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:

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

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**

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

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

β-Conversion **Definition 2.1.8**

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

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:

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

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

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

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.

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

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*.

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

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 $\perp$,

- $\perp$ 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 $\Lambda\perp$-terms is defined as:

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

β-reduction is extended to $\rightarrow_\perp$ to include:

$$\lambda x.\perp \rightarrow_\perp \perp \quad \text{and} \quad \perp M \rightarrow_\perp \perp$$

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

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

$$A ::= \perp \mid \lambda x.A\ (A \neq \perp) \mid xA_1 \dots A_n$$

Approximant **Definition 2.4.1**

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

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

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

- 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:

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

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

$$\begin{array}{cccccccc} \mathcal{S}\ \mathcal{K} = & (\lambda xyz.xz(yz))\ (\lambda ab.a) & \rightarrow_\beta & \lambda yz.(\lambda ab.a)z(yz) & \rightarrow_\beta & \lambda yz.(\lambda b.z)(yz) & \rightarrow_\beta & \lambda yz.z \\ & \{\perp\} & & \{\perp\} & & \{\perp\} & & \{\perp, \lambda yz.z\} \end{array}$$

2.

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

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.

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

Beta reduction is irrelevant.

$$M_1 \sqsubseteq M \wedge M_2 \sqsubseteq M \Rightarrow M_1 \sqcup M_2 \text{ is defined} \wedge M_1 \sqsubseteq M_1 \sqcup M_2 \wedge M_2 \sqsubseteq M_1 \sqcup M_2 \wedge M_1 \sqcup M_2 \sqsubseteq M$$

$$M =_\beta N \Rightarrow \mathcal{A}(M) = \mathcal{A}(N)$$

Join ($\sqcup$) **Definition 2.4.2**

Join is a partial mapping on $\Lambda\bot$ ($\sqcup : \Lambda\bot \times \Lambda\bot \to \Lambda\bot$):

$$
\begin{aligned}
\bot \sqcup M \equiv M \sqcup \bot &\equiv M \\
x \sqcup x &\equiv x \\
(\lambda x.M) \sqcup (\lambda x.N) &\equiv \lambda x.(M \sqcup N) \\
(M_1\ M_2) \sqcup (N_1\ N_2) &\equiv (M_1 \sqcup N_1)\ (M_2 \sqcup N_2)
\end{aligned}
$$

If $M \sqcup N$ is defined, then M and N are compatible.

- Compatible terms are equal, but with $\bot$ in some locations.
- It is undefined for terms with different structures, e.g (x and $\lambda x.x$)

2.5 Explicit Lambda Calculus

- Substitution in λ-calculus is atomic. $M[N/x]$ replaces all x in M in a single step.
- Substitution is not cost-free in some execution models, hence we may want to make substitution explicit so it can be tracked as part of β-reduction.

Explicit λ-calculus ($\lambda\mathbf{x}$) is defined as:

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid M\langle x := N\rangle$$

For $M\langle x := N\rangle$ occurrences of x in M are bound, and by barendregt's convention x cannot occur (free or bound) in N.

$$\begin{aligned}(\lambda x.M)\ N &\to M\langle x := N\rangle \\ (M\ N)\langle x := L\rangle &\to (M\langle x := L\rangle)(N\langle x := L\rangle) \\ (\lambda y.M)\langle x := L\rangle &\to \lambda y.(M\langle x := L\rangle) \\ x\langle x := L\rangle &\to L \\ M\langle x := L\rangle &\to M \text{ given } (x \notin fv(M))\end{aligned}$$

$$M \to N \Rightarrow \begin{cases} \lambda x.M & \to \lambda x.N \\ M\ L & \to N\ L \\ L\ M & \to L\ N \\ M\langle x := L\rangle & \to N\langle x := L\rangle \\ L\langle x := M\rangle & \to L\langle x := N\rangle \end{cases}$$

If $\to_\beta$ is not applied the $\to_{:=}$ is used. The combination of both reductions for this system is $\to_x$.

$$M \to_\beta N \Rightarrow M \to_x^* N$$

Can reduce anything β-reduction can

$$M \in \Lambda \wedge M \to_x^* N \Rightarrow \exists L \in \Lambda.[N \to_{:=}^* L \wedge M \to_\beta^* L]$$

β-reduction is equivalent to doing all explicit substitutions, then β reducing

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 $A, B \dots$ 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:

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

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:

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

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:

if there is a derivation for $\Gamma \vdash_C M : A$ 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:

$$pp_c\ \ x \quad = \langle x : \varphi; \varphi\rangle \text{ where } \varphi \text{ is fresh}$$

$$pp_c\ \ \lambda x.M \quad = \begin{cases} \langle \Pi'; A \to P\rangle & (\Pi = \Pi', x : A) \\ \langle \Pi; \varphi \to P\rangle & (x \notin \Pi) \end{cases} \\ \text{where} \quad \begin{array}{ll} \langle \Pi; P\rangle & = pp_c\ M \\ \varphi & \text{is fresh} \end{array}$$

$$pp_c\ \ M\ N \quad = S_2 \circ S_1 \langle \Pi_1 \cup \Pi_2; \varphi\rangle \\ \text{where} \quad \begin{array}{ll} \langle \Pi_1; P_1\rangle & = pp_c\ M \\ \langle \Pi_2; P_2\rangle & = pp_c\ N \\ S_1 & = unify\ P_1\ (P_2 \to \varphi) \\ S_2 & = unifyContexts\ (S_1\ \Pi_1)\ (S_1\ \Pi_2) \\ \varphi & \text{is fresh} \end{array}$$

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 $\mathcal{E}$ is a mapping on $names \to \mathcal{T}_c$.

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

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

We have extended the curry type inference to include the environment $\mathcal{E}$.

$$(\epsilon) : \frac{}{\mathcal{E} \vdash \epsilon : \Diamond}$$

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.

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

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.

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

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

4.3 Principal Types for Λ^N

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

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:

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

- 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

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

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

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

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:

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

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:

$M, N ::= \ldots \mid \mathsf{Y}$	$\mathsf{Y}\ M \rightarrow M(\mathsf{Y}\ M)$	$\overline{\Gamma \vdash \mathsf{Y} : (A \rightarrow A) \rightarrow A}$
Add Y as a special term in the syntax.	Add the reduction rule for Y.	Add a type assignment rule.

5.2 Type Assignment for Λ^{NR}

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

The main 3 typing rules remain unchanged.

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

- 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.

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

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

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

5.3 Principle Types for Λ^{NR}

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

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

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

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

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

Chapter 6

Milner's ML

$\mathcal{L}_{ML}$ **Definition 6.0.1**

$\mathcal{L}_{ML}$ is a simple programming language supporting *shallow polymorphic* procedures on a wide variety of objects.

- It is an extension of λ-calculus
- Adds a construct for expressing recursion
- Adds a construct for expressing sub-terms can be used in different ways.

A new type-assignment algorithm is paired with $\mathcal{L}_{ML}$ called $\mathcal{W}$:

- *Semantically Sound* - all typed programs are correct.
- *Syntactically Sound* - if $\mathcal{W}$ accepts a program, then it is well-typed.

6.1 The ML Type Assignment System

ML expressions are of the form:

$$E ::= x \mid c \mid \lambda x.E \mid E_1\ E_2 \mid \text{let } x = E_1 \text{ in } E_2 \mid \text{fix } g.E$$

where:

- x is bound over E_2 in let $x = E_1$ in E_2
- g is bound over E in fix $g.E$
- c is a term constant, such as a number, character or operator

6.1.1 Term Substitution

Term substitution is defined as with the following rules:

$$\begin{aligned} x[E/x] &= x \\ y[E/x] &= y \qquad (y \neq x) \end{aligned}$$

Basic substitution of variables.

$$\begin{aligned} (\lambda y.E')[E/x] &= \lambda y.(E'[E/x]) \\ (E_1\ E_2)[E/x] &= E_1[E/x]\ E_2[E/x] \end{aligned}$$

Substitution of sub-terms.

$$\begin{aligned} (\text{let } x = E_1 \text{ in } E_2)[E/x] &= \text{let } y = E_1[E/x] \text{ in } E_2[E/x] \\ (\text{fix } g.E')[E/x] &= \text{fix } g.E'[E/x] \end{aligned}$$

let statements and fixed point (for recursion)

Note that *barendregt's convention* assumed here.

- The *let* construction is added to cover cases where $(\lambda x.E_1)E_2$ is not typeable but where the contraction $E_1[E_2/x]$ is typeable.
- The *fix* construction introduces model recursion. It is not a combinator, but rather another abstraction mechanism (e.g like λ.).

6.1.2 Reduction

Reduction on $\mathcal{L}ML$ is $\rightarrow_{ML}$ and is defined as an extension of $\rightarrow_{\beta}$, with the additional rules:

$$\begin{aligned}\text{let } x_1 = E_1 \text{ in } E_2 &\rightarrow_{ML} E_2[E_1/x]\\ \text{fix } g.E &\rightarrow_{ML} E[(\text{fix } g.E)/g]\end{aligned}$$

We also add some contextual rules.

$$E \rightarrow_{ML} E' \Rightarrow \begin{cases}\text{let } x = E \text{ in } E_2 & \rightarrow_{ML} \text{let } x = E' \text{ in } E_2\\ \text{let } x = E_1 \text{ in } E & \rightarrow_{ML} \text{let } x = E_1 \text{ in } E'\\ \text{fix } g.E & \rightarrow_{ML} \text{fix } g.E'\end{cases}$$

Under reduction both let $x = E_2$ in E_1 and $(\lambda x.E_1)\ E_2$ are *reducible expressions* and both reduce to $E_1[E_2/x]$

- $(\lambda x.E_1)\ E_2$ semantically interpreted as a function with an operand x
- let $x = E_2$ in E_1 interpreted as a substitution.

Type assignment treats both differently.

6.1.3 Type Assignment

The set of types is defined similarly to with curry types ($\mathcal{T}_c$).

- Extended with type constants $'C$ that includes $\text{int}, \text{bool}, \ldots$.
- Ranged over by type $A, B, \ldots$ much like with $\mathcal{T}_c$.
- Types can be *quantified*, creating *generic types* / *type schemes* ranged over by $\sigma, \tau, \ldots$.

$$\begin{array}{lll} A, B & ::= \varphi \mid c \mid (A \rightarrow B) & \text{(basic types)}\\ \sigma, \tau & ::= A \mid (\forall \varphi.\tau) & \text{(polymorphic types)}\end{array}$$

Types of the form $\forall\varphi.\tau$ are called *quantified types.*

- $(\forall\varphi_1.(\forall\varphi_2.\ldots(\forall\varphi_n.A)\ldots))$ is abbreviated by $\forall\overrightarrow{\varphi}.A$
- φ is bound in $\forall\varphi.\tau$
- Free and bound type variables can be defined just as with variables in λ-calculus, but must have names kept separate.

ML type substitution is defined as:

$$\begin{array}{llll}(\varphi \mapsto C) & \varphi & = C & \\ (\varphi \mapsto C) & c & = c & \\ (\varphi \mapsto C) & \varphi' & = \varphi' & (\varphi' \neq \varphi)\\ (\varphi \mapsto C) & A \rightarrow B & = ((\varphi \mapsto C)A) \rightarrow ((\varphi \mapsto C)B) & \end{array}$$

Basic type substitutions

$$(\varphi \mapsto C) \quad \forall\varphi'.\psi \quad = \forall\varphi'.((\varphi \mapsto C)\psi)$$

Quantified types

Unification is also extended with type constants as:

$$\begin{array}{llll} unify & \varphi & c & = (\varphi \mapsto c)\\ unify & c & \varphi & = unify\ \varphi\ c\\ unify & c & c & = Id_S\end{array}$$

- Here a unification of all other cases including a type constant will fail (e.g cannot unify int and bool)
- Types are considered modulo a kind of α-conversion (similar to barendregt's convention - avoid type variable capture)
- As φ' is bound in $\forall\varphi'.\psi$ we can assume in $(\varphi \mapsto C)\ \forall\varphi'.\psi$ we have $\varphi \neq \varphi'$ and $\varphi' \notin fv(C)$.
- As we can have free type variables, the set of types occurring in $\forall\varphi_1 \ldots \forall\varphi_n.A$ is not necessarily $\{\varphi_1, \ldots, \varphi_n\}$.

$$\overline{\Gamma}\ A = \forall\overrightarrow{\varphi}.A$$

$\overrightarrow{\varphi}$ appear free in A, but are not in the context of A.

For the inference system expressing type assignment, we include a function ν which maps constants to their type (e.g a constant type such as Char, Int or a closed polymorphic type).

$$(Ax) : \frac{}{\Gamma, x : \tau \vdash x : \tau}$$

Basic substitution of free variable

$$(\mathcal{C}) : \frac{}{\Gamma \vdash c : \nu\ c}$$

Substituting constants

$$(\to I) : \frac{\Gamma, x : A \vdash E : B}{\Gamma \vdash \lambda x.E : A \to B} \qquad (\to E) : \frac{\Gamma \vdash E_1 : A \to B \quad \Gamma \vdash E_2 : A}{\Gamma \vdash E_1\ E_2 : B}$$

$$(\text{let}) : \frac{\Gamma \vdash E_1 : \tau \quad \Gamma, x : \tau \vdash E_2 : B}{\text{let } x = E_1 \text{ in } E_2 : B} \qquad (\text{fix}) : \frac{\Gamma, g : A \vdash E : A}{\Gamma \vdash \text{fix } g.E : A}$$

$$(\forall I) : \frac{\Gamma \vdash E : \tau}{\Gamma \vdash E : \forall \varphi.\tau}(\varphi \text{ not free in } \Gamma) \qquad (\forall E) : \frac{\Gamma \vdash E : \forall \varphi.\tau}{\Gamma \vdash E : \tau[A/\varphi]}$$

Quantification is introduced to model substitution operations on types, rasther than replacing all type variables at once.

- $\forall \varphi : \tau$ - all occurrences of type variable φ can be replaced by some basic type.
- The side condition on $\forall I$ ensures that the type variables used do not also occur in the context (there is no reference back to the context).

We can model the substitution of φ in A, by type B as $(\varphi \mapsto B)\ A$.

$$\cfrac{\cfrac{\emptyset \vdash_{ML} E : A}{\emptyset \vdash_{ML} E : \forall \varphi.A}(\forall I)}{\emptyset \vdash_{ML} E : A[B/\varphi]}(\forall E)$$

The let construct corresponds to definitions in A^{NR}.

- Can occur anywhere within a term.
- Given let $x = E_1$ in E_1, E_1 does not need to be a *closed-term*, so it is possible to define terms that are *partially-polymorphic* (a term of type $\forall \vec{\varphi}.A$ where A contains free type variables).
- When applying $\forall I$ only the type variable that we attempt to bind must not occur in the context.

To allow for recursion to be typed, the syntax for fix is added.

- Previously we have seen **Y** added as a typed constant.
- It is also possible to solve this by defining *letrec* as a combination of *let* and *fix* (letrec $g = \lambda x.E_1 in E_2$)

6.1.4 Lemmas for Type Assignment

Free Variables

$$(\Gamma \vdash_{ML} E : \tau \wedge x \in fv(E)) \Rightarrow \exists \sigma.[x : \sigma \in \Gamma]$$

All free variables in some expression must have a type in the context.

Weakening

$$(\Gamma \vdash_{ML} E : \tau \wedge \forall x : \sigma \in \Gamma'[x : \sigma \in \Gamma \vee x \notin (fv(E) \cup bv(E))]) \Rightarrow \Gamma' \vdash_{ML} E : \tau$$

Given some context Γ, any context that extends Γ without adding any of E's variables is equivalent.

Thinning

$$(\Gamma x : \sigma \vdash_{ML} E : \tau \wedge x \notin fv(E)) \Rightarrow \Gamma \vdash_{ML} E : \tau$$

We can remove variables that are not free in E from the context, and the context will still be able to type E.

Generation

$>_\Gamma$ — **Definition 6.1.1**

$$\Gamma \vdash_{ML} E : A[B/\varphi]$$

The smallest reflexive and transitive relation such that:

$$\begin{aligned} \rho &>_\Gamma \forall\varphi.\rho \qquad (\varphi \text{ is not free in } \Gamma \text{ and not bound in } \rho) \\ \forall\varphi.\rho &>_\Gamma \rho[B/\varphi] \end{aligned}$$

Where there are no free φ' in A.

- If $\sigma >_\Gamma \tau$ then τ is a *generic instance* of σ.
- Each context Γ induces a new relation.
- This relation represents applying the $\forall I$ and $\forall E$ steps.

$$\Gamma \vdash_{ML} E : \sigma \wedge \sigma >_\Gamma \tau \Rightarrow \Gamma_{ML} E : \tau$$

$$\begin{array}{lllll}
(1) & \Gamma \vdash_{ML} & x : \sigma & \Rightarrow & \exists x : \tau \in \Gamma. \quad \tau >_\Gamma \sigma \\[1em]
(2) & \Gamma \vdash_{ML} & \lambda x.E : \sigma & \Rightarrow & \exists A, B. \quad \begin{array}{ll} & \Gamma, x : A \vdash_{ML} E : B \\ \wedge & \sigma = \forall \overrightarrow{\varphi_i}.A \to B \\ \wedge & A \to B >_\Gamma \sigma \end{array} \\[1em]
(3) & \Gamma \vdash_{ML} & E_1\ E_2 : \sigma & \Rightarrow & \exists A, B. \quad \begin{array}{ll} & \Gamma \vdash_{ML} E_1 : A \to B \\ \wedge & \Gamma \vdash_{ML} E_2 : A \\ \wedge & B >_\Gamma \sigma \end{array} \\[1em]
(4) & \Gamma \vdash_{ML} & \text{fix } g.E : \sigma & \Rightarrow & \exists A. \quad \begin{array}{ll} & \Gamma, g : A \vdash_{ML} E : A \\ \wedge & \sigma = \sigma = \forall \overrightarrow{\varphi_i}.A \\ \wedge & A >_\Gamma \sigma \end{array} \\[1em]
(5) & \Gamma \vdash_{ML} & \text{let } x = E_1 \text{ in } E_2 : \sigma & \Rightarrow & \exists A, \tau \quad \begin{array}{ll} & \Gamma, x : \tau \vdash_{ML} E_2 : A \\ \wedge & \Gamma \vdash_{ML} E_1 : \tau \\ \wedge & A >_\Gamma \sigma \end{array}
\end{array}$$

System F — ***Extra Fun!* 6.1.1**

The ML type assignment is a restriction on the *polymorphic type discipline* (System F).

- In the ML type assignment covered in these notes, $\forall$ occurs outside of a type (*shallow polymorphism*). It is also decidable.
- In System F $\forall$ is a general type constructor, so $A \to \forall\varphi.B$ is a valid type. It is not decidable.

Complex Types — **Example Question 6.1.1**

Type let $i = \lambda x.x$ in $i\ i$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \varphi \vdash x : \varphi}(Ax)}{\emptyset \vdash \lambda x.x : \varphi \to \varphi}(\to I)}{\emptyset \vdash \lambda x.x : \forall\varphi.\varphi \to \varphi}(\forall I) \qquad \dfrac{(1) \qquad (2)}{i : \forall\varphi.\varphi \to \varphi \vdash i\ i : A \to A}(\to E)}{\emptyset \vdash \text{let } i = \lambda x.x \text{ in } i\ i : A \to A}(\text{let})$$

where:

$$(1) = \dfrac{\dfrac{}{i : \forall\varphi.\varphi \to \varphi \vdash i : \forall\varphi.\varphi \to \varphi}(Ax)}{i : \forall\varphi.\varphi \to \varphi \vdash i : (A \to A) \to A \to A}(\forall E)$$

$$(2) = \cfrac{\overline{i : \forall\varphi.\varphi \to \varphi \vdash i : \forall\varphi.\varphi \to \varphi}(Ax)}{i : \forall\varphi.\varphi \to \varphi \vdash i : A \to A}(\forall E)$$

Addition **Example Question 6.1.2**

Express Addition in ML.

We can use type constants by defining then in ν:

$$\nu\ x = \begin{cases} Num \to Num & x = Succ \\ Num \to Num & x = Pred \\ Num \to Bool & x = IsZero \\ \forall\varphi.Bool \to \varphi \to \varphi \to \varphi & x = Cond \\ \vdots & \vdots \end{cases}$$

We can define it recursively as:

$$Add = \lambda xy.Cond\ (IsZero x)\ y\ (Succ\ (Add\ (Pred x) y))$$

This recursion can be implemented in ML using fix.

$$Add = \text{fix}\ a.\lambda xy.Cond\ (IsZero x)\ y\ (Succ\ (a\ (Pred x) y))$$

6.2 Milner's $\mathcal{W}$

Milner's Type Assignment Algorithm **Definition 6.2.1**

Milner's $\mathcal{W}$ is a type assignment algorithm for ML.

- It has a *principle type property* - given any Γ and E there is a principle type computed by $\mathcal{W}$.
- Its does not have the *principle pair property* as if $\Gamma, x : \tau \vdash_{ML} E : A$ may exist, but $\lambda x.E$ may not be typeable.
- Type assignment is decidable.

It is complete, given some E, contexts Γ and Γ' and type A:

$$\Gamma' \text{ is an instance of } \Gamma \land \Gamma' \vdash_{ML} E : A \Rightarrow \mathcal{W}\ \Gamma\ E = \langle S, B\rangle \land \exists\ S'.\ [\Gamma' = S'(S\ \Gamma) \land S'(S\ B) >_{\Gamma'} A]$$

It is also sound:

$$\forall\ E.\ [\mathcal{W}\ \Gamma\ E = \langle S, A\rangle \Rightarrow S\ \Gamma \vdash_{ML} E : A]$$

6.2.1 Basic Cases

$$\begin{array}{llll} \mathcal{W}\ \Gamma\ c & = \langle id, B\rangle & & \\ & \text{where} & \nu\ c & = \forall\overrightarrow{\varphi}.A \\ & & B & = A[\overrightarrow{\varphi'/\varphi}] \\ & & \text{all } \varphi' & \text{are fresh} \end{array}$$

$$\begin{array}{llll} \mathcal{W}\ \Gamma\ (\lambda x.E) & = \langle S, S\ (\varphi \mapsto A)\rangle & & \\ & \text{where} & \langle S, A\rangle & = \mathcal{W}\ (\Gamma, x : \varphi)\ E \\ & & \varphi & \text{is fresh} \end{array}$$

$$\begin{array}{llll} \mathcal{W}\ \Gamma\ x & = \langle id, B\rangle & & \\ & \text{where} & x : \forall\overrightarrow{\varphi}.A & \in \Gamma \\ & & B & = A[\overrightarrow{\varphi'/\varphi}] \\ & & \text{all } \varphi' & \text{are fresh} \end{array}$$

6.2.2 Let Construct

$$\begin{array}{lll} \mathcal{W}\ \Gamma\ (\text{let } x = E_1 \text{ in } E_2) & = \langle S_2 \circ S_1, B\rangle & \\ & \text{where} \quad \langle S_1, A\rangle & = \mathcal{W}\ \Gamma\ E_1 \\ & \qquad\quad \langle S_2, B\rangle & = \mathcal{W}\ (S_1\Gamma, x : \sigma)\ E_2 \\ & \qquad\quad \sigma & = \overline{S_1\Gamma} A \end{array}$$

1. Get the type and substitutions for E_1 given the context Γ
2. Get the type and substitutions for E_2, the context needs to have E_1's substitutions applied, we add in a new variable x (it will be free in E_2) and give it a $\forall$ type that uses no type variables already bound in $S_1\ \Gamma$.
3. The resulting type for E_2 is the type of the whole term, we must compose the substitutions for E_1 and E_2.

6.2.3 Fix Construct

$$\begin{array}{lll} \mathcal{W}\ \Gamma\ (\text{fix } g.E) & = \langle S_2 \circ S_1, S_2\ A\rangle & \\ & \text{where} \quad \langle S_1, A\rangle & = \mathcal{W}\ (\Gamma, g : \varphi)\ E \\ & \qquad\quad S_2 & = unify\ (S_1\varphi)\ A \\ & \qquad\quad \varphi & \text{is fresh} \end{array}$$

1. g must have the same type as E (recursion, the inner call has the same type as the outer), hence to compute the pair for E we add g to the context with fresh type variable φ
2. We then get a substitution S_1 and type A, we must unify this with the type of g (with S_1 applied) to type the whole term.

6.2.4 Application

$$\begin{array}{lll} \mathcal{W}\ \Gamma\ (E_1\ E_2) & = \langle S_3 \circ S_2 \circ S_1, S_2\varphi\rangle & \\ & \text{where} \quad \langle S_1, A\rangle & = \mathcal{W}\ \Gamma\ E_1 \\ & \qquad\quad \langle S_2, B\rangle & = \mathcal{W}\ (S_1\ \Gamma)\ E_2 \\ & \qquad\quad S_3 & = unify\ (S_2\ A)\ (B \to \varphi) \\ & \qquad\quad \varphi & \text{is fresh} \end{array}$$

1. First the type of E_1 is computed ass type A, with substitutions S_1.
2. Next we get the type of E_2, first applying the substitution S_1 to the context Γ.
3. We now have $E_1 : A$ and $E_2 : B$. A must be equal to some type $B \to \varphi$ (E_1 is a function taking E_2 as input), hence we unify A with $B \to \varphi$

6.3 Polymorphic Recursion

Mycroft generalised Milner's system in an attempt to improve typing for recursively defined objects.

```
map f ls       = if null ls then ls else cons (f (head ls)) (map f (tail ls))
squarelist ls = map (\x -> x^2) ls
```

In Λ^{NR} this would be defined as:

$$\begin{array}{ll} \vdots & \text{(definitions of head, tail, cons, etc)} \\ map & = \lambda f\ ls.Cond\ (null\ ls)\ ls\ (cons\ (f\ (head\ ls))\ (map\ f\ (tail\ ls))) \\ squarelist & = \lambda ls.map\ (\lambda x.mul\ x\ x)\ ls \end{array}$$

The name *squarelist* could then be used in a program.

In ML there is no check to see if functions are independent or mutually recursive, so all definitions must be done in a single step. Hence we can extend $\mathcal{L}_{ML}$ with a pairing function $\langle\ .\ ,\ .\ \rangle$:

let $\langle map, squarelist\rangle = \text{fix}\ \langle m, s\rangle.\langle \lambda f\ ls.Cond\ (null\ ls)\ l\ (cons\ (f\ (head\ ls))\ (m\ f\ (tail\ ls))), \lambda ls.m\ (\lambda x.mul\ x\ x)\ ls\rangle$ in ...

However we still have a type assignment issue, $\mathcal{W}$ will get the following types:

$$\begin{array}{ll} map & :: (num \to num) \to [num] \to [num] \\ squarelist & :: [num] \to [num] \end{array}$$

The definition of map has the type:

$$map :: \forall\varphi_1\varphi_2.(\varphi_1 \to \varphi_2) \to [\varphi_1] \to [\varphi_2]$$

For fix $g.E$ milner's $\mathcal{W}$ unifies the type of E and g, this results in the second type of map not being found by type inference.

One way to avoid this problem is to treat the term as a single definition.

let $map = \text{fix } m.\lambda f\ ls.Cond\ (null\ ls)\ ls\ (cons\ (f\ (head\ ls))\ (m\ f\ (tail\ ls)))$ in let $squarelist = \lambda ls.map\ (\lambda x.mul\ x\ x)\ ls$ in $\ldots$

Instead in Mycroft's system the fix rule is altered.

$$(\text{fix}) : \frac{\Gamma, g : A \vdash E : A}{\Gamma \vdash \text{fix } g.E : A} \qquad\qquad (\text{fix}) : \frac{\Gamma, g : \tau \vdash_{MYC} E : \tau}{\Gamma \vdash_{MYC} \text{fix } g.E : \tau}$$

Milner's **Mycroft's**

Hence the derivation rule allows for type-schemes (the τ) which means different curry types (e.g A) may be used.

6.3.1 Mycroft-Style Assignment for Λ^{NR}

The rules for ϵ, Call, Rec Call, Def and Rec Def can be replaced by the rules:

$$(\epsilon) : \frac{}{\mathcal{E} \vdash \epsilon : \diamond} \qquad (\text{Call}) : \frac{}{\Gamma; \mathcal{E}, name : A \vdash name : S\ A} \qquad (\text{Defs}) : \frac{\mathcal{E}, name : A \vdash Defs : \diamond \quad \emptyset; \mathcal{E}, name : A \vdash M : A}{\mathcal{E}, name : A \vdash Defs; name = M : \diamond}$$

With the principle pair algorithm as:

$$pp_{\Lambda^{RN}}\ x\ \mathcal{E} = \langle x : \mathcal{E}; \mathcal{E}\rangle \text{ where } \mathcal{E} \text{ is fresh}$$

$$pp_{\Lambda^{RN}}\ name\ \mathcal{E} = \langle \emptyset; FreshInstance(\mathcal{E}\ name)\rangle$$

$$pp_{\Lambda^{RN}}\ (\lambda x.M)\ \mathcal{E} = \begin{cases} \langle \Pi'; A \to P\rangle & (\Pi = \Pi', x : A) \\ \langle \Pi; \varphi \to P\rangle & (x \notin \Pi) \end{cases}$$
$$\begin{array}{lll} \text{where} & \langle \Pi; P\rangle & = pp_{\Lambda^{RN}}\ M\ \mathcal{E} \\ & \varphi & \text{is fresh} \end{array}$$

$$\begin{array}{lll} pp_{\Lambda^{RN}}\ (M\ N)\ \mathcal{E} & = S_2 \circ S_1 \langle \Pi_1 \cup \Pi_2; \mathcal{E}\rangle & \\ & \text{where} \quad \langle \Pi_1; P_1\rangle & = pp_{\Lambda^{RN}}\ M\ \mathcal{E} \\ & \qquad\quad\ \langle \Pi_2; P_2\rangle & = pp_{\Lambda^{RN}}\ M\ \mathcal{E} \\ & S_1 \quad = unifyP_1\ (P_2 \to \varphi) & \\ & s_2 \quad = unifyContexts\ (S_1\ \Pi_1)\ (S_1\ \Pi_2) & \\ & \varphi \quad\ \text{is fresh} & \end{array}$$

$$pp_{\Lambda^{RN}}\ (Defs; M)\ \mathcal{E} = \begin{cases} pp_{\Lambda^{RN}}\ M\ \mathcal{E} & \text{if } CheckEnv\ Defs\ \mathcal{E} \\ \text{untypeable} & otherwise \end{cases}$$
$$\begin{array}{lll} \text{where} & CheckEnv\ (Defs; name = M)\ \mathcal{E} & = (CheckEnv\ Defs\ \mathcal{E}) \wedge (\mathcal{E}\ name) = P \\ & & \quad \text{where} \langle \emptyset, P\rangle = pp_{\Lambda^{RN}}\ M\ \mathcal{E} \\ & CheckEnv\ \epsilon\ \mathcal{E} & = true \end{array}$$

6.3.2 Milner's and Mycroft's System's Differences

As Mycroft's system is an extension of Milner's:

Typeable in Milner's $\Rightarrow$ Typeable in Mycroft's

- Many terms are typeable in Mycroft's but not in Milner's
- Some terms can be given more general type in Mycroft's than Milner's

The key difference between the systems is that in Milner's recursive calls use the same curry type, but in Mycroft's these can be a more general type, this allows for polymorphic recursion.

Typeability **Example Question 6.3.1**

Create a term typeable in Mycroft's System but not in Milner's.

$$\text{fix } g.(\lambda(ab.a)\ (g\ \lambda c.c)\ (g\ \lambda de.d))$$

We can write this as:

```
func :: a -> b
func = (\a b -> a) (func (\c -> c))  (func (\lambda d e -> d))
-- or more idomatically
func' = const (func' id) (func' const)

-- in execution this looks like:
func x
  = const (func' id) (func' const) x
  = func' id x
  = const (func' id) (func' const) id x
  = func' id id x
    ...
  = func' id id ... id x
```

This is not typeable in Milner's as here g effectively has two types. Mycroft's allows this as both types can come from the polymorphic type $\forall\varphi_1\varphi_2.\varphi_1 \to \varphi_2$.

Chapter 7

Pattern Matching

Term Rewriting System **Definition 7.0.1**

An extension of lambda calculus allowing for formal parameters to have structure.

- Terms are built out of variables, function symbols and application.
- There is no abstraction, functions are modelled by rewrite rules specifying how terms are modified.

7.1 Syntax

An alphabet/signature consists of a finite, countable set of variables and a non-empty set of function symbols (each with fixed *arity* - number of parameters).

$\mathcal{X} = \{x_1, x_2, \dots\}$	$\mathcal{F} = \{F, G, \dots\}$
Variables	Function Symbols

The set of *terms* $T(\mathcal{F}, \mathcal{X})$ ranged over by t is:

$$t ::= x \mid F \mid (t_1\ t_2)$$

A *replacement* is where a term variable is consistently replaced (corresponds to the substitution of terms in λ-calculus).

$\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$	t^R
A *replacement*	apply R to term t

7.2 Reduction

Rewrite Rule **Definition 7.2.1**

A pair of terms (l, r), often written as a named rule $\mathbf{r} : l \to r$.

Given that $l = F\ t_1 \dots t_n$ for some $F \in \mathcal{F}(\text{arity } n)$ and $t_1, \dots, t_n \in T(\mathcal{F}, \mathcal{X}) \land fv(r) \subseteq fv(l)$

- A *patterns* of a rule are the terms t_t where either t_i is not a variable or it is a variable x and is a free variable in some term t_j.
- A rewrite rule $l \to r$ defines a set of rewrites $l^R \to r^R$ for all replacements R.

l	$\to$	r
redex		contractum

- A *redex* can be substituted by its *contractum* in a context $C\lceil\cdot\rfloor$ for rewrite step $C\lceil t\rfloor \to C\lceil t'\rfloor$
- Rewrite steps can be concatenated into a series $t_0 \to t_1 \to t_2 \to \dots t_n$. We can also write this as $t_0 \to^* t_n$
- If $l \to r$ is a rule, then l is not a variable, or an application starting with a variable (e.g $x\ F$). Hence r cannot introduce new variables

Term Rewriting System (TRS) — **Definition 7.2.2**

$$\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle \text{ of an alphabet } \sum$$

In a rewrite rule $\mathbf{r} : F\ t_1 \dots t_n \to r \in \mathbf{R}$.

- $F \in \mathcal{F}$ is the *defined symbol* of $\mathbf{r}$.
- $\mathbf{r}$ *defines* F.
- For any $Q \in \mathcal{F}$, if a rule defines it, it is a *defined symbol*, otherwise it is a *constructor*.
- TRS is turing-complete, however if lambda calculus is extended to include its pattern matching feature the Church-Rosser property no longer holds (ordering of reduction rules changes the end term / no longer confluent).

Definitions and Examples — **Example Question 7.2.1**

Provide a set of rewrite rules for appending to a list, and mapping over the list.

$$\begin{aligned}
\mathcal{F} &= \{\text{cons}, \text{nil}, \text{append}, \text{map}\} \\
\mathcal{X} &= \{f, x, y, l, l', l''\} \\
\mathbf{R} &= \left\{ \begin{array}{ll}
\text{append nil } l & \to l \\
\text{append (cons } x\ l)\ l' & \to \text{cons } x \text{ (append } l\ l') \\
\text{append (append } l\ l')\ l'' & \to \text{append } l \text{ (append } l'\ l'') \\
\text{map } f \text{ nil} & \to \text{nil} \\
\text{map } f \text{ (cons } y\ l) & \to \text{cons } (f\ y) \text{ (map } f\ l)
\end{array} \right\}
\end{aligned}$$

cons and nil are constructors, map and append are defined functions.

In a term rewriting system defined functions can appear in the terms (as well as the function position F in a rule $F\ t_1 \dots t_n \to r$).

Surjective Pairing — **Example Question 7.2.2**

Is the following a valid TRS?

$$\begin{array}{ll}
\text{In-Left (Pair } x\ y) & \to x \\
\text{In-Right (Pair } x\ y) & \to y \\
\text{Pair (In-Left } x) \text{ (In-Right } x) & \to x
\end{array}$$

It is a valid TRS.

7.3 Type Assignment for TRS

Environment — **Definition 7.3.1**

Given $\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle$ there is environment $\mathcal{E} : \mathcal{F} \to \mathcal{T}_c$

TRS-Context — **Definition 7.3.2**

A set of statements with variables as subjects.

$$(Ax) : \frac{}{\Gamma, x : A; \mathcal{E} \vdash x : A} \qquad (\text{Call}) : \frac{}{\Gamma; \mathcal{E}, F : A \vdash F : S\ A} \qquad (\to E) : \frac{\Gamma; \mathcal{E} \vdash t_1 : A \to B \quad \Gamma; \mathcal{E} \vdash t_2 : A}{\Gamma; \mathcal{E} \vdash t_1\ t_2 : B}$$

Note that (Call) uses a substitution S on the type A. The environment provides the principle type for a function symbol, a substitution can be used to get a specific instance of the principle type.

7.3.1 Principle Pair for a TRS term

Given some TRS $\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle$ and environment $\mathcal{E}$:

$$\begin{array}{ll} pp\ x\ \mathcal{E} & \langle x : \varphi ; \varphi \rangle \text{ where } \varphi \text{ is fresh} \\ \\ pp\ F\ \mathcal{E} & = \langle \emptyset ; FreshInsstance(\mathcal{E}\ F) \rangle \\ \\ pp\ (t_1\ t_2)\ \mathcal{E} & = S\langle \Pi_1 \cup \Pi_2 ; \varphi \rangle \\ & \quad \begin{array}{lll} \text{where} & \langle \Pi_1 ; P_1 \rangle & = pp\ t_1\ \mathcal{E} \\ & \langle \Pi_2 ; P_2 \rangle & = pp\ t_2\ \mathcal{E} \\ & S & = unify\ P_1\ (P_2 \to \varphi) \\ & \varphi & \text{is fresh} \end{array} \end{array}$$

As a context can contain several statements for each variable, there ios not need to unify contexts Π_1 and Π_2 in the principle type of $(t_1\ t_2)$.

Substitution is complete:

$$\Gamma ; \mathcal{E} \vdash t : A \Rightarrow \exists \Pi, P, S. [pp\ t\ \mathcal{E} = \langle \Pi ; P \rangle \wedge S\Pi \subseteq \Gamma \wedge S\ P = A]$$

7.4 Subject Reduction

In order to ensure the subject reduction property, we must only accept rules $l \to r$ that satisfy:

$$\forall R, \Gamma, A. [\Gamma ; \mathcal{E} \vdash l^R : A \Rightarrow \Gamma ; \mathcal{E} \vdash r^R : A]$$

- $l \to r$ with defined symbol F is typeable with respect to $\mathcal{E}$ if there are Π, P and $\mathcal{E}$ such that:

$$pp\ l\ \mathcal{E} = \langle \Pi ; P \rangle \wedge \Pi ; \mathcal{E} \vdash r : P \wedge \text{the leftmost occurrence of } F \text{ is typed with } \mathcal{E}(F)$$

- $\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle$ is typeable with respect to $\mathcal{E}$ if all $r \in R$ are typeable with respect to $\mathcal{E}$

Replacement Lemma

Given $\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle$ is a TRS, with environment $\mathcal{E}$. R is a replacement.

$$pp\ t\ \mathcal{E} = \langle \Pi ; P \rangle \wedge \Gamma ; \mathcal{E} \vdash t^R : A \Rightarrow \exists S.\ [S\ P = A \wedge \forall x : C \in \Pi.\ [\Gamma, \mathcal{E} \vdash x^R : S\ C]]$$

Given the principle type of t, any reduction is this type & context substituted.

$$\Gamma ; \mathcal{E} \vdash t : A \wedge \forall x : C \in \Gamma.\ [\Gamma' ; \mathcal{E} \vdash x^R : C] \Rightarrow \Gamma ; \mathcal{E} \vdash t^R : A$$

If $t : A$ and all variables when typed by Γ' and replaced by R go to C, then $t^R : A$

Subject Reduction Theorem

Given $\langle \mathcal{F}, \mathcal{X}, \mathbf{R} \rangle$ is a TRS, with environment $\mathcal{E}$.

$$\text{All rules in } \mathbf{R} \text{ are typeable} \Rightarrow (\Gamma ; \mathcal{E} \vdash t : A \wedge t \to t' \Rightarrow \Gamma ; \mathcal{E} \vdash t' : A)$$

Reduction does not change the type.

7.5 Combinatory Logic

Combinatory logic is an alternative approach to the λ-calculus to express computability.

- Developed around the same time as λ-calculus was developed by Church.
- A type of applicative TRS.
- Formal parameters of function symbols cannot have structure.
- The right hand side of term rewriting rules can only contain term-variables.

7.5.1 Syntax

Only permitted rules are:

$$\begin{array}{lll} K\ x\ y & \to x & \text{(removal of information)} \\ S\ x\ y\ z & \to x\ z(y\ z) & \text{(distribution of information)} \end{array}$$

Terms are defined as:

$$t ::= K \mid S \mid t_1\ t_2$$

We can also add implicit rule $I = SKK$ as $SKKx \to Kx(kx) \to x$

7.5.2 Extending CL

We can add a bracket abstraction to CL to extend it. By adding the rules on the left, we ca make the optimisations on the right:

$$\begin{array}{lllll|lllll} I & x & & & \to x & & & & & \\ K & x & y & & \to x & S & (Kx) & (Ky) & \to K(xy) \\ S & x & y & z & \to xz(yz) & S & (Kx) & I & \to x \\ B & x & y & z & \to x(yz) & S & (Kx) & y & \to Bxy \\ C & x & y & z & \to xzy & S & x & (Ky) & Cxy \\ W & x & y & & \to xyy & & & & \end{array}$$

Adding these rules means that the structure of combinators is made precise (e.g $S(Kx)\ (Ky) \to K(xy)$ requires terms of structure $k\ t$ as arguments). This effectively adds pattern matching in, as we can now specify precise structures for rules to match.

7.5.3 Type Assignment for CL

Type assignment is done with a basic inference system:

$$(S) : \frac{}{\Gamma \vdash S : (A \to B \to C) \to (A \to B) \to A \to C}$$

The S abstraction's type is *built in*

$$(K) : \frac{}{\Gamma \vdash K : A \to B \to A}$$

The K abstraction's type is *built in*

$$(I) : \frac{}{\Gamma \vdash I : A \to A}$$

I is implied by SKK

$$(Ax) : \frac{}{\Gamma, x : A \vdash x : A}$$

Typing variables

$$(\to E) : \frac{\Gamma \vdash t_1 : A \to B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\ t_2 : B}$$

Application elimination

Chapter 8

Extensions to Type Systems

8.1 Data Structures

Tuples	Structs/Data Classes/Records	Combine data, equivalent to product.
Choice	Enums/Variants/Tagged Unions	Choose between variants of data.

8.1.1 Pairing

We extend the grammar of types to:

$$A, B ::= \ldots \mid A \times B | A + B$$

We can then extend the λ-calculus to:

$$E ::= \ldots \mid \langle E_1 E_2 \rangle \mid left(E) \mid right(E)$$

And can add the following rules to the curry's type assignment system:

$$(Pair) : \frac{\Gamma \vdash E_1 : A \quad \Gamma \vdash E_2 : B}{\Gamma \vdash \langle E - 1, E_2 \rangle : A \times B} \qquad (left) : \frac{\Gamma \vdash E : A \times B}{\Gamma \vdash left(E) : A} \qquad (right) : \frac{\Gamma \vdash E : A \times B}{\Gamma \vdash right(E) : B}$$

The reduction rules for $left$ and $right$ are as follows:

$$\begin{matrix} left\ \langle E_1, E_2 \rangle \to E_1 \\ right\ \langle E_1, E_2 \rangle \to E_2 \end{matrix} \qquad E \to E' \Rightarrow \begin{cases} \langle E', E_2 \rangle & \to \langle E', E_2 \rangle \\ \langle E_1, E \rangle & \to \langle E_1, E' \rangle \\ left(E) & \to left(E') \\ right(E) & \to right(E') \end{cases}$$

Here $left$ and $right$ are constructors (in the same spirit as seem with pattern matching), we could add a rule to reconstruct tuples as below, but this would remove confluence (this is the surjective pairing mentioned in pattern matching).

$$\langle left(E), right(E) \rangle \to E$$

8.1.2 Disjoint Unions

We can then extend the λ-calculus to:

$$E ::= \ldots \mid case(E_1, E_2, E_3) \mid inj \cdot l(E) \mid inj \cdot r(E)$$

Here the unions can only be of two types, and $case$(expression, if A, if B) is a match/case of statement. inj is used to construct the left or right type. And can add the following rules to the curry's type assignment system:

$$(case) : \frac{\Gamma \vdash E_1 : A + B \quad \Gamma \vdash E_2 : A \to C \quad \Gamma \vdash E_2 : B \to C}{\Gamma \vdash case(E_1, E_2, E_3) : C}$$

$$(inj \cdot l) : \frac{\Gamma \vdash E : A}{\Gamma \vdash inj \cdot l(E) : A + B} \qquad (inj \cdot r) : \frac{\Gamma \vdash E : B}{\Gamma \vdash inj \cdot r(E) : A + B}$$

The reduction rules for the new constructors are as follows:

$$\begin{array}{r} case(inj \cdot l(E_1), E_2, E_3) \to E_2\ E_1 \\ case(inj \cdot r(E_1), E_2, E_3) \to E_3\ E_1 \end{array} \qquad E \to E' \Rightarrow \begin{cases} case(E, E_2, E_3) & \to case(E', E_2, E_3) \\ case(E_1, E, E_33) & \to case(E_1, E', E_33) \\ case(E_1, E_2, E) & \to case(E_1, E_2, E') \\ inj \cdot l(E) & \to inj \cdot l(E') \\ inj \cdot r(E) & \to inj \cdot r(E') \end{cases}$$

8.2 Recursive Types

Recursive types are required for many types of data structure (single linked lists, trees, other structures with unbounded size).

Unit Type — **Definition 8.2.1**

A unit type is an empty type containing no data.

- Considered an empty tuple
- Supported by many languages (e.g Rust, Haskell)

To properly express recursive types, many programming languages include a unit type (to be used as the base case for some recursive types).

$$(\text{unit}) : \frac{}{\Gamma \vdash () : \text{unit}}$$

We can extend the grammar of types to:

$$A, B = \dots \mid X \mid \mu X.A$$

$=_\mu$ is the smallest equivalence relation containing:

$$\mu X.A =_\mu A[\mu X.A/X]$$

lists — **Example Question 8.2.1**

Define a singly linked list using the recursive types scheme described in this section.

$$[B] \triangleq \mu X.unit + (B \times X)$$

We can show this using $=_\mu$ as:

$$\begin{array}{rcl} [B] & \triangleq & \mu X.unit + (B \times X) \\ & =_\mu & (unit + (B \times X))[\mu X.unit + (B \times X)/X] \\ & = & (unit + (B \times \mu X.unit + (B \times X))) \\ & = & (unit + (B \times [B])) \end{array}$$

8.2.1 Equi-recursive Approach

$$(\mu) : \frac{\Gamma \vdash E : A}{\Gamma \vdash E : B}(A =_\mu B)$$

Hence we can now type the list as:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash () : \text{unit}}(\text{unit})}{\Gamma \vdash inj \cdot l() : \text{unit} + (B \times [B])}(inj \cdot l)}{\Gamma \vdash inj \cdot l() : [B]}(\mu)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma \vdash E_1 : B} \quad \dfrac{\dots}{\Gamma \vdash E_2 : [B]}}{\Gamma \vdash \langle E_1, E_2 \rangle : B \times [B]}(\text{Pair})}{\Gamma \vdash inj \cdot r\langle E_1, E_2 \rangle : \text{unit} + (B \times [B])}(inj \cdot r)}{\Gamma \vdash inj \cdot r\langle E_1, E_2 \rangle : [B]}(\mu)$$

This approach requires no explicit type annotations or declarations (full type inference is preserved)

Meaningless program have types, for example self application can be typed.

$$\lambda x.x\ x : \mu X.X \to \varphi$$

8.2.2 Iso-recursive Approach

Here syntactic markers for fold and unfold are added, recursive types are folded and unfolded on demand.

Disallows typing of self-application, as is the case with *equi-recursive types*.

The syntax is extended with:

$$E ::= \ldots \mid fold(E) \mid unfold(E)$$

The following reduction rule is added:

$$unfold(fold(E)) \to E \quad E \to E' \Rightarrow \begin{cases} unfold(E) & \to unfold(E') \\ fold(E) & \to fold(E') \end{cases}$$

We then must add the type assignment rules:

$$\text{(fold)} : \frac{\Gamma \vdash E : A[\mu X.A/X]}{\Gamma \vdash fold(E) : \mu X.A} \qquad \text{(unfold)} : \frac{\Gamma \vdash E : \mu X.A}{\Gamma \vdash unfold(E) : A[\mu X.A/X]}$$

$$\cfrac{\cfrac{\cfrac{}{\Gamma \vdash () : \text{unit}}\text{(unit)}}{\Gamma \vdash inj \cdot l() : \text{unit} + (B \times [B])}(inj \cdot l)}{\Gamma \vdash fold(inj \cdot l()) : [B]}\text{(fold)}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\ldots}{\Gamma \vdash E_1 : B} \quad \cfrac{\ldots}{\Gamma \vdash E_2 : [B]}}{\Gamma \vdash \langle E_1, E_2 \rangle : B \times [B]}\text{(Pair)}}{\Gamma \vdash inj \cdot r\langle E_1, E_2 \rangle : \text{unit} + (B \times [B])}(inj \cdot r)}{\Gamma \vdash fold(inj \cdot r\langle E_1, E_2 \rangle) : [B]}\text{(fold)}$$

8.3 Recursive Data Types

Rather than specifically use $\mu.X.A$ for fold/unfold, we can instead generalise to fold/unfold any type. This allows us to build recursive data types.

$$(fold_{\mu X.A}) : \frac{\Gamma \vdash E : A[\mu X.A/X]}{\Gamma \vdash fold_{\mu X.A}(E) : \mu X.A} \qquad (unfold_{\mu X.A}) : \frac{\Gamma \vdash E : \mu X.A}{\Gamma \vdash unfold_{\mu X.A}(E) : A[\mu X.A/X]}$$

We can hence create some identifier for a recursive type, and then use the relevant fold and unfolds for it.

$$A = \text{unit} + (B \times A) \qquad \text{with identifier } [A]$$

More generally a recursive data type is defined as $C\overrightarrow{\varphi} = A_C[\overrightarrow{\varphi}]$.

The syntax can be extended by:

$$E ::= \ldots \mid fold_C(E) \mid unfold_C(E)$$

We also add the reduction rules as:

$$unfold_C(fold_C(E)) \to E \qquad E \to E' \Rightarrow \begin{cases} unfold_C(E) & \to unfold_C(E') \\ fold_C(E) & \to fold_C(E') \end{cases}$$

With type assignbment rules:

$$(fold_C) : \frac{\Gamma \vdash E : A_C[\overrightarrow{B}]}{\Gamma \vdash fold_C(E) : C\overrightarrow{B}} \qquad (unfold_C) : \frac{\Gamma \vdash E : C\overrightarrow{B}}{\Gamma \vdash unfold_C(E) : A_C[\overrightarrow{B}]}$$

for every type definition $C\overrightarrow{\varphi} = A_C[\overrightarrow{\varphi}]$.

Chapter 9

Credit

Content

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

These notes were written by Oliver Killane.