increase expressibility and allow reuse of functions over multiple types.

We add the following constructs to our language:

$$\begin{array}{lcl} \mathcal{T} & ::= & ... \mid \lambda\alpha.\mathcal{T} \mid \mathcal{T}[\mathcal{T}] \\ \text{Type} & ::= & ... \mid \text{TVar} \mid \forall\alpha.\text{Type} \end{array}$$

We add a new rule for β-reduction

$$(\lambda\alpha.M)\tau \to_\beta M[\tau/\alpha]$$

We also need typing rules for the new terms.

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.\sigma}$$

$$\frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M\tau : \sigma[\tau/\alpha]}$$

As an example we can now type self application as

$$\lambda x : (\forall\alpha.\alpha).(x(\sigma \to \tau))(x\sigma) : (\forall\alpha.\alpha) \to \tau$$

1.1 Properties of $\lambda 2$

- Uniqueness of types:

$$\Gamma \vdash M : \sigma \wedge \Gamma \vdash M : \tau \implies \sigma = \tau$$

- Church-Rosser property holds.
- Subject reduction:

$$\Gamma \vdash M : \tau \wedge M \to_{\beta\eta} M' \implies \Gamma \vdash M' : \tau$$

- Strong normalization:

$$\Gamma \vdash M : \tau \implies M \in \mathrm{SN}$$

Sidenote: Strong normalization of $\lambda 2$ is a Gödel sentence, i.e. it is expressible in Peano arithmetic, but not provable in this system.

1.2 Expressiveness of $\lambda 2$

1.2.1 Natural numbers

Can be expressed as the type

$$\forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$$

The terms corresponds to the Church numerals:

$$0 \equiv \lambda\alpha.\lambda x : \alpha.\lambda f : \alpha \to \alpha.x$$

$$n \equiv \lambda\alpha.\lambda x : \alpha.\lambda f : \alpha \to \alpha.f^n x$$

$$succ \equiv \lambda n : Nat.\lambda\alpha.\lambda x : \alpha.\lambda f : \alpha \to \alpha.f(n\alpha x f)$$

Theorem: $\lambda 2$ types exactly all primitive recursive functions.

This means that almost all meaning full programs are typable in $\lambda 2$.

An example of a function that cannot be typed in $\lambda 2$: The Ackerman-Péter function:

$$\begin{array}{lcl} A(0,n) & = & n+1 \\ A(m+1,0) & = & A(m,1) \\ A(m+1,n+1) & = & A(m,A(m+1,n)) \end{array}$$

- This is recursive, but not primitive recursive

1.3 Curry-Howard correspondence

$\lambda 2$ corresponds to constructive second order propositional logic (PROP2), i.e. a second order logic formula holds if the corresponding type is inhabited in $\lambda 2$.

Theorem: (Girard, Reynolds, Curry-Howard) (see [3] for a summary).

$$\vdash \sigma \iff \exists M. \vdash M : \sigma$$

Note that this is constructive logic, e.g. Pierce's law

$$\forall \alpha . \forall \beta . ((\alpha \to \beta) \to \alpha) \to \alpha$$

is not inhabited, and therefore does not hold constructively. (It does hold in classical logic).

We can define some connectives from PROP2:

$$\begin{array}{lcl} \bot & \equiv & \forall \alpha . \alpha \\ \sigma \wedge \tau & \equiv & \forall \alpha . (\sigma \to \tau \to \alpha) \to \alpha \\ \sigma \vee \tau & \equiv & \forall \alpha . (\sigma \to \alpha) \to (\tau \to \alpha) \to \alpha \\ \exists \alpha . \sigma & \equiv & \forall \beta . (\forall \alpha . \sigma \to \beta) \to \beta \end{array}$$

- In constructive propositional logic, connectives are independent.
- PROP is minimal logic - implicational fragment of constructive propositional logic.

1.4 Type checking, inference and inhabitation

Theorem: In $\lambda 2$

- Type checking is undecidable.

- Type inference is undecidable.
- Type inhabitation in undecidable.

Proof:

- Type inhabitation is equivalent to provability in PROP2, by Curry-Howard correspondence.
- Cases for type checking and type inference was proved by 1990 by Wells [4]

Note that a restriction of $\lambda 2$ used by languages like Haskell and ML allows for an efficient type inference algorithm due to Hindley and Milner [1, 2].

2 $\lambda\underline{w}$

The main idea of $\lambda\underline{\omega}$ is to extend $\lambda\rightarrow$ to let types depend on types, similarly to how $\lambda 2$ lets terms depend on types. E.g. we would like to express a function f, that we can apply to some type σ to to get the type

$$f(\sigma) = \sigma \rightarrow \sigma$$

In $\lambda\underline{\omega}$ we would express this as

$$f \equiv \lambda\alpha : \star.\alpha \rightarrow \alpha$$

$\star$ here is a kind. Intuitively it describes the "type" of types, so $\alpha : \star$ tells us that α is a type. We write $f : \star \rightarrow \star$ to describe the kind of f. We can see this as a function from types to types.

We can informally define the set of kinds as

$$\mathcal{K} = \{\star, \star \rightarrow \star, ...\}$$

Formally we write $k \in \mathcal{K}$ as $k : \square$.

We have

- If σ is a type, then $\sigma : \star$
- If $k : \square$, and $f : k$, then f is the *constructor of kind* k.

2.1 Formal definition

We define pseudo-expressions (including both terms and types) as

$$\mathcal{T} ::= \mathcal{V} \mid \mathcal{C} \mid \mathcal{T}\mathcal{T} \mid \lambda\mathcal{V} : \mathcal{T}.\mathcal{T} \mid \mathcal{T} \to \mathcal{T}$$

Furthermore we have the sorts

$$S = \{\star, \Box\} \subseteq \mathcal{C}$$

In a typing judgement $\Gamma \vdash M : A$, both M and A are pseudo-expressions.

Γ can now also contain elements like $\alpha : \star$. This demands that Γ is ordered, since types of other variables in Γ can depend on α. E.g.

$$\begin{array}{rcl} \alpha : \star, x : \alpha & \vdash & x : \alpha \\ \alpha : \star & \vdash & \lambda x : \alpha.x : \alpha \to \alpha \end{array}$$

are valid typing judgements. However

$$\begin{array}{rcl} x : \alpha, \alpha : \star & \vdash & x : \alpha \\ x : \alpha & \vdash & \lambda\alpha : \star.x : \star \to \alpha \end{array}$$

are **not**.

The typing rules for $\lambda\underline{\omega}$ is

$$\text{(ax/sort)} \vdash \star : \Box$$

$$\text{(weak)}\ \frac{\Gamma\vdash M:B \qquad \Gamma\vdash A:s}{\Gamma,x:A\vdash M:B}\ \text{if } x \notin \Gamma$$

$$\lambda\ \frac{\Gamma,x.A\vdash M:B \qquad \Gamma\vdash A\to B:s}{\Gamma\vdash\lambda x:A.M:A\to B}$$

$$\text{(app)}\ \frac{\Gamma\vdash M:A\to B \qquad \Gamma\vdash N:A}{\Gamma\vdash MN:B}$$

$$(\text{conv}_\beta)\ \frac{\Gamma\vdash M:A \qquad \Gamma\vdash B:s}{\Gamma\vdash M:B}\ A \equiv_\beta B$$

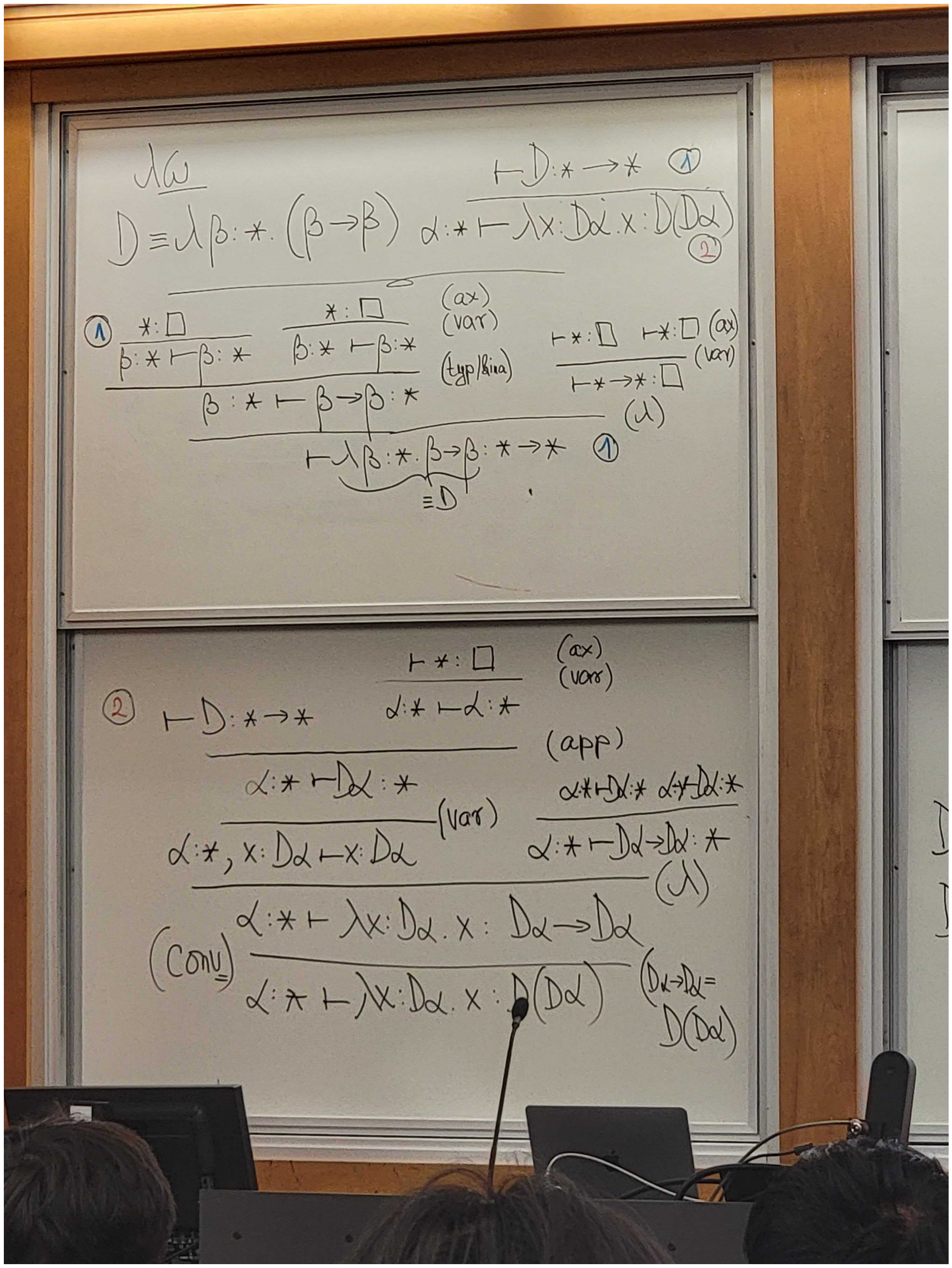
$$\text{(type/kind)}\ \frac{\Gamma\vdash A:s \qquad \Gamma\vdash B:s}{\Gamma\vdash A\to B:s}$$

$$\text{(var)}\ \frac{\Gamma\vdash A:s}{\Gamma,x:A\vdash x:A}\ \text{if } x \notin \Gamma$$

As an example, here is a typing derivation of

$$\alpha : \star \vdash \lambda x : D\alpha.x : D(D\alpha)$$

where $D \equiv \lambda\beta : \star.(\beta \to \beta)$.


λω
D ≡ λβ:*. (β→β)
⊢ D : * → * ①
α:* ⊢ λx:Dα.x : D(Dα) ②
① *:□ *:□ (ax)
β:* ⊢ β:* β:* ⊢ β:* (var)
(typ/bina)
β:* ⊢ β→β:*
⊢*:□ ⊢*:□ (ax)
⊢ *→*:□ (var)
(λ)
⊢ λβ:*.β→β : *→* ①
≡D
② ⊢ D:*→* ⊢*:□ (ax)
α:* ⊢ α:* (var)
(app)
α:* ⊢ Dα:*
(var)
α:*, x:Dα ⊢ x:Dα
α:* ⊢ Dα→Dα:*
(λ)
α:* ⊢ λx:Dα.x : Dα→Dα
(conv)
α:* ⊢ λx:Dα.x : D(Dα)
(Dα→Dα = D(Dα))

2.2 Properties of $\lambda\underline{\omega}$

- Uniqueness of types:

$$\Gamma \vdash M : \sigma \wedge \Gamma \vdash M : \tau \implies \sigma = \tau$$

- Church-Rosser property holds.
- Subject reduction:

$$\Gamma \vdash M : \tau \wedge M \to_{\beta\eta} M' \implies \Gamma \vdash M' : \tau$$

- Strong normalization:

$$\Gamma \vdash M : \tau \implies M \in \mathrm{SN}$$

2.3 Expressiveness of $\lambda\underline{\omega}$

- $\lambda\underline{\omega}$ has the same expressive power as $\lambda \to$.
- $\lambda\underline{\omega}$ types exactly all extended polynomials.

References

[1] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[2] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[3] Philip Wadler. The girard-reynolds isomorphism.

[4] J.B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111–156, 1999.