60023

Type Systems for Programming Languages

Imperial College London

Contents

Chapter 1

Introduction

Chapter 2

Lambda Calculus

2.1 Introduction to Lambda Calculus

λ-Terms — **Definition 2.1.1**

Given the set of term-variables $\mathcal{V} = \{x, y, z, \dots\}$, a λ-term is defined by the grammar:

$$M, N ::= \underset{variable}{x} \quad | \quad \underset{abstraction}{(\lambda x.M)} \quad | \quad \underset{application}{(M\ N)}$$

We can also describe this using an *inference system*:

$$\frac{}{x \in \Lambda}(x \in \mathcal{V}) \qquad \frac{M \in \Lambda}{(\lambda x.M) \in \Lambda}((x \in \mathcal{V})) \qquad \frac{M \in \Lambda \quad N \in \Lambda}{(M\ N) \in \Lambda}$$

- In a lambda term $M \cdot N$, M is in the *function position* and N is an *argument*
- The leftmost, outer brackets can be ommitted ($M\ N\ (P\ Q) = ((M\ N)\ (P\ Q))$)
- Abstractions can be abbreviated $\lambda xyz.M = (\lambda x.(\lambda y.(\lambda z.M)))$
- Computation is expressed through term substitution.

Free Variables — **Definition 2.1.2**

$$\begin{aligned} fv(x) &= \{x\} \\ fv(\lambda y.M) &= fb(M) \setminus \{y\} \\ fb(M\ N) &= fv(M) \cup fv(N) \end{aligned}$$

A λ-term M is closed if $fv(M) = \emptyset$.

Bound Variables — **Definition 2.1.3**

$$\begin{aligned} bv(x) &= \emptyset \\ bv(\lambda y.M) &= bv(M) \cup y \\ bv(M\ N) &= bv(M) \cup bv(N) \end{aligned}$$

A term with no free variables is *closed*.

We can define term substitution inductively as: Where $P[N/x]$ means replace x by N in λ-term P.

$$\begin{aligned} x[N/x] &= N \\ y[N/x] &= y \\ (P\ Q)[N/x] &= P[N/x]\ Q[N/x] \\ (\lambda y.M)[N/x] &= \lambda y.(M[N/x]) \text{ where } y \neq x \\ (\lambda x.M)[N/x] &= \lambda x.M \end{aligned}$$

This definition can result in variable capture, for example:

$$(\lambda x.y\ x)[y/x] = \lambda x.x\ x$$

Here the free y was substituted for another free variable x, however has been captured by the bound x in the abstraction.

Barendregt's convention — **Definition 2.1.4**

Given some $(\lambda x.M)N$ we can assume:

$$\begin{aligned} x \notin fv(N) \qquad & \text{x is not free in N} \\ \forall y \in bv(M).[y \notin fv(N)] \qquad & \text{All bound variables in } M \text{ are not free in } N \end{aligned}$$

We can always rename the bound variables of a term, this is a fundamental feature to the degree that α-conversion rarely plays a role and terms are considered modulo α-conversion.

Equivalence Relation — **Definition 2.1.5**

A binary relation that is reflexive, symmetric and transitive.

α-Conversion — **Definition 2.1.6**

$$(\lambda x.M)N \to_\alpha (\lambda z.M[z/x])N \text{ where } z \text{ is a new}$$

Renaming bound variables within a term.

α-Equivalence — **Definition 2.1.7**

$$N \to_\alpha M \wedge M \to_\alpha N \Leftrightarrow M =_\alpha N$$

Terms that can be made equal by α-conversion are α-Equivalent

β-Conversion — **Definition 2.1.8**

$$\underset{ReducibleExpression/Redex}{(\lambda x.M)N} \to_\beta \underset{Contractum/Reduct}{M[N/x]}$$

The *one-step* reduction $\to_\beta$ can be defined with contexual closure rules:

$$M \to_\beta N \Rightarrow \begin{cases} \lambda x.M & \to_\beta \lambda x.N \\ P\ M & \to_\beta P\ N \\ M\ P & \to_\beta N\ P \end{cases}$$

$\to *_\beta$ or $\twoheadrightarrow_\beta$ is the transitive closure of $\to_\beta$.
We can also define this using an inference system:

$$(\beta) : \frac{}{(\lambda x.M)N \to_\beta M[N/x]} \quad (\text{Appl-L}) : \frac{M \to_\beta N}{M\ P \to_\beta N\ P} \quad (\text{Appl-R}) : \frac{M \to_\beta N}{P\ M \to_\beta P\ N}$$

$$(\text{Abstr}) : \frac{M \to_\beta N}{\lambda x.M \to_\beta \lambda x.N}$$

$$(\text{Inherit}_r) : \frac{M \to_\beta N}{M \to_\beta^* N} \quad (\text{Refl}) : \frac{}{M \to_\beta^* M} \quad (\text{Trans}_r) : \frac{M \to_\beta^* N \quad N \to_\beta^* P}{M \to_\beta^* P}$$

$$(\text{Inherit}_r) : \frac{M \to_\beta^* N}{M =_\beta N} \quad (\text{Symm}) : \frac{M =_\beta N}{N =_\beta M} \quad (\text{Trans}_{eq}) : \frac{M =_\beta N \quad N =_\beta P}{M =_\beta P}$$

$=_\beta$ is the equivalence relation generated by $\to_\beta^*$:

$$M \to_\beta^* N \quad \Rightarrow M =_\beta N$$

As $=_\beta$ is an equivalence relation we also have:

$$\begin{array}{ll} M =_\beta N & \Rightarrow N =_\beta M \\ M =_\beta N \wedge N =_\beta P & \Rightarrow M =_\beta P \end{array}$$

β-reduction is confluent/satisfies the Church-Rosser property:

$$\forall N, M, P.[M \to_\beta^* N \wedge M \to_\beta^* P \Rightarrow \exists Q.[N \to_\beta^* Q \wedge P \to_\beta^* Q]]$$

β-conversion does not conform to *Barendregt's convention*, for example:

$$\begin{array}{lll} (\lambda xy.xy)(\lambda xy.xy) & \to (\lambda xy.xy)[(\lambda xy.xy)/x] & = \lambda y.(\lambda xy.xy)y \\ & \to \lambda y.(\lambda xy.xy)[y/x] & = \lambda y.(\lambda y.yy) \end{array}$$

We can avoid this by alpha converting the term to $\lambda y.(\lambda xz.xz)y$ before β-conversion.

η-Reduction — **Definition 2.1.9**

$$\text{Given } x \notin fv(M) \text{ then } \lambda x.M\ x \to_\eta M$$

η-reduction can be used for eta equivalence. If $f\ x = g\ x$ then we can eta reduce both to $f = g$.

- Eta reduction is a common lint provided hy hlint for haskell.

2.2 Reduction Strategies

Evaluation Context **Definition 2.2.1**

A term with a single hole $\lceil\rfloor$:

$$C ::= \lceil\rfloor \mid C\ M \mid M\ C \mid \lambda x.C$$

$C\lceil M\rfloor$ is the term obtained from context C by replacing the *hole* $\lceil\rfloor$ with M.

- This allows any variables to be captured.

The one step β-reduction rule can be defined for any evaluation context as:

$$C_N\lceil(\lambda x.M)N\rfloor \to C_N\lceil M[N/x]\rfloor$$

2.2.1 Head Reduction

$$\frac{}{(\lambda x.M)N \to_H M[N/x]} \qquad \frac{M \to_H N}{\lambda x.M \to_H \lambda x.N} \qquad \frac{M \to_H N}{M\ P \to_H N\ P}$$

Reduce the leftmost term, if this is an abstraction, reduce the inside of the abstraction.

2.2.2 Call By Name / Lazy

$$\frac{}{(\lambda x.M)N \to_N M[N/x]} \qquad \frac{M \to_N N}{M\ P \to_N N\ P}$$

Reduce the leftmost term. Do not reduce unless a term is applied (lazy evaluation).

We can also express reduction strategy with an evaluation context:

$$C_N ::= \lceil\rfloor \mid C_N M \qquad \text{where } \to_\beta^N \text{ is defined as } C_N\lceil(\lambda x.M)N\rfloor \to C_N\lceil M[N/x]\rfloor$$

Note that there is only ever one redex to contract.

2.2.3 Call By Value

Given V denotes abstractions and variables (values):

$$\frac{}{(\lambda x.M)V \to_V M[V/x]} \qquad \frac{M \to_V N}{M\ P \to_V N\ P} \qquad \frac{M \to_V N}{V\ M \to_V V\ N}$$

We can apply values, the leftmost term that is not a value is reduced first.

We can also express reduction strategy with an evaluation context:

$$C_V ::= \lceil\rfloor \mid C_V M \mid V C_V \qquad \text{where } \to_\beta^V \text{ is defined as } C_V\lceil(\lambda x.M)V\rfloor \to C_V\lceil M[V/x]\rfloor$$

Note that there is only ever one redex to contract.

2.2.4 Normal Order

$$\frac{}{(\lambda x.M)N \to_N M[N/x]} \qquad \frac{M \to_N N}{M\ P \to_N N\ P} \qquad \frac{M \to_N N}{P\ M \to_N P\ N}(P \text{ contains no redexes}) \qquad \frac{M \to_N N}{\lambda x.M \to_N \lambda x.N}$$

Reduce the leftmost term until it contains no redexes (then continue to other terms), can reduce the inside of an abstraction.

2.2.5 Applicative Order

$$\frac{}{(\lambda x.M)N \to_A M[N/x]}(M, N \text{ contain no redexes}) \qquad \frac{M \to_A N}{M\ P \to_a N\ P}$$

$$\frac{M \to_A N}{P\ M \to_A P\ N}(P \text{ contains no redex}) \qquad \frac{M \to_A N}{\lambda x.M \to_A \lambda x.N}$$

2.2.6 Computability

SKI Combinator Calculus — **Definition 2.2.2**

$$\mathcal{S} = \lambda xyz.xz(yz) \qquad \mathcal{K} = \lambda xy.x \qquad \mathcal{I} = \lambda x.x$$

Any operation in lambda calculus can be encoded (by *abstraction elimination*) into the SKI calculus as a binary tree with leaves of symbols $\mathcal{S}$, $\mathcal{K}$ & $\mathcal{I}$.

It is possible to encode all Turing Machines within *lambda*-calculus and vice versa. This makes λ-calculus (along with Turing Machines) a model for what is computable.

Church-Turing thesis — ***Extra Fun!* 2.2.1**

The Church-Turing thesis equivocates the computational power of Turing machines and the lambda calculus. (Wikipedia)

It is possible to write terms that do not terminate under β-reduction:

$$(\lambda x.xx)\ (\lambda x.xx) \to_\beta (xx)[(\lambda x.xx)/x] = (\lambda x.xx)\ (\lambda x.xx)$$

We can also apply functions continuously.

$$\begin{aligned}
\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)) &\to_\beta \lambda f.(f(x\ x))[(\lambda x.f(x\ x))/x] && = \lambda f.f((\lambda x.f(x\ x))(\lambda x.f(x\ x))) \\
&\to_\beta \lambda f.f(f((\lambda x.f(x\ x))(\lambda x.f(x\ x)))) \\
&\to_\beta \lambda f.f(f(f((\lambda x.f(x\ x))(\lambda x.f(x\ x))))) \\
&\vdots \\
&\to_\beta \lambda f.f(f(f(f(f(\ldots)))))
\end{aligned}$$

This term is a *fixed point constructor*.

Fixed-Point Theorem — **Definition 2.2.3**

$$\forall M.\exists N.[M\ N =_\beta N]$$

Take $N = Y\ M$ where $Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$:

$$\begin{aligned}
Y\ M &\triangleq \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))\ M \\
&\to_\beta (\lambda x.M(x\ x))(\lambda x, M(x\ x)) \\
&\to_\beta (\lambda x.M(x\ x))(\lambda x, M(x\ x)) \\
M(Y\ M) &\triangleq M(\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))\ M) \\
&\to_\beta M((\lambda x.M(x\ x))(\lambda x, M(x\ x)))
\end{aligned}$$

Hence $M(Y\ M) =_\beta Y\ M$ meaning that Y is the fixed point constructor of M

2.3 Normal Forms

Normal Form — **Definition 2.3.1**

A λ-term is in normal form if it does not contain a *redex*.

$$N ::= x \mid \lambda x.N \mid xN_1 \ldots N_n \text{ where } (n \geq 0)$$

No β or η reductions are possible

Head Normal Form — **Definition 2.3.2**

A λ-term is in head normal form if it is an abstraction with a body that is not *reducible*.

$$H ::= x \mid \lambda x.H \mid xM_1 \dots M_n \text{ where } n \geq 1 \wedge M_i \in \Lambda$$

This will mean the term is of the form x or $\lambda x_1 \dots x_n.yM_1 \dots M_m$

- y is the *head-variable*
- If a term has a head-normal form, then head-reduction on the term terminates.

Head Normalisable — **Definition 2.3.3**

A term M is head normalisable if it has a head-normal form.

$$M \to_\beta^* N \text{ where } N \text{ is in head normal form}$$

Strongly Normalisable — **Definition 2.3.4**

A term M is strongly normalisable if all reduction sequences starting from M are finite.

Meaningless — **Definition 2.3.5**

A term without a head-normal form is meaningless as it can never interact with any context (can never apply it to some argument).

Normal Forms — **Example Question 2.3.1**

Determine the normality of the following terms:

1. $\lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))$
2. $(\lambda x.x\ x)\ (\lambda x.x\ x)$
3. $\mathcal{S}\ \mathcal{K}$
4. $(\lambda ab.b)\ ((\lambda x.x\ x)\ (\lambda x.x\ x))$

---

1. Not in either head normal form or normal form (contains a redex).

$$\begin{aligned} &\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x)) \\ \to_\beta &\lambda f.f((\lambda x.f(x\ x))\ (\lambda x.f(x\ x))) \end{aligned}$$

   However the β-reduction is in head normal form (head-variable is f).

2. It is a redex, so its not in a normal form. It does not have a normal form as it reduces to itself, so all reducts contain a redex. It has no head-normal form.

3. Hence the original λ-term is not normal form, but it can be normalised.

$$\begin{array}{lll} & \mathcal{S}\ \mathcal{K} & \text{Must expand } \mathcal{S} \text{ and } \mathcal{K} \\ = & (\lambda xyz.xz(yz))\ (\lambda xy.x) & \text{Is a redex} \\ \to_\beta & (\lambda xyz.xz(yz))\ (\lambda xy.x) & \text{We rename } y \text{ as per barendregt's convention} \\ =_\alpha & (\lambda xyz.xz(yz))\ (\lambda xa.x) & \\ \to_\beta & (\lambda yz.(\lambda xa.x)z(yz)) & \\ \to_\beta & (\lambda yz.(\lambda a.z)(yz)) & \\ \to_\beta & (\lambda yz.z) & \end{array}$$

   As all possible redexes are contracted it is *strongly normalisable*.

4. Contracting the outermost redex results in normal form ter $\lambda b.b$. However contracting the inner term yields itself. Hence it is normalisable, but not *strongly normalisable*.

2.4 Approximation Semantics

There are many methods of describing the semantics of the λ-calculus.

- Reduction rules with *operational semantics*
- set theory with *denotational semantics*

The approach studied in this module defines semantics in a denotational style, but using a reduction system for its definition.

We introduce an extension to the λ-calculus syntax by adding the constant $\bot$,

- $\bot$ means unknown/meaningless/no information
- used to mask sub-terms (typically containing redexes) to allow us to focus on the the *stable* parts of the term that do not change under reduction.

The set of $\Lambda\bot$-terms is defined as:

$$M, N ::= z \mid \bot \mid \lambda x.M \mid M\ N$$

β-reduction is extended to $\to_\bot$ to include:

$$\lambda x.\bot \to_\bot \bot \quad \text{and} \quad \bot M \to_\bot \bot$$

Note that $\lambda x.\bot$ is considered a redex.

The set of normal forms of $\Lambda\bot$ with respect to $\to_\bot$ is the set $\mathcal{A}$:

$$A ::= \bot \mid \lambda x.A\ (A \neq \bot) \mid xA_1 \ldots A_n$$

Approximant — **Definition 2.4.1**

An approximant is a redex-free $\Lambda\bot$-normal forms that can contain $\bot$ and are used to represent finite parts of potentially infinitely large λ-terms in head-normal form.

The partial order $\sqsubseteq \subseteq (\Lambda\bot)^2$ is defined as the smallest pre-order (reflexive and transitive) such that:

$$\begin{array}{lll} \bot \sqsubseteq M & M \sqsubseteq M' & \Rightarrow \quad \lambda x.M \sqsubseteq \lambda x.M' \\ x \sqsubseteq x & M_1 \sqsubseteq M_1' \wedge M_2 \sqsubseteq M_2' & \Rightarrow \quad M_1M_2 \sqsubseteq M_1'M_2' \end{array}$$

- For $A \in \mathcal{A}, M \in \Lambda$, if $A \sqsubseteq M$ then A is the *direct approximant* of M
- The set of *approximants* of $M, \mathcal{A}(M)$ is defined as:

$$\mathcal{A}(M) \triangleq \{A \in \mathcal{A} | \exists M' \in \Lambda.[M \to_\beta^* M' \wedge A \sqsubseteq M']\}$$

- If A is a *direct approximant* of M, then A and M have the same structure, but some parts A contains $\bot$ ($\bot$ masking part of M).
- Redexes in M are masked by $\bot$ in A ($\bot$ masks the redex, or a larger location that contains the redex).

Direct Approximants — **Example Question 2.4.1**

Show the direct approximants for each reduction step of:

1. $\mathcal{S}\ \mathcal{K}$
2. $\mathcal{S}\ a\ \mathcal{K}$

1.

$$\mathcal{S}\ \mathcal{K} = \underset{\{\bot\}}{(\lambda xyz.xz(yz))\ (\lambda ab.a)} \to_\beta \underset{\{\bot\}}{\lambda yz.(\lambda ab.a)z(yz)} \to_\beta \underset{\{\bot\}}{\lambda yz.(\lambda b.z)(yz)} \to_\beta \underset{\{\bot, \lambda yz.z\}}{\lambda yz.z}$$

2.

$$\begin{array}{lll} \mathcal{S}\ a\ \mathcal{K} = & (\lambda xyz.xz(yz))\ a\ (\lambda cd.c) & \{\bot\} \\ \to_\beta & (\lambda yz.az(yz))\ (\lambda cd.c) & \{\bot\} \\ \to_\beta & (\lambda z.az((\lambda cd.c)z)) & \{\bot, \lambda z.a\bot\bot, \lambda z.az\bot\} \\ \to_\beta & (\lambda z.az(\lambda d.z)) & \{\bot, \lambda z.a\bot\bot, \lambda z.az\bot, \lambda a\bot(\lambda d.z), \lambda az(\lambda d.z)\} \end{array}$$

Some basic approximants are:

$$\begin{aligned}
\mathcal{A}(\lambda x.x) &= \{\bot, \lambda x.x\} \\
\mathcal{A}(\lambda x.x\ x) &= \{\bot, \lambda x.x\bot, \lambda x.x\ x\} \\
\mathcal{A}(\lambda x.x((\lambda y.yy)(\lambda y.yy))) &= \{\bot, \lambda x.x\bot\} \\
\mathcal{A}(\mathcal{S}) = \mathcal{A}(\lambda xyz.xz(yz)) &\ \{\bot, \lambda xyz.x\bot\bot, \lambda xyz.x\bot(y\bot), \lambda xyz.x\bot(yz), \lambda xyz.xz\bot, \lambda xyz.xz(y\bot), \lambda xyz.xz(yz)\} \\
\mathcal{A}(\lambda f.(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))) &\ \{\bot, \lambda f.f(\bot), \lambda f.f(f(\bot)), \lambda f.f(f(f(\bot))), \dots\}
\end{aligned}$$

2.4.1 Properties of Approximants

$$(A \in \mathcal{A}(xM_1 \dots M_n) \wedge A \neq \bot \wedge A' \in \mathcal{A}(N)) \Rightarrow AA' \in \mathcal{A}(xM_1 \dots M_n N)$$

Given A is in the approximants of some variable x are lambda terms $M_1 \dots M_n$, and A' in the approximants of N, then AA' is in the approximants of $A\ A'$ (Applying A to A').

$$(A \in \mathcal{A}(Mz) \wedge z \notin fv(M)) \Rightarrow \begin{pmatrix} & A = \bot \\ \vee & A \equiv A'z \text{ where } z \notin fv(A') \wedge A' \in \mathcal{A}(M) \\ \vee & \lambda x.A \in \mathcal{A}(M) \end{pmatrix}$$

If A is an approximant of Mz, and z is not free in M, then either:

- A is $\bot$
- A is some $A'z$, hence be η-reduction, we can see $A' \in \mathcal{A}(M)$ (the z part can be disregarded, and just look at approximates of M).

$$A \sqsubseteq M \wedge M \to_\beta^* N \Rightarrow A \sqsubseteq N$$

If A is ordered before M, and M β-reduces to N, then A is also before N.

$$A \in \mathcal{A}(M) \wedge M \to_\beta^* N \Rightarrow A \in \mathcal{A}(N) \qquad A \in \mathcal{A}(N) \wedge M \to_\beta^* N \Rightarrow A \in \mathcal{A}(M)$$

UNFINISHED!!!

Chapter 3

Curry Type Assignment

Type assignment follows the syntactic structure of terms. For example $\lambda x.M$ will be of the form $A \to B$ where the input x is of type A, and M is of type B.

$\mathcal{T}_C$ is the set of *types*.

- This is ranged over by $A, B \ldots$ and defined over the set of *type variables* Φ.
- The set of *type variables* Φ is ranged over by φ

$$A, B ::= \varphi \mid (A \to B)$$

A type can be either some type variable (some type e.g Int), or a function converting one type to another.

Statement **Definition 3.0.1**

An expression of the form $M : A$ where $M \in \Lambda$ and $A \in \mathcal{T}_c$.

- M is the *subject*
- A is the *predicate*

Context **Definition 3.0.2**

A context Γ is a set of statements with distinct variables as subjects.

- $\Gamma, x : A$ is shorthand for $\Gamma \cup \{x : A\}$ where x does not occur as a subject in Γ (variables must be distinct).
- $x : A$ is shorthand for $\emptyset, x : A$.
- $x \in \Gamma$ is shorthand for $\exists A \in \mathcal{T}_C.[x : A \in \Gamma]$, likewise, if x is not typed in the context we use $x \notin \Gamma$.

For example:

$$\Gamma_{\text{my context}} = \{x : A, y : B, c : B\}$$

$\to$ is used for function types, it is right associative, so:

$$(A \to B) \to C \to D \equiv (A \to B) \to (C \to D)$$

3.0.1 Curry Type Assignment

$$(Ax) : \frac{}{\Gamma, x : A \vdash_C x : A} \qquad (\to I) : \frac{\Gamma, x : A \vdash_C M : B}{\Gamma \vdash_C \lambda x.M : A \to B}(x \notin \Gamma) \qquad (\to E) : \frac{\Gamma \vdash_C M_1 : A \to B \quad \Gamma \vdash_C M_2 : A}{\Gamma \vdash_c M_1 M_2 : B}$$

- We can extend barendregt's convention to ommit the side-condition on $\to I$ by adding the assertion that:

$$\Gamma \vdash M : A \text{ we ensure } \forall x \in bv(M).[x \notin \Gamma]$$

- The definition provided is *sound*:

$$(\Gamma \vdash_c M : A) \wedge (M \to_\beta^* N) \Rightarrow \Gamma \vdash_C N : A$$

Some terms are not typeable under this definition, as self-application is not possible:

- $\lambda x.x\ x$ is not typeable, neither is $\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$
- Type assignment rules do not cover approximants, and hence they are not typeable.

Self Application — **Example Question 3.0.1**

Is it possible to type *self-application* $x\ x$?

We can attempt to use the inference system, however run into a contradiction:

$$\frac{\dfrac{}{\Gamma, x : A \to B \vdash_C x : A \to B}(Ax) \qquad \dfrac{}{\Gamma, x : A \vdash_C x : A}(Ax)}{\Gamma, x :? \vdash_C x\ x : B}(\to E)$$

Hence we need a type such that $A \to B = A$.

3.0.2 Important Lemmas For Type Assignment

Term Substitution

$$\exists C.[(\Gamma, x : C \vdash_C M : A) \wedge (\Gamma \vdash_C N : C)] \Rightarrow \Gamma \vdash_C M[N/x] : A$$

Free Variables

$$\Gamma \vdash_C M : A \wedge x \in fv(M) \Rightarrow \exists B \in \mathcal{T}_C.[x : B \in \Gamma]$$

All free variables in M are typed.

Weakening

$$\Gamma \vdash_C M : A \wedge \Gamma' \text{ is such that } \forall x : B \in \Gamma'.[x : B \in \Gamma \vee (x \notin fv(M) \wedge x \notin bv(M)) \Rightarrow \Gamma' \vdash_C M : A]$$

We can create a new context Γ' that types variables$x, y, z, dots$. If for every variable in the context Γ' it is either not in Γ, or is in Γ with the same type, then we can use Γ' to type M.

Thinning

$$\Gamma, x : B \vdash_C M : A \wedge x \notin fv(M) \Rightarrow \Gamma \vdash_C M : A$$

If a variable is not free in M, then we do not need a type for it.

3.1 Principle Type Property

Principle type theory expresses the idea that a whole family of types could be assigned to a term, however only one is the *principle type*.

Type Substitution — **Definition 3.1.1**

$$(\varphi \mapsto C) : \mathcal{T}_C \to \mathcal{T}_C \text{ where } \varphi \text{ is a type variable and } C \in \mathcal{T}_C$$

Substitution is defined by:

$$\begin{array}{lll} (\varphi \mapsto C) & \varphi & = C \\ (\varphi \mapsto C) & \varphi' & = \varphi' \qquad (\varphi \neq \varphi') \\ (\varphi \mapsto C) & A \to B & = ((\varphi \mapsto C)\ A) \to ((\varphi \mapsto C)\ B) \end{array}$$

Here $(\varphi \mapsto C)$ is a substitution substituting the type variable φ for the type C

$$S_1 \circ S_2 \text{ means } S_1 \circ S_2\ A = S_1(S_2\ A) \qquad S\ \Gamma = \{x : S\ B | x : B \in \Gamma\} \qquad S\langle \Gamma; A\rangle = \langle S\ \Gamma; S\ A\rangle$$

- If there is a substitution S such that $S\ A = B$ then B is the *substitution instance* of A.
- Id_S (*identity substitution*) maps every type variable to itself.

For each typeable term M there is a principal pair:

$$\langle \Pi; P\rangle \text{ where } \Pi \textit{ is a context and } P \in \mathcal{T}_C \text{ such that } \forall \Gamma, A \in \mathcal{T}_C.\exists \text{ substitution } S.[S\langle \Pi; P\rangle = \langle \Gamma; A\rangle]$$

Soundess — **Definition 3.1.2**

A logical system is sound if every formula provable using the system is logically valid according to the semantics of the system.

$$Provable \Rightarrow True$$

Completness — **Definition 3.1.3**

A logical system is complete if any true statement can be proved using the system.

$$True \Rightarrow Provable$$

This definition is sound, for every substitution S:

$$\text{if there is a derivation for } \Gamma \vdash_C M : A \text{ then we can construct a derivation for } S\ \Gamma \vdash_C M : S\ A$$

3.1.1 Unification

Robinson's Unification — **Definition 3.1.4**

$$\begin{array}{llll}
unify & \varphi & \varphi & = (\varphi \mapsto \varphi) \\
unify & \varphi & B & = (\varphi \mapsto B) \text{ given } \varphi \text{does not occur in B} \\
unify & A & \varphi & = unify\ \varphi\ A \\
unify & (A \to B) & (C \to D) & = S_1 \circ S_2 \text{ where} \\
 & & & \quad S_1 = unify\ A\ C \\
 & & & \quad S_2 = unify\ (S_1\ B)\ (S_1\ D)
\end{array}$$

- Unification is associative and commutative
- It returns the most general unifier of two types (the *common substitution instance*)

Robinson's Unification can be generalised to unify contexts.

$$\begin{array}{llll}
unifyContexts & (\Gamma_1, x : A) & \Gamma_2 & = unifyContexts\ \Gamma_1\ \Gamma_2 \text{ given } x \text{ does not occur in } \Gamma_2 \\
unifyContexts & \emptyset & \Gamma_2 & = Id_S \\
unifyContexts & (\Gamma_1, x : A) & (\Gamma_2, x : B) & = S_1 \circ S_2 \text{ where} \\
 & & & \quad S_1 = unify\ A\ B \\
 & & & \quad S_2 = unifyContexts\ (S_1\ \Gamma_1)\ (S_1\ \Gamma_2)
\end{array}$$

3.1.2 Curry Principle Pair

Curry Principle Pair — **Definition 3.1.5**

Every term M has a *(Curry) Principle Pair* defined as $pp_c\ M = \langle \Pi; P\rangle$ by:

$$\begin{array}{lll}
pp_c & x & = \langle x : \varphi; \varphi\rangle \text{ where } \varphi \text{ is fresh} \\
pp_c & \lambda x.M & = \begin{cases} \langle \Pi'; A \to P\rangle & (\Pi = \Pi', x : A) \\ \langle \Pi; \varphi \to P\rangle & (x \notin \Pi) \end{cases} \\
 & & \quad \text{where} \quad \langle \Pi; P\rangle = pp_c\ M \\
 & & \qquad\qquad\ \ \varphi \quad \text{is fresh} \\
pp_c & M\ N & = S_2 \circ S_1 \langle \Pi_1 \cup \Pi_2; \varphi\rangle \\
 & & \quad \text{where} \quad \langle \Pi_1; P_1\rangle = pp_c\ M \\
 & & \qquad\qquad\ \ \langle \Pi_2; P_2\rangle = pp_c\ N \\
 & & \qquad\qquad\ \ S_1 = unify\ P_1\ (P_2 \to \varphi) \\
 & & \qquad\qquad\ \ S_2 = unifyContexts\ (S_1\ \Pi_1)\ (S_1\ \Pi_2) \\
 & & \qquad\qquad\ \ \varphi \quad \text{is fresh}
\end{array}$$

Substitution is complete:

$$\forall \Gamma, M \in \Lambda, A \in \mathcal{T}_c.[\Gamma \vdash_c M : A \Rightarrow \exists \Pi, P \in \mathcal{T}_c, S.[pp_c\ M = \langle \Pi; P\rangle \wedge s\ \Pi \subseteq \Gamma \wedge S\ P = A]]$$

Chapter 4

Polymorphism

We can extend the λ-calculus to allow for functions that are *polymorphic* (can be applied to many different types of inputs).

- We can extend to include names and definitions (e.g $name = M$)
- When type checking we can associate a call to a name with its definition, avoiding the need to re-type check for each call to a function.

4.1 Language Λ^N

Λ^N is Lambda Calculus with names. The syntax is as follows:

$$\begin{aligned} name &::= \text{'A string of characters'} \\ N, M &::= x \mid name \mid \lambda x.N \mid M\ N \\ Defs &::= Defs; name = M | \epsilon \text{ where } M \text{ is closed and name-free} \\ Program &::= Defs : M \end{aligned}$$

Reduction on terms can be defined by an inference system.

$$\frac{}{(\lambda x.M)N \to M[N/x]}$$

Substitution of terms

$$\frac{}{name \to M}(name = M \in Defs)$$

Substituting of names for definitions (inlining)

$$\frac{M \to N}{\lambda x.M \to \lambda x.N} \qquad \frac{M \to N}{M\ P \to N\ P} \qquad \frac{M \to N}{P\ M \to P\ N}$$

Reduction of terms

$$\frac{M \to N}{M \to^* N} \qquad \frac{}{M \to^* M} \qquad \frac{M \to^* N \quad N \to^* P}{M \to^* P}$$

Transitive closure of reduction

$$\frac{M \to N}{Defs : M \to Defs : N}$$

Reduction on Programs

- Names are closed λ-terms (have no free variables).
- If a name is used but not defined, then the program is irreducible.
- Programs written in Λ^N can be translated to Λ by substituting names.

We can translate using the transformation $\langle\cdot\rangle_\lambda : \Lambda^N \to \Lambda$:

$$\begin{aligned} \langle x\rangle_\lambda &= x \\ \langle name\rangle_\lambda &= \begin{cases} \langle M\rangle_\lambda & \text{if } (name = M) \in Defs \\ undefined & otherwise \end{cases} \\ \langle \lambda x.N\rangle_\lambda &= \lambda x.\langle N\rangle_\lambda \\ \langle \lambda N\ M\rangle_\lambda &= \langle \lambda N\rangle_\lambda\ \langle \lambda M\rangle_\lambda \end{aligned}$$

4.2 Type Assignment for Λ^N

By extending Curry's type assignment system for λ-calculus we must consider the types of names in definitions.

Environment — **Definition 4.2.1**

An environment ε is a mapping on $names \to \mathcal{T}_c$.

- Similar to a context, but for names rather than terms.
- $\varepsilon, name : A = \varepsilon \cup \{name : A\}$ where either $name : A \in \varepsilon$ or $name$ does not occur in ε.

$$(Ax) : \frac{}{\Gamma, x : A; \varepsilon \vdash x : A} \qquad (\to I) : \frac{\Gamma, x : A; \varepsilon \vdash N : B}{\Gamma; \epsilon \vdash \lambda x.N : A \to B} \qquad (\to E) : \frac{\Gamma; \varepsilon \vdash P : A \to B \quad \Gamma; \varepsilon \vdash Q : A}{\Gamma; \varepsilon \vdash P\ Q : B}$$

We have extended the curry type inference to include the environment ε.

$$(\epsilon) : \frac{}{\varepsilon \vdash \epsilon : \Diamond}$$

We do not need to consider contexts (definitions use closed terms, no free variables from a context are required to type). $\Diamond$ is not a type, but rather notation of showing there is a type.

$$(Defs) : \frac{\varepsilon \vdash Defs : \Diamond \quad \emptyset; \emptyset \vdash M : A}{\varepsilon, name : A \vdash Defs; name = M : \Diamond}$$

A name can be defined, it must be closed (hence why context is $\emptyset$). Notice this definition ensures definitions are closed and name-free as the rule provides an empty environment and context.

$$(Call) : \frac{}{\Gamma; \varepsilon, name : A \vdash name : S\ A}$$

$$(\text{Program}) : \frac{\varepsilon \vdash Defs : \Diamond \quad \Gamma; \varepsilon \vdash M : A}{\Gamma; \varepsilon \vdash Defs : M : A}$$

4.3 Principal Types

$$\begin{array}{lll}
pp_{\Lambda^N}\ x & \varepsilon & = \langle x : \varphi; \varphi \rangle \text{ where } \varphi \text{ is fresh} \\
pp_{\Lambda^N}\ name & \varepsilon & = \langle \emptyset; FreshInstance(\varepsilon\ name) \rangle \\
pp_{\Lambda^N}\ (\lambda x.M) & \varepsilon & = \begin{cases} \langle \Pi'; A \to P \rangle & (\Pi = \Pi', x : A) \\ \langle \Pi; \varphi \to P \rangle & (x \notin \Pi) \end{cases} \\
 & & \quad \text{where} \quad \langle \Pi; P \rangle = pp_{\Lambda^N}\ M\ \varepsilon \\
 & & \qquad\qquad\ \ \varphi \quad \text{is fresh} \\
pp_{\Lambda^N}\ (M\ N) & \varepsilon & = S_2 \circ S_1 \langle \Pi_1 \cup \Pi_2; \varphi \rangle \\
 & & \quad \text{where} \quad \langle \Pi_1; P_1 \rangle = pp_{\Lambda^N}\ M\ \varepsilon \\
 & & \qquad\qquad\ \ \langle \Pi_2; P_2 \rangle = pp_{\Lambda^N}\ N\ \varepsilon \\
 & & \qquad\qquad\ \ S_1 = unify\ P_1\ P_2 \to \varphi \\
 & & \qquad\qquad\ \ S_2 = unifyContexts\ (S_1\Pi_1)\ (S_1\Pi_2) \\
 & & \qquad\qquad\ \ \varphi \quad \text{is fresh}
\end{array}$$

We also need to define pp_{Λ^N} for definitions.

$$\begin{array}{ll}
BuildEnv\ (Defs; name = M) & = (BuildEnv\ Defs), name : A \text{ where } \langle \emptyset; A \rangle = pp_{\Lambda^N}\ M\ \emptyset \\
BuildEnv\ \epsilon & = \emptyset
\end{array}$$

Hence we can now define:

$$pp_{\Lambda^N}\ (Defs; M) = pp_{\Lambda^N}\ M\ \varepsilon \text{ where } \varepsilon = BuildEnv\ Defs$$

- For each name encountered, the environment is checked to find its principle type, a fresh instance of this type is taken (with all type variables replaced by fresh ones) this allows for polymorphism.

Identity **Example Question 4.3.1**

Derive the Λ^N type for $\lambda x.x$ where it is named Id

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \epsilon : \Diamond}(\epsilon) \quad \dfrac{\dfrac{}{x : \varphi; \emptyset \vdash x : \varphi}(Ax)}{\emptyset; \emptyset \vdash_c \lambda x.x : \varphi \to \varphi}(\to I)}{Id : \varphi \to \varphi \vdash I = \lambda x.x}(Defs) \quad \dfrac{Call_1 \quad Call_2}{\emptyset; Id : \varphi \to \varphi \vdash Id\ Id : A \to A}(\to E)}{\emptyset; Id : \varphi \to \varphi \vdash I = \lambda x.x : Id\ Id : A \to A}(Program)$$

$$Call_1 = \dfrac{}{\emptyset; I : \varphi \to \varphi \vdash I : (A \to A) \to A \to A}(Call)$$

$$Call_2 = \dfrac{}{\emptyset; I : \varphi \to \varphi \vdash I : A \to A}(Call)$$

Chapter 5

Recursion

We can extend Λ^N to include recursion as language Λ^{NR}.

- Definitions can reference their own names, as well as other's names (e.g for mutually recursive functions)

5.1 Language Λ^{NR}

$$\begin{aligned} name &::= \text{'A string of characters'} \\ N, M &::= x \mid name \mid \lambda x.N \mid M\ N \\ Defs &::= Defs; name = M \mid Defs; (rec\ name = M) \mid \epsilon \text{ where } M \text{ is closed} \\ Program &::= Defs : M \end{aligned}$$

The requirement that M be name-free is removed, and a function labeled rec can be recursive.

Y combinator — **Definition 5.1.1**

$$\mathsf{Y} = \lambda f.(\lambda x.fx\ x)\ (\lambda x.fx\ x)$$

Can be used to encode recursion:

$$F = C[F] \to \mathsf{Y}\ (\lambda f.C[f])$$

Factorial — **Example Question 5.1.1**

Write factorial in Λ^{NR} given you can use arithmetic and the $Cond$ function. Then encode it using the Y combinator.

$$Factorial = \lambda n.(Cond\ (n == 0)\ 1\ (n \times (Factorial\ (n-1))))$$

And with the Y combinator:

$$Fac = Y.(\lambda fn.Cond\ (n == 0)\ 1\ (n \times f(n-1)))$$

We cannot directly translate Λ^{NR} to lambda calculus as with Λ^N, and instead must alter recursive functions to make use of the Y combinator.

Y is not typeable under the $\vdash_c$ scheme discussed in these notes, so we must add an extension:

$$M, N ::= \ldots \mid \mathsf{Y} \qquad \mathsf{Y}\ M \to M(\mathsf{Y}\ M) \qquad \frac{}{\Gamma \vdash \mathsf{Y} : (A \to A) \to A}$$

Add Y as a special term in the syntax. Add the reduction rule for Y. Add a type assignment rule.

5.2 Λ^{NR} Type Assignment

$$(Ax) : \frac{}{\Gamma, x : A; \varepsilon \vdash x : A} \qquad (\to I) : \frac{\Gamma, x : A; \varepsilon \vdash N : B}{\Gamma; \epsilon \vdash \lambda x.N : A \to B} \qquad (\to E) : \frac{\Gamma; \varepsilon \vdash P : A \to B \quad \Gamma; \varepsilon \vdash Q : A}{\Gamma; \varepsilon \vdash P\ Q : B}$$

The main 3 typing rules remain unchanged.

$$(Call) : \frac{}{\Gamma; \varepsilon, name : A \vdash name : S\ A} \qquad (Rec\ Call) : \frac{}{\Gamma; \varepsilon, rec\ name : A \vdash name : A}$$

- Call remains the same (still just substitutes the definition)
- A recursive call is added, however the type cannot be substituted for this as the definition internally relies on the type.

$$(Def) : \frac{\varepsilon \vdash Defs : \Diamond \quad \emptyset; \varepsilon \vdash M : A}{\varepsilon, name : A \vdash Defs; name = M : \Diamond} \qquad (Rec\ Def) : \frac{\varepsilon \vdash Defs : \Diamond \quad \emptyset; \varepsilon, rec\ name : A \vdash M : A}{\varepsilon, name : A \vdash Defs; rec\ name = M : \Diamond}$$

- Definitions are no longer name-free and thus we must carry the environment in Def.
- Recursive calls are typed with the same environment.

$$(\epsilon) : \frac{}{\varepsilon \vdash \epsilon : \Diamond} \qquad (\text{Program}) : \frac{\varepsilon \vdash Defs : \Diamond \quad \Gamma; \varepsilon \vdash M : A}{\Gamma; \varepsilon \vdash Defs : M : A}$$

$$\begin{array}{lll}
pp_{\Lambda^{NR}}\ x & \varepsilon & = \langle x : \varphi; \varphi; \varepsilon \rangle \text{ where } \varphi \text{ is fresh} \\
pp_{\Lambda^{NR}}\ name & \varepsilon & = \begin{cases} \langle \emptyset; A; \varepsilon \rangle & (rec\ name : A \in \varepsilon) \\ \langle \emptyset; FreshInstance(\varepsilon\ name); \varepsilon \rangle & (name : A \in \varepsilon) \end{cases} \\
pp_{\Lambda^{NR}}\ (\lambda x.M) & \varepsilon & = \begin{cases} \langle \Pi'; A \to P; \varepsilon' \rangle & (\Pi = \Pi', x : A) \\ \langle \Pi; \varphi \to P; \varepsilon' \rangle & (x \notin \Pi) \end{cases} \\
 & & \quad \text{where} \quad \langle \Pi; P; \varepsilon' \rangle = pp_{\Lambda^{NR}}\ M\ \varepsilon \\
 & & \qquad\qquad \varphi \quad \text{is fresh} \\
pp_{\Lambda^{NR}}\ (M\ N) & \varepsilon & = S_2 \circ S_1 \langle \Pi_1 \cup \Pi_2; \varphi; \varepsilon'' \rangle \\
 & & \quad \text{where} \quad \langle \Pi_1; P_1; \varepsilon' \rangle = pp_{\Lambda^{NR}}\ M\ \varepsilon \\
 & & \qquad\qquad \langle \Pi_2; P_2; \varepsilon'' \rangle = pp_{\Lambda^{NR}}\ N\ \varepsilon \\
 & & \qquad\qquad S_1 = unify\ P_1\ P_2 \to \varphi \\
 & & \qquad\qquad S_2 = unifyContexts\ (S_1\Pi_1)\ (S_1\Pi_2) \\
 & & \qquad\qquad \varphi \quad \text{is fresh}
\end{array}$$

For defs we must modify the *buildEnv* function to use environments:

$$\begin{array}{ll}
BuildEnv\ (defs; name = M)\ \varepsilon & = (BuildEnv\ Defs\ \varepsilon), name : A \text{ where } \langle \emptyset; A; \varepsilon \rangle = pp_{\Lambda^{NR}}\ M\ \varepsilon \\
BuildEnv\ (defs; rec\ name = M)\ \varepsilon & = (BuildEnv\ Defs\ \varepsilon), name : S\ A \\
 & \quad \text{where} \quad \langle \emptyset; A; \varepsilon' \langle = pp_{\Lambda^{NR}}\ M(\varepsilon, rec\ name : \varphi) \\
 & \qquad\qquad S = unify\ A\ B \\
 & \qquad\qquad rec\ name : B \in \varepsilon' \\
 & \qquad\qquad \varphi \quad \text{is fresh} \\
BuildEnv\ \epsilon\ \varepsilon & = \varepsilon \\
pp_{\Lambda^{NR}}(Defs; M) &
\end{array}$$

Hence we can now define $pp_{\Lambda^{NR}}$ for $Defs$:

$$pp_{\Lambda^{NR}}\ (Defs; M) = pp_{\Lambda^{NR}}\ M\ \varepsilon \text{ where } \varepsilon = BuildEnv\ Defs\ \emptyset$$

Chapter 6

Credit

Content

Based on the Type Systems course taught by Dr Steffen van Bakel.

These notes were written by Oliver Killane.