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:

$$
\mathcal{T} ::= \ldots \mid \lambda \alpha. \mathcal{T} \mid \mathcal{T}[\mathcal{T}]
$$

Type $::= \ldots \mid \text{TVar} \mid \forall \alpha. \text{Type}$

We add a new rule for $\beta$-reduction

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

We also need typing rules for the new terms.
As an example we can now type self application as

\[ \lambda x : (\forall \alpha. \alpha). (x(\sigma \rightarrow \tau))(x\sigma) : (\forall \alpha. \alpha) \rightarrow \tau \]

1.1 Properties of \( \lambda^2 \)

- Uniqueness of types:
  \[ \Gamma \vdash M : \sigma \land \Gamma \vdash M : \tau \implies \sigma = \tau \]

- Church-Rosser property holds.

- Subject reduction:
  \[ \Gamma \vdash M : \tau \land M \rightarrow_{\beta\eta} M' \implies \Gamma \vdash M' : \tau \]

- Strong normalization:
  \[ \Gamma \vdash M : \tau \implies M \in SN \]

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

\[ \forall \alpha. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \]

The terms corresponds to the Church numerals:

\[ 0 \equiv \lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \rightarrow \alpha. x \]
\[ n \equiv \lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \rightarrow \alpha. f^n x \]
\[ succ \equiv \lambda n : Nat. \lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \rightarrow \alpha. f(n\alpha x f) \]
**Theorem:** $\lambda^2$ types exactly all primitive recursive functions.

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

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

\[
\begin{align*}
A(0, n) &= n + 1 \\
A(m+1, 0) &= A(m, 1) \\
A(m+1, n+1) &= A(m, A(m+1, n))
\end{align*}
\]

- 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

\[
\forall \alpha. \forall \beta. (\alpha \to \beta) \to \alpha
\]

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

We can define some connectives from PROP2:

\[
\begin{align*}
\bot & \equiv \forall \alpha. \alpha \\
\sigma \land \tau & \equiv \forall \alpha. (\sigma \to \tau \to \alpha) \to \alpha \\
\sigma \lor \tau & \equiv \forall \alpha. (\sigma \to \alpha) \to (\tau \to \alpha) \to \alpha \\
\exists \alpha. \sigma & \equiv \forall \beta. (\forall \alpha. \sigma \to \beta) \to \beta
\end{align*}
\]

- 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 λ2 used by languages like Haskell and ML allows for an efficient type inference algorithm due to Hindley and Milner [1, 2].

2 **λω**

The main idea of λω is to extend λ → to let types depend on types, similarly to how λ2 lets terms depend on types. E.g. we would like to express a function \( f \), that we can apply to some type \( \sigma \) to to get the type

\[ f(\sigma) = \sigma \rightarrow \sigma \]

In \( \lambda \omega \) we would express this as

\[ f \equiv \lambda \alpha : \star \cdot \alpha \rightarrow \alpha \]

\( \star \) here is a kind. Intuitively it describes the “type” of types, so \( \alpha : \star \) tells us that \( \alpha \) 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 \( \sigma \) 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

\[ \mathcal{T} ::= \mathcal{V} | \mathcal{C} | \mathcal{T} \mathcal{T} | \lambda \mathcal{V} : \mathcal{T}.\mathcal{T} | \mathcal{T} \rightarrow \mathcal{T} \]

Furthermore we have the sorts

\[ S = \{ \star, \square \} \subseteq \mathcal{C} \]

In a typing judgement \( \Gamma \vdash M : A \), both \( M \) and \( A \) are pseudo-expressions. \( \Gamma \) can now also contain elements like \( \alpha : \star \). This demands that \( \Gamma \) is ordered, since types of other variables in \( \Gamma \) can depend on \( \alpha \). E.g.

\[
\alpha : \star, x : \alpha \vdash x : \alpha \\
\alpha : \star \vdash \lambda x : \alpha. x : \alpha \rightarrow \alpha
\]

are valid typing judgements. However

\[
x : \alpha, \alpha : \star \vdash x : \alpha \\
x : \alpha \vdash \lambda \alpha : \star. x : \star \rightarrow \alpha
\]

are not.

The typing rules for \( \lambda \omega \) is

\[
\begin{align*}
& \text{(ax/sort) } \vdash \star : \square \\
& \text{(weak) } \frac{\Gamma \vdash M : B \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash M : B} \quad \text{if } x \notin \Gamma \\
& \lambda \frac{\Gamma \vdash A \vdash M : B \quad \Gamma \vdash A \rightarrow B : s}{\Gamma \vdash \lambda x : A. M : A \rightarrow B} \\
& \text{(app) } \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \\
& \text{(conv}_\beta \frac{\Gamma \vdash M : A}{\Gamma \vdash N : A} \quad \frac{\Gamma \vdash B : s}{A \equiv_\beta B} \\
& \text{(type/kind) } \frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash A : s \rightarrow B : s} \\
& \text{(var) } \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad \text{if } x \notin \Gamma
\end{align*}
\]

As an example, here is a typing derivation of

\[ \alpha : \star \vdash \lambda x : D\alpha.x : D(D\alpha) \]

where \( D \equiv \lambda \beta : \star. (\beta \rightarrow \beta) \).
2.2 Properties of $\lambda\omega$

- Uniqueness of types:
  \[
  \Gamma \vdash M : \sigma \land \Gamma \vdash M : \tau \implies \sigma = \tau
  \]

- Church-Rosser property holds.

- Subject reduction:
  \[
  \Gamma \vdash M : \tau \land M \rightarrow_{\beta\eta} M' \implies \Gamma \vdash M' : \tau
  \]

- Strong normalization:
  \[
  \Gamma \vdash M : \tau \implies M \in \text{SN}
  \]

2.3 Expressiveness of $\lambda\omega$

- $\lambda\omega$ has the same expressive power as $\lambda \rightarrow$.

- $\lambda\omega$ types exactly all extended polynomials.

References


