Introduction to Rewriting and Functional Programming

Cours Programmation 1.2, Licence d’Informatique
ENS Cachan, AA 2012-2013
Last update: January 16, 2013

Roberto M. Amadio

Université Paris Diderot (Paris 7)
The setting

An introduction to the foundations of:

- (term) rewriting systems,
- functional programming,
- type theory,

oriented towards syntactic/proof-theoretic methods.
Plan

• Rewriting Systems.
• First-order terms: rewriting and unification.
• Termination.
• Confluence and Completion.
• First-order functional programs.

• The $\lambda$-calculus (higher-order rewriting).
• Confluence and Turing equivalence.
• Simply-typed $\lambda$-calculus.
• Subject reduction and Termination.
• Type inference (for simple and polymorphic types).

• Complements (if time allows).
Strongly related courses

Semester 1

• Programming 1.1.
• Computability and Logic.

Semester 2

• Programming 2.
• $\lambda$-calculus and Computer Science Logic.
• Logic programming.
Contrôle des Connaissances

Soient $P$ la note du partiel et $E$ la note d’examen. La note de la partie 1.2 du cours est déterminée par:

$$(2/3) \cdot E + (1/3) \cdot P$$

- Les transparents sont en anglais mais les sujets seront en français et les réponses pourront être redigées en français ou en anglais.

- Aux épreuves écrites les transparents du cours et vos notes de cours manuscrites sont admises. Tout autre document ou dispositif électronique est interdit.

- Les travaux pratiques suivent et complètent le cours mais ils sont soutenus et notés avec le projet de compilation.
Reduction systems
Plan

• Some motivation.

• A few basic definitions: termination, confluence.

• A few basic facts: Induction principle, Newman’s lemma.
Definition reduction system

A reduction (or rewrite) system is a pair $(A, \rightarrow)$ where $A$ is a set and $\rightarrow \subseteq A \times A$.

NB This is a very general notion that one encounters very frequently in informatics, e.g.:

- to formalise the computation step of an automaton,
- the generation step of a grammar,
- the semantics of a programming language,...
A motivating example

• Suppose we have a set of equations such as:

\[ x + Z = Z + x = x \]
\[ S(x) + y = x + S(y) = S(x + y) \]
\[ (x + y) + z = x + (y + z) \]

which describes the relationships between a ‘zero’ \( Z \), a ‘successor’ \( S \), and an ‘addition’ \(+\).

• A natural attitude is to orient the equations so as to simplify the expression. E.g.

\[ x + Z \rightarrow x \]

• This is not always obvious. For instance, what is the orientation of

\[ S(x) + y = S(x + y) \text{ or } (x + y) + z = x + (y + z) \]
From equations to rewriting rules

Suppose we come out with the following orientation:

\[
\begin{align*}
  x + Z & \rightarrow x \\
  Z + x & \rightarrow x \\
  S(x) + y & \rightarrow S(x + y) \\
  x + S(y) & \rightarrow S(x + y) \\
  (x + y) + z & \rightarrow x + (y + z)
\end{align*}
\]

This is a (term) rewriting system. It is a special case of rewriting system where the relation $\rightarrow$ is represented \textbf{schematically} by a finite set of rules.
Some interesting questions

**Termination** By applying the rules (in any context), do we always reach an expression where no more rules apply?

**Normalisation** A weaker requirement is: is there a reduction strategy that leads to an expression where no more rules apply?

**Normal forms** What is the shape of the terms where no more rules apply?

**Confluence** Suppose an expression $e$ reduces to $e_1$ and $e_2$, Can we always find an expression $e'$ to which both $e_1$ and $e_2$ reduce?
A quick perspective

• In general the questions about termination/normalisation and confluence are **undecidable**.

• However there are **automatic tools** that address these questions and that work in many interesting cases.

• A particularly pleasant case is when the system is **terminating**. Then there is a simple criteria to **check confluence** (a local test is enough).

• If the confluence test fails the system may try to add new rewriting rules (compatible with the initial equations) so as to get confluence while preserving termination. This process is called **completion**.

• When the completion process succeeds we obtain a **decision procedure** for the initial equational theory: two terms are equal iff their normal forms coincide.
‘Demo’ for our example (termination)

Tool used: http://colo6-c703.uibk.ac.at/ttt2/interface/index.php

(VAR x y z)
(RULES
  +(Z,x)  ->  x
  +(x,Z)  ->  x
  +(x,S(y))  ->  S(+x,(y))
  +(S(x),y)  ->  S(+x,y)
  +(+(x,y),z)  ->  +(x,+(y,z))
)

YES...

Time: 0.004470

LPO:
  Precedence:
    + > S

LPO=Lexicographic Path Order, a technique for proving termination.
‘Demo’ for our example (confluence)

Tool used: http://colo6-c703.uibk.ac.at/mkbt6/interface/index.php

% SZS status Success for tmp.trs
0.13 (total time)
...
COMPLETE SYSTEM:
+(Z,x) → x
+(x,Z) → x
+(x,S(y)) → S+(x,y))
+(S(x),y) → S+(x,y))
+(+(x,y),z) → +(x,+(y,z))

(** The same !! **) 

The system is already confluent and so the ‘complete system’ coincides with the initial one.
Another example: group theory (termination)

(VAR x y z)
(RULES
   *(e,x) -> x
   *(x,e) -> x
   *(i(x),x) -> e
   *(x,i(x)) -> e
   *(*(x,y),z) -> *(x,*(y,z))
)

TERMINATION : Yes.

POLY=

POLY=Polynomial interpretation, another technique for proving termination.
Another example: group theory (completion)

COMPLETE SYSTEM:
*(e(),x) -> x
*(x,e()) -> x
*(i(x),x) -> e()
i(e()) -> e()  // new
*(x,i(x)) -> e()
*(*(x,y),z) -> *(x,*y,z)
*(i(x),*(x,y)) -> y  // new
i(i(x)) -> x  // new
*(x,*(i(x),y)) -> y  // new
i(*(x,y)) -> *(i(y),i(x))  // new

The system has discovered derived rules that allow to get confluence while preserving termination.
Terminology

• Fix a reduction relation \((A, \rightarrow)\).

• \(\rightarrow\) is **terminating** if there is no infinite sequence:

\[
a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \cdots
\]

• A **normal form** is an element that cannot be further reduced.

• \(\rightarrow\) is **normalising** if for all \(a \in A\), there is a (finite) reduction sequence to a normal form.

**NB** Terminating implies normalising, but not vice versa. Example:

\[
1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots \text{ and } i \rightarrow 0 \text{ for } i \geq 1
\]
• $\rightarrow$ is confluent if

\[
\forall b, c \ (b \leftrightarrow a \rightarrow c) \\
\exists d \ (b \rightarrow d \leftrightarrow c)
\]

We also write $b \downarrow c$ if $\exists d \ (b \rightarrow d \leftrightarrow c)$.

• Let $\leftrightarrow = (\rightarrow \cup \rightarrow^{-1})^*$. The relation $\rightarrow$ is **Church-Rosser** if $b \leftrightarrow c$ implies $b \downarrow c$.

**NB** Church and Rosser proved this property for the $\lambda$-calculus: two $\lambda$-terms are ‘equivalent’ iff they can be rewritten to a common $\lambda$-term.
Exercise (confluent vs. Church-Rosser)

Show that a reduction relation $\rightarrow$ is **Church-Rosser** iff it is **confluent**.
Exercise (confluence, normalisation, and normal form)

Show that:

1. If a reduction relation $\rightarrow$ is confluent then every element has at most one normal form.

2. Moreover, if $\rightarrow$ is also normalising then every element has a unique normal form.
Decision procedure by rewriting

• Let $\sim$ be an equivalence relation on, say, the natural numbers, a set of terms, ... 

• Let $\rightarrow$ be a reduction relation such that $\sim=\leftrightarrow^*$. 

• Suppose that $\rightarrow$ has an effective normalisation procedure and that the equality of normal forms is decidable. 

• Then to decide $a \sim b$ reduce $a$ and $b$ to normal form and compare them.
Well-founded sets and well-founded induction

• A partial order \((W, >)\) is a set \(W\) with a transitive relation \(>\).

• A partial order \((W, >)\) is well-founded if it is not possible to define a sequence \(\{x_i\}_{i \in \mathbb{N}} \subseteq W\) such that:

\[
x_0 > x_1 > x_2 > \cdots
\]

Note that this fails if there is an \(x\) such that \(x > x\).

• Clearly, every well-founded partial order is a terminating reduction system.

• Conversely, every terminating reduction system \((W, \rightarrow)\) induces the well-founded partial order \((W, \rightarrow^+)\) where \(\rightarrow^+\) is the transitive closure of \(\rightarrow\).
The principle of well-founded induction

The following reasoning principle holds in a well-founded partial order \((W, >)\):

\[
\forall x \in W \ (\forall y < x \ P(y)) \rightarrow P(x)
\]

\[
\forall x \in W \ P(x)
\]

where we abuse notation by writing \(P(x)\) for \(x \in P\).

**Question**  Explain why the principle fails if \(W = \{\ast\}\) is a singleton and \(>\) is reflexive.
Justification of the principle

- If $x$ is minimal then the principle requires $P(x)$.
- Otherwise, suppose $x_0$ is not minimal and $\neg P(x_0)$.
- Then there must be $x_1 < x_0$ such that $\neg P(x_1)$.
- Again $x_1$ is not minimal and we can go on to build:

$$x_0 > x_1 > x_2 > \cdots$$

which contradicts the hypothesis that $W$ is well-founded.
Remark (special case)

On the **natural numbers** the principle is stated as follows:

\[
\forall n \ (\forall n' < n \ P(n')) \rightarrow P(n) \\
\forall n \ P(n)
\]

or equivalently:

\[
P(0) \land (\forall n \ (\forall n' < n + 1 \ P(n')) \rightarrow P(n + 1)) \\
\forall n \ P(n)
\]

This is a variant (and equivalent) to the usual reasoning principle:

\[
P(0) \land (\forall n \ (P(n) \rightarrow P(n + 1))) \\
\forall n \ P(n)
\]
Exercise (on well-founded orders)

Let $\mathbb{N}$ be the set of natural numbers, $\mathbb{N}^k$ the cartesian product $\mathbb{N} \times \cdots \times \mathbb{N}$, $k$-times, and $A = \bigcup \{\mathbb{N}^k \mid k \geq 1\}$.

Let $>$ be a binary relation on $A$ such that:

$$(x_1, \ldots, x_m) > (y_1, \ldots, y_n) \text{ iff } \exists k \ (k \leq \min(n, m), x_1 = y_1, \ldots, x_{k-1} = y_{k-1}, x_k > y_k)$$

Prove or disprove the assertion that $>$ is a well-founded order.
Exercise (from well-founded induction to well-founded set)

We have shown that on a well-founded order the induction principle holds. Conversely, show that:

if the \textit{induction principle} holds on a partial order \((W, >)\)
then \((W, >)\) is \textit{well-founded}.
Local confluence

In general, it is hard to prove confluence because we have to consider arbitrary long reductions. It is much simpler to reason locally.

Definition  A relation $\rightarrow$ is **locally confluent** if

$$
\forall b, c \ (b \leftarrow a \rightarrow c) \\
\exists d \ (b \rightarrow^* d \leftarrow^* c)
$$

Theorem (Newman)  If a reduction system $(A, \rightarrow)$ is **locally confluent** and **terminating** then it is confluent.
Proof of Newman’s lemma

• We apply the principle of **well-founded induction** to \((A, \rightarrow)\)!

• Suppose

\[
\begin{align*}
    c_1 & \leftarrow b_1 \leftarrow a \rightarrow b_2 \rightarrow c_2
\end{align*}
\]
• By local confluence, \( \exists d \ (b_1 \rightarrow d \leftarrow b_2) \).

• By induction hypothesis on \( b_1 \), \( \exists d' \ (c_1 \rightarrow d' \leftarrow d) \).

• By induction hypothesis on \( b_2 \), \( \exists d'' \ (d' \rightarrow d'' \leftarrow c_2) \).

• But then \( c_1 \downarrow c_2 \).

• Thus by the principle of well-founded induction, the reduction relation is confluent.
Exercise (commutation and modular confluence)

• Let $\rightarrow_1$ and $\rightarrow_2$ be two reduction relations.

• We say that they commute if $a \overset{*}{\rightarrow}_1 b$ and $a \overset{*}{\rightarrow}_2 c$ implies $\exists d \ (b \overset{*}{\rightarrow}_2 d$ and $c \overset{*}{\rightarrow}_1 d)$.

• Show that:

  if $\rightarrow_1$ and $\rightarrow_2$ are confluent and commute then $\rightarrow_1 \cup \rightarrow_2$ is also confluent.
Exercise (modular termination)

Let $\to_1$ and $\to_2$ be two reduction relations such that $\to_1 \cup \to_2$ is transitive. Show that if $\to_1$ and $\to_2$ are terminating then their union is terminating too.

**Hint:** use the following basic version of Ramsey theorem.

In an undirected graph with countably many nodes there is an infinite subgraph where all elements are connected or all elements are disconnected.
Ramsey theorem

Let $k, r \geq 1$.

If $X$ is a set, denote with $X^{[k]}$ the parts of $X$ of cardinality $k$.

**Ramsey theorem (1930)** If $f : \mathbb{N}^{[k]} \to \{1, \ldots, r\}$ then there is an infinite subset $X$ of $\mathbb{N}$ such that $f$ is constant over $X^{[k]}$.

**Terminology:** the elements $\{1, \ldots, r\}$ are also called **colours** and the function $f$ a **colouring**.
Example

• An undirected countable graph can be described by a function \( f : \mathbb{N}^2 \rightarrow \{1, 2\} \) where, say, \( f\{i, j\} = 1 \) iff the nodes \( i \) and \( j \) are connected.

• Then Ramsey theorem asserts that there is an infinite subgraph \( X \) where either all nodes are connected or all nodes are disconnected.
Finite version of Ramsey theorem

Let $[n] = \{1, \ldots, n\}$. Given any $k, m, r$ numbers there is an $n$ such that any $r$ colouring of $[n]^{[k]}$ contains a mono-chromatic subset of cardinality $m$ of $[n]$.

Examples

• Of three ordinary people, two must have the same sex.
  Here: $k = 1$, $m = 2$, $r = 2$, and $n = 3$ is the least solution.

• In any collection of six people either three of them mutually know each other or three of them mutually do not know each other.
  Here: $k = 2$, $m = 3$, $r = 2$, and $n = 6$ is the least solution.

NB These problems become quickly very hard and computing/estimating the value of $n$ is a non-trivial business.
Exercise (proof of Ramsey theorem)

It is easy to prove (the infinite case of) Ramsey theorem for $k = 1$ (check this out!).

We outline a proof of Ramsey theorem for the case $k = 2$ (the one used next).

1. Consider a function $f : \mathbb{N}^{[2]} \to \{1, \ldots, r\}$. Show that there is an infinite sequence of infinite sets $X_0 \supset X_1 \supset X_2 \supset \cdots$ such that:

   $$\forall x \in X_{i+1} \ f(\{ \min(X_i), x \}) \text{ is constant}$$

2. For $x \in X_{i+1}$, let $c_i = f(\{ \min(X_i), x \}) \in \{1, \ldots, r\}$. Show that for some $c \in \{1, \ldots, r\}$ there are infinitely many $i$ such that $c_i = c$.

3. Conclude that there is a subsequence $\{x_{\sigma(i)}\}_{i \in \omega}$ such that $\forall i < j \ f(x_{\sigma(i)}, x_{\sigma(j)}) = c$. 
Exercise (counter-examples)

The previous exercises provide some sufficient conditions for modular confluence and termination:

<table>
<thead>
<tr>
<th>$\rightarrow_1, \rightarrow_2$</th>
<th>condition</th>
<th>$\rightarrow_1 \cup \rightarrow_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confluent</td>
<td>$\rightarrow_1, \rightarrow_2$ commute</td>
<td>Confluent</td>
</tr>
<tr>
<td>Terminate</td>
<td>$(\rightarrow_1 \cup \rightarrow_2)$ transitive</td>
<td>Terminates</td>
</tr>
</tbody>
</table>

Find (simple, finite) examples that show that:

1. Dropping the condition the implication fails.
2. The conditions are not necessary.
Exercise (confluence and termination on words)

Let $\Sigma = \{f, g_1, g_2, a\}$ be a signature (symbols with an arity) where $f, g_1, g_2$ are unary symbols and $a$ is a constant.

Let $T_\Sigma$ be the set of closed terms over $\Sigma$ (in this case, this is the same as the words over $f, g_1, g_2$!)

Let $\rightarrow$ be the smallest binary relation on $T_\Sigma$ such that:

\[
\begin{align*}
  f(g_1(t)) & \rightarrow g_1(g_1(f(f(t)))) & \text{for all } t \in T_\Sigma \\
  f(g_2(t)) & \rightarrow g_2(f(t)) & \text{for all } t \in T_\Sigma \\
  f(a) & \rightarrow a
\end{align*}
\]

and such that if $t \rightarrow s$ and $h$ is a unary symbol in $\Sigma$ then $h(t) \rightarrow h(s)$.
Prove or give a counter-example to the following assertions:

1. If \( t \xrightarrow{*} t_1 \) and \( t \xrightarrow{*} t_2 \) then there exists \( s \) such that \( t_1 \xrightarrow{*} s \) and \( t_2 \xrightarrow{*} s \).

2. Every reduction sequence \( t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots \) is finite.

3. If one replaces the rule
\[
    f(g_1(t)) \rightarrow g_1(g_1(f(f(t))))
\]

with the rule
\[
    f(g_1(t)) \rightarrow g_1(g_1(f(t)))
\]

the answers to the questions (1) and (2) are unchanged.
Summary

• The following concepts are ‘equivalent’:
  – Terminating Rewrite System.
  – Well-founded set.
  – Partial order with well-founded induction principle.

• Newman’s lemma:

  (Local confluence + Termination) implies Confluence.
Recommended reading

Baader, Nipkow, *Term rewriting and all that*, Cambridge University Press, chapters 1 and 2.
Terms: rewriting rules and syntactic unification
Plan

We introduce:

- **First-order terms** (in the sense of first-order logic).
- **Term rewriting rules** which give a schematic presentation of a rewriting system.
- An algorithm to solve equations on terms: **syntactic unification**.
Some motivation

• A variety of methods have been developed to prove termination.

• We will introduce some of them in the general framework of term rewriting systems (TRS).

• TRS occur everywhere in symbolic computation and they are a topic of study in its own.

• Termination methods developed for TRS can be specialized for the programs of first-order functional languages.
Term rewriting systems

We introduce term rewriting systems.

- The **signature** \( \Sigma = \{ f_1, \ldots, f_n \} \) is a finite set of symbols \( f_i \) of given arity \( ar(f_i) \in \mathbb{N} \).
- \( V \) is a countable set of **variables**.
- If \( V' \subseteq V \) then \( T_\Sigma(V') \) is the collection of **first order terms** over the set \( V' \) with generic elements \( t, s, \ldots \) (respecting the arity).
- A **substitution** \( S : V \rightarrow T_\Sigma(V) \) is a function which is the identity almost everywhere.
• The substitution extension to $T_\Sigma(V)$ is:

$$S(f(t_1, \ldots, t_n)) = f(S(t_1), \ldots, S(t_n))$$

Thus substitutions can be composed.

• A context $C$ is a term with a special variable $[\ ]$ called hole. $C[t]$ is the term resulting from the replacement of $[\ ]$ by $t$ in $C$.

• A (rewriting) rule is a pair of terms $(l, r)$ that we write

$$l \rightarrow r$$

such that $\text{Var}(r) \subseteq \text{Var}(l)$.

• A set $R$ of rewriting rules on a signature $\Sigma$ induces a rewrite relation $\rightarrow_R$ over $T_\Sigma(V)$ as follows:

$$t \rightarrow_R s \quad \text{if} \quad t = C[Sl] \quad \text{and} \quad s = C[Sr]$$
Example (rewrite relation induced by a TRS)

Take as TRS $R$:

\[
\begin{align*}
  f(x) & \rightarrow g(f(s(x))) \\
  i(0, y, z) & \rightarrow y \\
  i(1, y, z) & \rightarrow z 
\end{align*}
\]

Then, for instance:

\[
\begin{align*}
  f(s(y)) & \rightarrow_R g(f(s(s(y)))) & \rightarrow_R g(g(f(s(s(s(y)))))) & \rightarrow_R \cdots \\
  i(0, 1, f(y)) & \rightarrow_R 1 \\
  i(0, 1, f(0)) & \rightarrow_R i(0, 1, g(f(s(0)))) & \rightarrow_R \cdots 
\end{align*}
\]
Exercise (initial algebra)

Let $\Sigma$ be a signature, we can look at its elements as the operations of an algebra.

To fix the ideas let us assume $\Sigma = \{c^0, g^2\}$, but what follows generalizes to an arbitrary signature.

- A $\Sigma$-algebra is a vector $A = (A, c_A, g_A)$ where $A$ is a set, $c_A \in A$ and $g_A : A \times A \to A$.

- If $A = (A, c_A, g_A)$ and $B = (B, c_B, g_B)$ are two $\Sigma$-algebras, a morphism $h$ from $A$ to $B$ is a function $h : A \to B$ such that $h(c_A) = c_B$ and for all $a, a' \in A$, $h(g_A(a, a')) = g_B(h(a), h(a'))$.

- If $X$ is a set of variables, then $T\Sigma(X)$ is the set of terms over $\Sigma$ that may contain variables in $X$.

- We note that $T\Sigma(x) = (T\Sigma(x), c_T, g_T)$ where $c_T = c$ and $g_T(t, t') = g(t, t')$ is a $\Sigma$-algebra.
1. Show that if $\mathcal{A} = (A, c_A, g_A)$ is a $\Sigma$-algebra then the (set-theoretic) functions from $X$ to $A$ are in bijective correspondence with the morphisms from $\mathcal{T}_\Sigma(X)$ to $\mathcal{A}$.

2. Conclude that there exists a **unique morphism** from $\mathcal{T}_\Sigma(\emptyset)$ to $\mathcal{A}$. Therefore $\mathcal{T}_\Sigma(\emptyset)$ is called the **initial algebra**.

**NB** A more informal way of saying the same thing is that the interpretation of a term in a $\Sigma$-algebra is uniquely determined by an assignment of algebra values to its variables.
A short history of the syntactic unification problem

Syntactic unification is about solving equations on terms (finite labelled trees).

- The notion of syntactic unification already occurs in Herbrand’s thesis (1930).
- An algorithm was later proposed by Robinson in conjunction with the resolution principle (1965).
Some notation and terminology

- We write \( t = s \) if the terms \( t \) and \( s \) are \textit{syntactically equal}.
- We denote with \( \text{Var}(t) \) the set of variables occurring in the term \( t \).
- Since a substitution \( S \) is a function which is the identity almost everywhere we may represent it as a \textbf{finite list}
  \[
  [t_1/x_1, \ldots, t_n/x_n]
  \]
  where \( x_i \neq x_j \) if \( i \neq j \).
- We also denote with \( id \) the \textbf{identity substitution}. 

51
• We define a **pre-order on substitutions** as follows:

\[ R \leq S \iff \exists T \ T \circ R = S \]

Thus \( R \leq S \) if \( S \) is an **instance** of \( R \) or, equivalently, if \( R \) is **more general** than \( S \) (note \( id \leq S \), for any \( S \)).

• A **system of equations** \( E \) is a finite set of pairs

\[ \{t_1 = s_1, \ldots, t_n = s_n\} \]

• A substitution \( S \) **unifies** a system of equations \( E \), written

\[ S \models E \]

if \( St = Ss \) (here \( = \) means identity on \( T_\Sigma(V) \)) for all \( t = s \in E \).

**NB** We are **abusing notation** by using \( = \) both for the identity on terms (semantic level) and for a constraint relation (syntactic level).
Exercise (on substitutions)

1. Show that if $S$ is a substitution unifying the system
   \{ $s_1 = s_2, x = t$ \} then $S$ unifies \{ $[t/x]s_1 = [t/x]s_2$ \} too.

2. Give an example of two substitutions $S, T$ such that:
   \[ S \neq T, \quad S \leq T \text{ and } T \leq S. \]
A basic unification algorithm

A basic algorithm can be presented as a rewrite system over pairs \((E, S)\) and a special symbol \(\perp\) (the symmetric rules for \((vt_i), \ i = 1, 2\) are omitted):

\[(v) \quad (E \cup \{x = x\}, S) \quad \rightarrow (E, S)\]
\[(vt_1) \quad (E \cup \{x = t\}, S) \quad \rightarrow ([t/x]E, [t/x] \circ S) \quad x \notin \text{Var}(t)\]
\[(vt_2) \quad (E \cup \{x = t\}, S) \quad \rightarrow \perp \quad x \neq t, x \in \text{Var}(t)\]
\[(f_1) \quad (E \cup \{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\}, S) \quad \rightarrow (E \cup \{t_1 = s_1, \ldots, t_n = s_n\}, S)\]
\[(f_2) \quad (E \cup \{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\}, S) \quad \rightarrow \perp \text{ if } f \neq g\]

NB This ‘abstract’ presentation of the algorithm is instrumental to the proof of its properties: we transform the system leaving the set of its solutions unchanged (cf. Gaussian elimination).
Example (unification)

Applying the unification method to the system

\[ \{ f(x) = f(f(z)), g(a, y) = g(a, x) \} \]

leads to the substitution

\[ S = [f(z)/y] \circ [f(z)/x] \]
Exercise (unification algorithm)

Apply the unification algorithm to the following systems of equations:

\[ \{ f(x, f(x, y)) = f(g(y), f(g(a), z)) \}, \quad (a \text{ is a constant}) \]
\[ \{ f(x, f(y)) = f(y, f(f(x))) \} \]
Properties of the algorithm

1. The reduction relation $\rightarrow$ terminates.

2. If $(E, id) \rightarrow^* (\emptyset, S)$ then $S$ unifies $E$.

3. If $T$ unifies $E$ then all reductions starting from $(E, id)$ terminate with some $(\emptyset, S)$ such that $S \leq T$. 
(1) Proof of termination

• Define a measure on a set of equations as $\mu(E) = (m, n)$ where pairs are lexicographically ordered, $m$ is the number of variables in $E$, and $n$ is the number of symbols in the terms in $E$.

• The measure is extended to pairs $(E, S)$ and $\perp$ by defining $\mu(E, S) = \mu(E)$ and $\mu(\perp) = (0, 0)$.

• Then check that $(E, S) \rightarrow U$ implies $\mu(E, S) > \mu(U)$.
Preliminary remark

• In \((E, S)\) the second component of \(S\) is just used to accumulate the substitutions.

• Therefore:

\[(E, S) \rightarrow^m (\emptyset, S_n \circ \ldots \circ S_1 \circ S) \iff (E, \emptyset) \rightarrow^m (\emptyset, S_n \circ \ldots \circ S_1)\]

where \(m \geq 1, n \geq 0\) and the \(S_i\) are the elementary substitutions \([t/x]\) introduced by rule (\(vt_1\)).
(2) Proof of ‘soundness’

By induction on the length of the derivation.

• For instance, suppose

\[(E \cup \{x = t\}, id) \rightarrow ([t/x]E, [t/x]) \rightarrow^* (\emptyset, S \circ [t/x])\]

• Then, by the preliminary remark, the inductive hypothesis applies to \([t/x]E, id\).

• Thus \(S \models [t/x]E\). Which entails \(S \circ [t/x] \models E\).

• Moreover, since \(x \notin Var(t)\), \(S \circ [t/x](x) = S(t) = S \circ [t/x](t)\).
(3) Proof of ‘completeness’

- By (1) all reduction sequences terminate. We proceed by induction on the length of the reduction sequence.

- We observe that if $E$ is not empty then at least one rule applies.

- Since $T \models E$ it is easily checked that rules (vt$_2$) and (f$_2$) do not apply.
• For instance, suppose \((E \cup \{x = t\}, id) \rightarrow ([t/x]E, [t/x])\) by \((vt_1)\).

• We recall (see exercise) that if \(T \models E \cup \{x = t\}\) then \(T \models [t/x]E\) and \(T = T \circ [t/x]\).

• Then, from \(T \models [t/x]E\) and by inductive hypothesis we conclude that \(([t/x]E, id) \rightarrow^* (\emptyset, S)\) and \(S \leq T\).

• Hence:

\[S \circ [t/x] \leq T \circ [t/x] = T.\]
Exercise (size of solution)

The size of a term is the number of nodes in its tree representation.

Consider the following unification problem:

\[ \{ x_1 = f(x_0, x_0), x_2 = f(x_1, x_1), \ldots, x_n = f(x_{n-1}, x_{n-1}) \} \]

Compute the most general unifier \( S \).

What is the size of \( S(x_n) \) as a function of \( n \)?
Remark (on complexity)

Efficient algorithms for syntactic unification are based on a dag representation of terms and they can be implemented to run in quasi-linear time.
Summary

• Notion of **term rewriting system**: a schematic presentation of a rewriting system.

• There are **efficient algorithms** to solve equations between first-order terms (a.k.a syntactic unification).

• When a solution exists, one can find a **most general one**.
Exercise (on filters)

Let \( t, s, \ldots \) be terms over a signature \( \Sigma \).

We say that \( t \) is a \textbf{filter} (or pattern) for \( s \) if there is a substitution \( S \) such that \( S t = s \). In this case we write: \( t \leq s \).

Show or give a counter-example to the following assertions:

1. If \( t \leq s \) then \( t \) and \( s \) are unifiable.
2. If \( t \) and \( s \) are unifiable then \( t \leq s \) and \( s \leq t \).
3. If \( t \leq s \) and \( s \leq t \) then \( s \) and \( t \) are unifiable.
4. For all \( t, s \) one can find \( r \) such that \( r \leq t \) and \( r \leq s \).
5. For all \( t, s \) one can find \( r \) such that \( r \geq t \) and \( r \geq s \).
Recommended reading

Baader, Nipkow, *Term rewriting and all that*, Cambridge University Press, chapter 3 and sections 4.5-4.6.
Termination of Term Rewriting Systems
Plan

We introduce two methods to prove termination of TRS:

- An interpretation method (already applied).
- Syntax-directed methods called recursive path orderings.

The second family of methods can be approached either through the notion of well-partial order and Kruskal’s theorem or through the reducibility candidates method.
Reduction orders

A reduction order $>$ is a well-founded order on $T_\Sigma(V)$ that is closed under context and substitution:

\[
\begin{align*}
  t &> s \\
  C[t] &> C[s] \text{ and } St > Ss
\end{align*}
\]

where $C$ is any one hole context and $S$ is any substitution.

The following follows just by unfolding the definitions.

**Proposition**  A TRS $R$ terminates iff there is a reduction order $>$ such that $(l, r) \in R$ implies $l > r$. 
Interpretation method
Well-founded $\Sigma$-algebras and strictly monotonic functions

Suppose the TRS is given over a signature $\Sigma$.

- Consider a $\Sigma$-algebra (cf. exercise on $\Sigma$-algebras) where the domain is a well-founded set $(A, >)$.
- Further assume for every symbol $f \in \Sigma$ the associated function over $A$ is strictly monotone:

\[ a_i > b_i \implies f^A(\ldots, a_i, \ldots) > f^A(\ldots, b_i, \ldots) \]

- Recall that given an assignment $\theta : V \rightarrow A$, for every $t \in T_\Sigma(V)$ there is a unique interpretation in the $\Sigma$-algebra which is defined as follows:

\[
[x] \theta = \theta(x) \\
[f(t_1, \ldots, t_n)] \theta = f^A([t_1] \theta, \ldots, [t_n] \theta)
\]

This is the usual interpretation of terms in first-order logic.
**Proposition**  If all functions $f^A$ are **monotone**, then the following is a **reduction order** on $T_\Sigma(V)$:

$$t >_A s \text{ if } \forall \theta \ [t]\theta >_A [s]\theta$$
Proof(1/4)

Well-foundation Suppose by contradiction

\[ t_0 >_A t_1 >_A \cdots \]

Then taking an arbitrary assignment \( \theta \) we have:

\[ \llbracket t_0 \rrbracket \theta >_A \llbracket t_1 \rrbracket \theta >_A \cdots \]

But this contradicts the hypothesis that \((A, >)\) is well-founded.
Proof (2/4)

Preserved by substitution

• Suppose \( t >_A t' \).

• For any \( s, x \) we show \([s/x]t >_A [s/x]t'\) (easy to generalise to a substitution \([s_1/x_1, \ldots, s_n/x_n]\)).

• That is for any \( \theta \),

\[
[[s/x]t]\theta >_A [[s/x]t']\theta
\]
Proof (3/4)

• We note that:

\[[[s/x]t] \theta = [t] \theta[[s] \theta/x]\]

• Thus taking \(\theta' = \theta[[s] \theta/x]\) we have:

\[[[s/x]t] \theta = [t] \theta' >_A [t'] \theta' = [[[s/x]t'] \theta\]

76
Proof (4/4)

Preserved by context

• We proceed by induction on the context.
• The case for the empty context is immediate.
• For the inductive step, suppose $C = f(\cdots, C', \cdots)$.
• By inductive hypothesis, $C'[t] >_A C'[s]$ if $t >_A s$.
• Then conclude by using the fact that $f^A$ is strictly monotone in every argument.
**Corollary**  Let $R$ be a set of rewriting rules. The TRS terminates if $l >_A r$, for all $(l,r) \in R$.

**Proof**  We have shown that $>_A$ is a reduction order and we have previously observed that a system is terminating if all the rules are compatible with a reduction order.
Example: polynomial interpretations

Take \( A = \{ n \in \mathbb{N} \mid n \geq a \geq 1 \} \). With \( f^n \in \Sigma \) associate a multivariate polynomial \( p_f(x_1, \ldots, x_n) \) such that:

1. Coefficients range over the natural numbers: thus no negative coefficients, and the polynomials are monotonic.

2. \( p_f(a, \ldots, a) \in A \): thus \( p_f \) defines a function over the domain \( A \).

3. Every variable appears in a monomial with non-zero multiplicative coefficient: thus we have a strictly monotonic function.

NB By taking \( a \geq 1 \) we avoid 0 which may be problematic for strict monotonicity.
Exercise

Check that the previous conditions are enough to have a strictly monotonic interpretation.
Example

A definition of **addition** and **multiplication** over **tally numbers** (unary notation):

\[
a(z, y) \rightarrow y \quad a(x, z) \rightarrow x \quad a(s(x), s(y)) \rightarrow s(s(a(x, y))
\]

\[
m(z, x) \rightarrow z \quad m(s(x), y) \rightarrow a(y, m(x, y))
\]

A polynomial interpretation is:

\[
p_z = 1 \quad p_s = x + 2 \quad p_a = 2x + y + 1 \quad p_m = (x + 1)(y + 1)
\]
Exercise (polynomial interpretation)

1. Find a polynomial interpretation for the TRS:

\[ f(f(x,y), z) \rightarrow f(x, f(y, z)) \quad f(x, f(y, z)) \rightarrow f(y, y) \]
2. Find a polynomial interpretation for the TRS

\[ x + 0 \rightarrow x \quad x + s(y) \rightarrow s(x + y) \quad \text{(addition)} \]
\[ d(0) \rightarrow 0 \quad d(s(x)) \rightarrow s(s(d(x))) \quad \text{(double)} \]
\[ q(0) \rightarrow 0 \quad q(s(x)) \rightarrow q(x) + s(d(x)) \quad \text{(square)} \]
Limitations of polynomial interpretations

• Hard to find. Even checking whether an interpretation is valid is in general **undecidable**.

• This follows from the undecidability of the so-called Hilbert’s 10\(^{th}\) problem (stated in 1900, finally proved by Matiyasevich in 1970).

  The set of polynomials \(P(x_1, \ldots, x_k)\) such that:

  \[
  \exists n_1, \ldots, n_k \in \mathbb{Z} \quad P(n_1, \ldots, n_k) = 0
  \]

  is undecidable.

• Cannot handle fast growing functions. Length of reductions can be at most **double exponential**: the previous exercise provides a lower bound, the upper bound is not too hard to obtain.
Simplification orders
Simplification orders

A strict order $>$ on $T_\Sigma(V)$ is a simplification order (ordre de simplification) if it is closed under context and substitution and moreover for all functions $f \in \Sigma$ it satisfies:

$$f(x_1, \ldots, x_n) > x_i \text{ for } i = 1, \ldots, n$$
Exercise (subterm property)

Show that if $>$ is a simplification order and $C$ is a one hole context with $C \neq [\ ]$ then

$$C[t] > t$$
Homeomorphic embedding

Let $\rightarrow$ be the TRS induced by the rules

$$f(x_1, \ldots, x_n) \rightarrow x_i \text{ for } i = 1, \ldots, n$$

**Definition** We write $t \triangleright s$, read $t$ embeds (prolonge) $s$, if $t \twoheadrightarrow s$, i.e., if we can rewrite $t$ in $s$ in a finite number of steps (possibly 0).
Example (homeomorphic embedding)

\[ f(f(h(a), h(x)), f(h(x), a)) \trianglerighteq f(f(a, x), x) \]
Exercise (alternative definition of homeomorphic embedding)

Here is another definition of the homeomorphic embedding:

\[ x \succeq x \]

\[ s_i \succeq t_i, i = 1, \ldots, n \]

\[ f(s_1, \ldots, s_n) \succeq f(t_1, \ldots, t_n) \]

\[ s_i \succeq t \text{ for some } i \]

\[ f(s_1, \ldots, s_n) \succeq t \]

Show that this definition is equivalent to the previous one.
Exercise (homeomorphic embedding vs. simplification order)

Show that if $>$ is a simplification order and $\geq$ is its reflexive closure then $t \triangleright s$ implies $t \geq s$ (in other terms, if $t \triangleright s$ and $t \neq s$ then $t > s$).

NB The reflexive closure of a simplification order may contain strictly the homeomorphic embedding.
Recursive path order (RPO)
RPO: overview

- A reduction order requires well-foundedness which is a global property.
- We want to replace the global property by a local subterm property.
- The recursive path orders (ordres recursifs sur les chemins) are an important class of simplification orders.
- As a corollary of Kruskal’s theorem (to come) one can derive that they also are reduction orders (so they guarantee termination).
• We fix a **strict partial order** $\succ_{\Sigma}$ on the function symbols in $\Sigma$.

• To **check** $t = f(t_1, \ldots, t_m) > g(s_1, \ldots, s_n) = s$, check first whether $s$ is a subterm of $t$. Otherwise, compare the top symbols $f, g$:
  
  - If $f \succ_{\Sigma} g$ then check recursively $t > s_i$ for $i = 1, \ldots, n$.
  
  - If $f = g$ then fix a way to compare $(t_1, \ldots, t_m)$ with $(s_1, \ldots, s_m)$.

  - If none of the above works, give up.
Ordering tuples

There are several ways to lift an order to tuples so as to preserve well-foundation. Two standard ones are:

\[
\begin{align*}
&s_1 \geq t_1, \ldots, s_n \geq t_n, \ \exists j \ s_j > t_j \\
&(s_1, \ldots, s_n) > (t_1, \ldots, t_n) \\
\end{align*}
\]

product

\[
\begin{align*}
&s_1 = t_1, \ldots, s_{i-1} = t_{i-1}, s_i > t_i \\
&(s_1, \ldots, s_n) > (t_1, \ldots, t_n) \\
\end{align*}
\]

lexicographic
Exercise (on product and lexicographic order)

Suppose \((M, >)\) is a well-founded set. Show that the following sets are well-founded:

1. the product \(M^k\) with the product order.
2. the product \(M^k\) with the lexicographic order.
(Finite) multi-sets

We prepare the ground for a third way of comparing vectors.

- A **multi-set** $M$ over a set $A$ is a function $M : A \rightarrow \mathbb{N}$.
- If $M(a) = k$ then $a$ occurs $k$ times in the multi-set.
- A **finite** multi-set is a multi-set $M$ such that $\{a \mid M(a) \neq 0\}$ is finite.
• Let $\mathcal{M}_{\text{fin}}(X)$ denote the finite multi-sets over a set $X$.

• Assume $(X, >)$ is an order and $M, N \in \mathcal{M}_{\text{fin}}(X)$. We write $M >_{1,m} N$ if $N$ is obtained from $M$ by replacing an element by a multi-set of elements which are strictly smaller.

• For instance, if $X = \mathbb{N}$ then

$$\{|1, 3|\} >_{1,m} \{|1, 2, 2, 1|\} >_{1,m} \{|0, 2, 2, 1|\} >_{1,m} \{|0, 1, 1, 2, 1|\}$$

• We define the multi-set order $>_{m}$ as the transitive closure of $>_{1,m}$. 
Exercise (multi-set)

Suppose $(X, >)$ well-founded. Show that the finite multi-sets with the multi-set order form again a well-founded set.

To prove this, assume:

**König lemma**  Every finitely branching tree with an infinite number of nodes admits an infinite path.
Formalisation of König lemma

- A **tree** is a subset of $\mathbb{N}^*$ satisfying:
  1. If $w \in D$ and $w'$ is a prefix of $w$ then $w' \in D$.
  2. If $w_i \in D$ and $j < i$ then $w_j \in D$.

- This representation includes trees with a countable number of nodes and even trees with nodes having a countable number of children.

- We say that a tree is **finitely branching** if every node has a finite number of children.
Proof of König lemma

We build an infinite path in $D$.

- Let $\pi = i_1 \cdots i_k \in D$ be a path such that the subtree with root $\pi$ is infinite.
- Since $D$ is finitely branching there exists a $i_{k+1}$ such that $\pi i_{k+1} \in D$ and the subtree with root $\pi i_{k+1}$ is infinite.
- Proceeding in this way, we can build an infinite path in $D$. 
Comparing tuples by multi-set ordering

• Let us go back to RPO.

• A third way to compare tuples is to consider them as finite multi-sets and use the induced multi-set order.

\[
\{s_1, \ldots, s_n\} > \{t_1, \ldots, t_n\} \\
(s_1, \ldots, s_n) > (t_1, \ldots, t_n)
\]
Recursive path order

The **status** \( \tau(f) \) of a function describes how tuples are to be compared (product, lexicographic, multi-set, ...)

\[
s \geq_r t \\
f(\ldots s \ldots) >_r t
\]

\[
f > \sum g \quad f(s_1, \ldots, s_m) >_r t_i \quad i = 1, \ldots, n
\]

\[
f(s_1, \ldots, s_m) >_r g(t_1, \ldots, t_n)
\]

\[
(s_1, \ldots, s_m) >_{r}^{(f)} (t_1, \ldots, t_m)
\]

\[
f(s_1, \ldots, s_m) >_r t_i \quad i = 1, \ldots, m
\]

\[
f(s_1, \ldots, s_m) >_r f(t_1, \ldots, t_m)
\]

**NB** For the lexicographic order, the condition \( f(s_1, \ldots, s_m) >_r t_i \) is needed.
The recursive path order is a simplification order on $T_\Sigma(V)$.

Given that RPO is a simplification order, there are two main strategies to derive that it is a reduction order too which will be described later:

- To appeal to Kruskal’s theorem.
- To apply a so called reducibility candidates method (inspired by the $\lambda$-calculus).
Proof of RPO is a simplification order

To fix ideas, we consider a particular case where we always compare tuples via the **product order**. We prove the following properties:

- $> \text{ is strict.}$
- $s > t \text{ implies } \text{Var}(s) \supseteq \text{Var}(t).$
- Transitivity.
- Subterm property.
- Closure under substitution.
- Closure under context.
Specialised definition for product

It is convenient to have the specialised definition for product:

\[(R_1)\]
\[
\frac{s \geq_r t}{f(\ldots s \ldots) >_r t}
\]

\[(R_2)\]
\[
\frac{f > \sum g \quad f(s_1, \ldots, s_m) >_r t_i \quad i = 1, \ldots, n}{f(s_1, \ldots, s_m) >_r g(t_1, \ldots, t_n)}
\]

\[(R_3)\]
\[
\frac{s_i \geq t_i \text{ for } i \in 1..m, \text{ and } \exists j \in 1..m \quad s_j > t_j}{f(s_1, \ldots, s_m) >_r f(t_1, \ldots, t_m)}
\]
> is strict

- By induction on $s$ show that $s > s$ is impossible.
- Note in particular that $x > t$ and $f > f$ are impossible.
$s > t$ implies $\text{Var}(s) \supseteq \text{Var}(t)$

By induction on the proof of $s > t$. 
Transitivity

Suppose \( s_1 > s_2 \) and \( s_2 > s_3 \). Show \( s_1 > s_3 \) by induction on \( |s_1| + |s_2| + |s_3| \) analysing the last rules applied in the proof of \( s_1 > s_2 \) and \( s_2 > s_3 \) (9 cases).
Subterm property

We check that $f(x_1, \ldots, x_n) > x_i$ for $i = 1, \ldots, n$. 
Closure under substitution

Show that $t > r$ implies $[s/x]t > [s/x]r$ by induction on $|t| + |r|$. 

Closure under context

Show by induction on the structure of a one hole context that $t > s$ implies $C[t] > C[s]$. 
Exercise

Prove termination by RPO by a suitable choice of the status of + and ∗.

\[(x + y) + z \rightarrow x + (y + z)\]
\[x \ast s(y) \rightarrow x + (y \ast x)\]
Exercise (Ackermann)

\[
\begin{align*}
ack(z, n) & \rightarrow s(z) \\
ack(s(z), z) & \rightarrow s^2(z) \\
ack(s^2(m), z) & \rightarrow s^2(m) \\
ack(s(m), s(n)) & \rightarrow ack(ack(m, s(n)), n)
\end{align*}
\]

Prove termination by RPO.
Remarks

- Ackermann function is an example of total recursive function that cannot be defined by primitive recursion (definition to come).

- Ackermann grows more quickly than any primitive recursive function.

  \[
  \text{#ack}(2,2);;
  \]
  - : int = 4

  \[
  \text{#ack}(3,3);;
  \]
  - : int = 16

  \[
  \text{#ack}(4,4);;
  \]
  Uncaught exception: Out_of_memory

- In particular, Ackermann cannot be shown to be terminating by a polynomial interpretation.
**Polynomial vs. RPO**

Sometimes the interpretation (polynomial) method beats RPO.

1. Show that the TRS

\[ b(x) \rightarrow r(s(x)) \quad r(s(s(x))) \rightarrow b(x) \]

terminates by polynomial interpretation.

2. Show that there is no rpo on Σ that will prove termination.

3. RPO is a particular type of simplification order. Is there a simplification order that shows termination of the TRS above?
Exercise (simplification vs. reduction)

Consider the TRS

\[ f(f(x)) \rightarrow f(g(f(x))) \]

1. Show that the TRS is terminating.

2. Show that there is no simplification order \( > \) that contains \( \rightarrow \).
Computational aspects

1. Given an order on the signature, the induced rpo can be decided in **time polynomial** in the size of the terms.

2. In general, existence of a rpo for a TRS is an **NP-complete** problem (guess an order non-deterministically, and check in polynomial time).

3. Functions computed by TRS admitting a lpo termination proof correspond to **multiple recursive functions**. Mpo and ppo are a bit weaker as they correspond to **primitive recursive functions** (to be discussed later).
RPO is a reduction order via the reducibility candidates method
Goal

• We know RPO is a simplification order (= a strict order closed under context and substitution).

• We want to show that it is well-founded (and therefore a reduction order).

• We apply the reducibility candidates method: a proof technique developed first for typed \(\lambda\)-calculi.
Simplified version of RPO

Assume functions’ arguments are always compared with the lexicographic order from left to right.

\[
\frac{s \geq_r t}{f(\ldots s \ldots) >_r t}
\]

\[
\frac{f > \Sigma g \quad f(s_1, \ldots, s_m) >_r t_i \quad i = 1, \ldots, n}{f(s_1, \ldots, s_m) >_r g(t_1, \ldots, t_n)}
\]

\[
\frac{(s_1, \ldots, s_m) >^{\text{lex}}_r (t_1, \ldots, t_m)}{f(s_1, \ldots, s_m) >_r t_i \quad i = 1, \ldots, m}
\]

\[
\frac{f(s_1, \ldots, s_m) >_r f(t_1, \ldots, t_m)}{}
\]
The well-founded set of terms $WF$

We work on the set of terms $T_\Sigma(V)$ and define:

$$WF = \{ t \in T_\Sigma(V) \mid \text{there is no infinite sequence } t = t_0 >_r t_1 >_r \cdots \}$$

$$Red(t) = \{ s \mid t >_r s \}$$

**Exercise**  Show that:

1. $(WF, >_r)$ is a well-founded set.

2. If $Red(t) \subseteq WF$ then $t \in WF$.

3. If $s \in WF$ and $s >_r t$ then $t \in WF$.

**NB** For historical reasons, in $\lambda$-calculus, the set $WF$ (well-founded terms) will be called $SN$ (strongly normalisable terms).
The key property

• Let $>^\text{lex}_r$ be the lexicographic ordered induced by $>_r$ on vectors of $n$ terms in $WF$.

• The key property is:

  If $s_1, \ldots, s_n \in WF$ and $f(s_1, \ldots, s_n) >_r t$ then $t \in WF$.

**Exercise**  Assuming the key property, derive by induction on the structure of the terms that all terms are in $WF$. 
The induction principle at work

We prove the key property by induction on the triple:

$$(f, (s_1, \ldots, s_n), |t|)$$

with the lexicographic order from left to right where:

- The first component is a function symbol ordered by $>_{\Sigma}$.
- The third is the size of the term with the usual order on natural numbers.
- For the second, consider the set $\bigcup_{f \in \Sigma} WF^{ar}(f)$ ordered by:

$$(s_1, \ldots, s_n) > (t_1, \ldots, t_m) \text{ iff } n = m \text{ and } (s_1, \ldots, s_n) >^r_{lex} (t_1, \ldots, t_m)$$

Two vectors of different lengths are incomparable.

Also $(WF, >_r)$ well-founded implies $(WF^n, >^r_{lex})$ is well-founded too.
Case $f(s_1, \ldots, s_n) >_r t$ as $s_i = t$ or $s_i >_r t$.

- If $s_i = t$ the conclusion is immediate as $s_i \in WF$ by hypothesis.
- If $s_i >_r t$ then $t \in WF$ as $s_i \in WF$. 
Proof by induction of key property (2/3)

Case \( t = g(t_1, \ldots, t_m), f \succ \Sigma g, f(s_1, \ldots, s_n) >_r t_i \) for \( i = 1, \ldots, m \).

- We notice that \((f, (s_1, \ldots, s_n), |t|) > (f, (s_1, \ldots, s_n), |t_i|)\) for \( i = 1, \ldots, m \). Hence by inductive hypothesis, \( t_i \in WF \).

- Suppose \( g(t_1, \ldots, t_m) >_r u \). We remark \((f, (s_1 \ldots, s_n), |t|) > (g, (t_1, \ldots, t_m), |u|)\). Hence by ind. hyp. \( u \in WF \) and by a preliminary remark \( g(t_1, \ldots, t_m) \in WF \).
Proof by induction of key property (3/3)

Case \[ t = f(t_1, \ldots, t_n), \quad f(s_1, \ldots, s_n) >_r t_i \text{ for } i = 1, \ldots, n, \]
\[ (s_1, \ldots, s_n) >_{lex} ^r (t_1, \ldots, t_n). \]

This case is similar to the previous one.

- We remark that \((f, (s_1, \ldots, s_n), |t|) > (f, (s_1, \ldots, s_n), |t_i|)\) for \(i = 1, \ldots, n\). Hence by ind. hyp. \(t_i \in WF\).

- Suppose \(f(t_1, \ldots, t_n) >_r u\). We notice
  \((f, (s_1 \ldots, s_n), |t|) > (f, (t_1, \ldots, t_n), |u|)\) (second component decreases!). By ind. hyp. \(u \in WF\) and by a preliminary remark \(f(t_1, \ldots, t_n) \in WF\).
Well-partial orders, Kruskal theorem, and applications
**Exercise (Dickson’s lemma as a worm up)**

Consider the **product order** $\geq$ on $\mathbb{N}^k$ (vectors of natural numbers):

$$(n_1, \ldots, n_k) \geq (m_1, \ldots, m_k) \text{ if } n_i \geq m_i, i = 1, \ldots, k$$

1. Show that $>$ (the strict part of $\geq$) is well-founded.

2. Show by induction on $k$, that from every sequence $\{v_n\}_{n \in \mathbb{N}}$ in $\mathbb{N}^k$ we can extract a **growing subsequence**, namely there is an injective function $\sigma : \mathbb{N} \to \mathbb{N}$ such that:

$$\forall n \ \ v_\sigma(n) \leq v_\sigma(n+1)$$

3. Show that every set of incomparable elements in $\mathbb{N}^k$ (an **anti-chain**) is finite.
Well partial order

A well partial order (bel ordre) \((A, <)\) is a strict \((\forall a \ a \not< a)\) partial order such that for any sequence \(\{a_i \mid i \in \mathbb{N}\}\) in \(A\),

\[
\exists i, j \in \mathbb{N} \ i < j \text{ and } a_i \leq a_j
\]

Such a sequence is called **good**.

Otherwise, we call the sequence **bad**. This means:

\[
\forall i, j \in \mathbb{N} \ (i < j \text{ implies } a_i \not\leq a_j)
\]

Note that if a sequence is bad all its subsequences are.

**Terminology**  Here \(\leq\) is the **reflexive closure** of \(<\). Remember that in this course the partial order relation is just required to be **transitive**. The reflexive closure of a well partial order is called a **well quasi-ordering**.
Well-partial order vs. well-founded order

Well partial orders are the \textbf{well-founded orders} that have \textbf{no infinite anti-chain}.

$(\Rightarrow)$

- A wpo must be well-founded for a strictly descending chain gives a bad sequence.
- For the same reason, a wpo cannot contain an infinite anti-chain.
• Vice versa, take a well-founded set without infinite anti-chain.

• Given an infinite sequence, the set of minimal elements of the sequence must be finite.

• Therefore there is a minimal element such that the sequence is infinitely often above it.
Extraction of an ascending sequence in a wpo

In a wpo, every sequence \( \{a_i\}_{i \in \mathbb{N}} \) has an **ascending subsequence**:

\[
i_1 < i_2 < i_3 < \ldots \text{ and } a_{i_1} \leq a_{i_2} \leq a_{i_3} \leq \ldots
\]

- In a good sequence there are **finitely many** \( a_i \) such that
  \( \forall j \; i < j \Rightarrow a_i \not\leq a_j \) (otherwise the sequence composed of all such elements is **bad**).

- Thus starting from a certain point \( i_1 \), if \( i \geq i_1 \) then
  \( \exists j > i \; a_i \leq a_j \).

- Now starting from \( i_1 \) we can inductively build a sequence
  \( i_1 < i_2 < \ldots \) such that \( a_{i_1} \leq a_{i_2} \leq \ldots \)
Product of wpo’s

The product $A \times B$ of wpo’s $A, B$, ordered componentwise (product order) is a wpo.

- Consider $\{(a_i, b_i) \mid i \in \mathbb{N}\}$ and suppose $\{a_i \mid i \in \mathbb{N}\}$ and $\{b_i \mid i \in \mathbb{N}\}$ are both infinite (otherwise it is easy).

- Consider the subsequence $i_0 < i_1 < i_2 < \cdots$ such that $a_{i_0} \leq a_{i_1} \leq a_{i_2} \leq \cdots$ (cf. previous proposition).

- Find $l < k$ such that $b_{i_k} \leq b_{i_l}$. 

134
**Kruskal theorem**

Recall that $\succeq$ is the **homeomorphic embedding** and that the **strict part** of $\succeq$, say $\succ$, is contained in every simplification order.

**Theorem (1960)**  Suppose $\Sigma$ and $V$ finite. Then the **strict** homeomorphic embedding $\succ$ on $T_\Sigma(V)$ is a **well partial order**.

**NB** If $\Sigma$ or $V$ are **infinite** then $(T_\Sigma(V), \succ)$ contains an **infinite anti-chain** and the theorem does **not** hold.
Proof of Kruskal theorem (after Nash-Williams)

By contradiction. Suppose there is a bad sequence in $T_\Sigma(X)$.

- Extract from the bad sequence a minimal one with respect to the size of the terms, say $t_1, t_2, \ldots$. This means that having built the sequence $t_1, \ldots, t_i$, we pick a term $t_{i+1}$ of minimal size among those that follow $t_i$.

- Define:

$$S_i = \begin{cases} 
\emptyset & \text{if } t_i \text{ variable} \\
\{s_1, \ldots, s_n\} & \text{if } t_i = f(s_1, \ldots, s_n) 
\end{cases}$$

$$S = \bigcup_{i \geq 0} S_i$$

- Show that (tricky point!): $(S, \triangleright)$ is a wpo.
Proof Kruskal theorem continued

- Since $\Sigma$ and $X$ are finite, there must be a symbol that occurs infinitely often at the root. If it is a variable or a constant we derive a **Contradiction**.

- Otherwise, we have $i_0 < i_1 < \ldots$ with

  \[ t_{i_k} = f(s_{i_1}^{i_k}, \ldots, s_{i_n}^{i_k}) \]

- Now $(S, \triangleright)$ is a wpo and the product of wpo’s is a wpo. Therefore $\{(s_{i_k}^{i_1}, \ldots, s_{i_n}^{i_k})\}_{k \geq 0}$ is a **good** sequence.

- So $\exists p, q \ p < q$ and $s_{i_q}^{i_l} \triangleright s_{i_p}^{i_l}, l = 1, \ldots, n$.

- And this entails $t_{i_q} \triangleright t_{i_p}$. **Contradiction**.
More on the tricky point (1/2)

$(S,\triangleright)$ is a wpo.

- Suppose not, and let $s_1, s_2, s_3 \ldots$ be a bad sequence. The $s_i$ must be all distinct.

- Suppose $s_1 \in S_k$. This entails $t_k \triangleright s_1$. Let $S_{<k} = S_1 \cup \cdots \cup S_{k-1}$. There is an index $l$ such that $s_i \notin S_{<k}$ for $i \geq l$.

- Consider

$$t_1, \ldots, t_{k-1}, s_1, s_l, s_{l+1}, \ldots$$

Since $s_1$ is smaller than $t_k$, by minimality (!) this sequence must be good.
More on the tricky point (2/2)

• Since $t_1, t_2, \ldots$ and $s_1, s_2, \ldots$ are bad, this means

$$s_j \trianglerighteq t_i \quad i \in \{1, \ldots, k - 1\}, j \in \{1, l, l + 1, \ldots\}$$

$j = 1 \quad t_k \triangleright s_j \trianglerighteq t_i$. **Contradiction.**

$j \geq l$ Suppose $s_j \in S_m \setminus S_{<k}$. Thus $i < k \leq m$ and $t_m \triangleright s_j \trianglerighteq t_i$. **Contradiction.**
Logical remarks

• The proof we have presented is non-constructive (two nested arguments by contradiction). Indeed it is a research problem to find constructive versions of the proof.

• The theorem is a simple example of a combinatorial statement that cannot be proved in Peano Arithmetic.
Exercise (Higman’s lemma)

The following is a special case of Kruskal’s theorem on words known as Higman’s lemma.

- Let \( \Sigma \) be a finite set (alphabet).
- Given two words \( w, w' \in \Sigma^* \) we say that \( w' \) is a subsequence of \( w \), and write \( w \geq w' \), if the word \( w' \) can be obtained from the word \( w \) by erasing some (possibly none) of its characters.
- Apply Kruskal’s theorem to conclude that \( > \) is a well partial order.
A generalisation to graphs

There is a famous generalisation of Kruskal’s theorem to graphs.

- An edge contraction of a graph consists in removing an edge while merging the two vertices.

- A graph $G$ is a **minor** of the graph $H$ if it can be obtained from $H$ by a sequence of edge contractions.

- Robertson and Seymour theorem (2004) states that the minor relation is a well partial order.
Application 1: simplification orders are well-founded

Every simplification order on $T_{\Sigma}(V)$ is well founded

Hence every simplification order (in particular RPO) is a reduction order.
Proof (1/2)

By contradiction, suppose $t_1 > t_2 > \ldots$.

- First, we prove by contradiction that $\text{Var}(t_1) \supseteq \text{Var}(t_2) \supseteq \ldots$:
  - Suppose $x \in \text{Var}(t_{i+1}) \setminus \text{Var}(t_i)$ and consider the substitution $S = [t_i/x]$.
  - Then $St_i = t_i > St_{i+1}$.
  - By the subterm property we have $t_{i+1} \geq t_i$.
  - Thus $t_i > t_i$ which contradicts the hypothesis that $>$ is strict.
Proof (2/2)

• Thus we can take $X = \text{Var}(t_1)$ which is \textbf{finite}.

• Then we can apply \textbf{Kruskal theorem} to the sequence and conclude:

$$\exists i, j \ i < j \text{ and } t_j \triangleright t_i$$

• Thus we have both $t_i > t_j$ and $t_j \geq t_i$.

• Hence $t_i > t_i$.

\textbf{NB} Note that we use $(T_\Sigma(X), \triangleright)$ \textbf{wpo}, not just well-founded...
Application 2: Interpretation over the reals

Take as domain \( A = \{ r \in \mathbb{R} \mid r \geq a \geq 1 \} \) (a non-well founded set!)

With every \( f^n \in \Sigma \) associate a multivariate polynomial \( p_f(x_1, \ldots, x_n) \) such that:

1. Coefficients range over the non-negative reals.
2. \( p_f(a, \ldots, a) \in A \): thus \( p_f \) defines a function over the domain \( A \).
3. \( p_f(a_1, \ldots, a_n) > a_i \) for \( i = 1, \ldots, n \) (the new condition!).

Write \( s > t \) if the polynomial associated with \( s \) is strictly larger than the one associated with \( t \) over the domain \( A \).

This is a simplification order, hence a reduction order.
Remark

• Thus we can use the non-negative reals in the interpretation method.

• A famous result by Tarsky states that the first-order theory of reals is decidable.

• A corollary of this result is that we can decide whether there is a polynomial interpretation over the reals where the polynomials have a bounded degree (recall that this problem is undecidable over the natural numbers: Hilbert’s 10th problem).

• Beware that this is a rather theoretical advantage because of the very high complexity of Tarsky’s decision method!
Summary

1. \((T_\Sigma(V), \triangleright)\) is a well partial order (assuming \(\Sigma\) and \(V\) finite).
2. Every simplification order is well-founded.
3. The recursive path order is a simplification order.
4. The interpretation method over the reals may produce simplication orders.
Recommended reading

F. Baader, T. Nipkow, Term rewriting and all that, Cambridge University Press. Chapter 5 (most of it).

Note  Various refinements/variations of the methods we have considered have been proposed and implemented. A list of research tools is here:

http://www.jaist.ac.jp/~hirokawa/tool/?tag=termination_tool

A pleasant one with a web interface is here:

http://colo6-c703.uibk.ac.at/ttt2/interface/index.php
Confluence and Completion
• In general confluence of a TRS is an **undecidable** property.

• However, if the TRS is **terminating and finite** then the property is **decidable**.

• By Newman theorem, we know that checking **local confluence** is enough.

• To do that it is enough to consider a **finite** number of cases known as **critical pairs**.
Critical pairs

- Consider the term rewriting rules \( l_i \rightarrow r_i \) for \( i = 1, 2 \) (it is possible to pick up twice the same rule).

- The variables in each rule can be \textit{renamed} without changing the induced reduction system. Thus we assume the variables in the two rules are disjoint.

- Suppose \( l_1 = C[l'_1] \) where \( l'_1 \) is \textbf{not} a variable.

- Let \( S \) be the \textbf{most general unifier} of \( l'_1 \) and \( l_2 \) (if it exists).

- Then \( (S(r_1), S(C[r_2])) \) is a \textbf{critical pair}. 
Preliminary remark

- The **domain** of a substitution $S$ is the set $\text{dom}(S) = \{x \in V \mid S(x) \neq x\}$.

- Given two substitutions $S_1, S_2$ let us define their **union** as follows:

$$ (S_1 \cup S_2)(x) = \begin{cases} 
S_1(x) & \text{if } x \in \text{dom}(S_1) \setminus \text{dom}(S_2) \\
S_2(x) & \text{if } x \in \text{dom}(S_2) \setminus \text{dom}(S_1) \\
x & \text{otherwise}
\end{cases} $$

- Suppose $\text{Var}(t) \cap \text{Var}(s) = \emptyset$, $\text{dom}(S_1) \subseteq \text{Var}(t)$, and $\text{dom}(S_2) \subseteq \text{Var}(s)$. Then $S_1(t) = S_2(s)$ entails that $S_1 \cup S_2 \models t = s$. 
**Confluence test**

**Theorem**  Suppose given a *finite* and *terminating* TRS. Then the TRS is *confluent* iff all the *critical pairs* induced by its rules are *joinable* (and the latter is a *decidable* condition).
Justification of confluence test

• The test is obviously necessary.

• Because the TRS is terminating, it is enough to show that the test guarantees local confluence.

• To check local confluence a finite case analysis suffices.

• If $s \rightarrow t_1$ and $s \rightarrow t_2$, then we can find rules $l_1 \rightarrow r_1, l_2 \rightarrow r_2$, contexts $C_1, C_2$ and substitutions $S_1, S_2$ such that

$$s = C_1[S_1l_1] = C_2[S_2l_2], \quad t_1 = C_1[S_1r_1], \quad t_2 = C_2[S_2r_2]$$
Case 1

The paths corresponding to the contexts $C_1$ and $C_2$ are incomparable. In this case one can close the diagram in one step.

Example  Assume the rules

$$g_i(x) \rightarrow k_i(x), \quad i = 1, 2.$$ 

Consider $h(g_1(x), g_2(x))$. 

Case 2

There is a variable $x$ in $l_1$ such that $S_2 l_2$ is actually a subterm of $S_1(x)$. In this case one can always close the diagram, though it may take several steps.

**Example**  Assume the rules

\[ f(x, x, x) \rightarrow h(x, x), \quad g(x) \rightarrow k(x). \]

Consider $f(g(x), g(x), g(x))$.  

Case 3

We can decompose \( l_1 \) in \( C[l'_1] \) so that:

\[ l'_1 \text{ is not a variable and } S_1 l'_1 = S_2 l_2 \]

One can show that this situation is always an instance of a critical pair.

Example  Assume the rules

\[
f(f(x, y), z) \rightarrow f(x, f(y, z)), \quad f(i(x_1), x_1) \rightarrow e .
\]

Consider \( f(f(i(x_1), x_1), z) \).
Consider the TRS:

\[ f(x, g(y, z)) \rightarrow g(f(x, y), f(x, z)) \quad g(g(x, y), z) \rightarrow g(x, g(y, z)) \]

Is the resulting reduction system terminating and/or confluent?
Knuth-Bendix completion

• The test for local confluence is the basis for an iterative symbolic computation method known as Knuth-Bendix completion.

• Given an equational theory, the goal is to obtain a confluent and terminating term rewriting system for it.
Main steps in KB completion

1. Orient the equations thus obtaining a TRS.

2. Check termination of the TRS.

3. Then check local confluence.

4. If a critical pair cannot be joined, then we add the corresponding equation and we repeat the process.

NB There is no guarantee that the process terminates! At various places, one may require a human intervention: orientation, order to check, selection of the rules to add, . . .
Example (success)

• The following law describes so called ‘central grupoids’:

$$(x * y) * (y * z) = y$$

• Any simplification ordering $>$ will satisfy:

$$(x * y) * (y * z) > y$$

so we orient the equation from left to right.

• A critical pair is:

$$((x' * y') * (y' * z')) * ((y' * z') * z) \rightarrow (y' * z'), \quad y' * ((y' * z') * z)$$

Any simplification ordering satisfies: $y' * ((y' * z') * z) > (y' * z')$. 
• Another critical pair is:
\[(x \ast (x' \ast y')) \ast ((x' \ast y') \ast (y' \ast z')) \rightarrow (x' \ast y'), \quad (x \ast (x' \ast y')) \ast y'\]

Again any simplification ordering satisfies:
\[(x \ast (x' \ast y')) \ast y' > (x' \ast y').\]

• Thus we get a terminating TRS with three rules. In the next iteration all critical pairs turn out to be joinable and thus the completion terminates successfully.
Example (failure)

- The equations for left/right distributivity of \( * \) over \( + \) are:

\[
x * (y + z) = (x * y) + (x * z),
(u + v) * w = (u * w) + (v * w)
\]

- Ordering from left to right, a critical pair is:

\[
(u+v)*(y+z) \rightarrow ((u+v)*y) + ((u+v)*z),
(u*(y+z)) + (v*(y+z))
\]

- If we normalise the two terms on the left-hand-side we get:

\[
((u*y)+(v*y)) + ((u*z)+(v*z)),
((u*y)+(u*z)) + ((v*y)+(v*z))
\]

and there is no reasonable way to order them.
Example (divergence)

• Consider the equations:

\[ x + z = x, \quad s(x + y) = x + s(y), \quad x + s(z) = s(x). \]

• Check that by orienting them from left to right we obtain a terminating TRS.

• There is a critical pair between the second and third rule taking

\[ s(x + s(z)) \rightarrow x + s(s(z)), s(s(x)) \]

• In turn this forces the rule:

\[ x + s(s(z)) \rightarrow s(s(x)) \]
• The (simple) completion **diverges** as one has to add all the rules of the shape:

\[ x + s^n(z) \rightarrow s^n(x) \]

• However, the completion strategy succeeds by orienting the second rule in the opposite direction:

\[ x + s(y) \rightarrow s(x + y) \]
Summary

1. Critical pair test to prove (local) confluence of TRS.

2. Idea of Knuth-Bendix completion to build confluent and terminating TRS.
Recommended reading


Note

- Many sophisticated refinements of the completion procedure have been proposed.

- Similar ideas have been developed in parallel and independently in the area of computer algebra (calcul formel) where a technique known as Gröbner bases is used to solve decision problems in rings of polynomials.

- A research tool for completion with a web interface is here: http://colo6-c703.uibk.ac.at/mkbtt/interface/index.php
Exercise (review)

Consider the TRS

\[ f(f(x, y), z) \rightarrow f(x, f(y, z)), \quad f(i(x), x) \rightarrow e \]

1. Can you show termination by RPO?
2. Can you show termination by polynomial interpretation?
3. Is the system confluent?
Now consider the TRS

\[ f(fx) \rightarrow gx \]

1. Is it confluent?

2. Add the rule \( f(g(x)) \rightarrow g(f(x)) \). Is this terminating by RPO?

3. And by polynomial interpretation?

4. Is the system confluent?

5. Same questions if we add the rule \( g(f(x)) \rightarrow f(g(x)) \).
Functional programming,
primitive recursion,
and polynomial time
Plan

• We introduce a simple first-order functional language (a fragment of ML).

• First we present a definition mechanism that guarantees termination: primitive recursion on unary and binary notation (cf. Logic course).

• Then we see under which conditions, we can guarantee termination in polynomial time.

NB This is a gentle introduction to a field known as implicit computational complexity where one tries to characterise complexity classes (such as PTIME) by logical/programming means.
A first-order functional language

- To define **types**, a system of mutually recursive equations:
  \[ t = \cdots | c \text{ of } t_1, \ldots, t_n | \cdots \]

- The symbol \( c \) is a **constructor** of the data type \( t \).

- Elements of type \( t \) can be defined **inductively**: constants, constructors applied to constants, \( \ldots \)
Examples of types

- Booleans.

$$bool = t \mid f$$

- Tally numbers.

$$tnat = z \mid s \text{ of } tnat$$

- Lists of tally numbers.

$$tnatlist = \text{nil} \mid c \text{ of } tnat, tnatlist$$

- Binary numbers (words).

$$bnat = 0 \text{ of } bnat \mid 1 \text{ of } bnat \mid \epsilon$$
On syntax specification

• A grammar is a way of specifying a formal language (a set of words) by a rewriting process.

• A context-free (a.k.a algebraic) grammar is a particular class of grammars which includes a set of non-terminal symbols $N$ a set of terminal symbols $T$ and rules of the shape $A \rightarrow \alpha$ for $A \in N$ and $\alpha \in T^*$. 

• A subset of such grammars is used to specify the syntax of programming languages.
Example (syntax specification)

\[
\text{id} ::= x \mid y \mid z \mid \cdots \\
\text{e} ::= \text{id} \mid (\text{e} + \text{e}) \mid (\text{e} \ast \text{e})
\]

- \( N = \{ \text{e}, \text{id} \} \) (non-terminal), \( T = \{(, +, ), \ast, x, y, z, \ldots \} \) (terminal).

- The set of words \([\text{id}]\) associated with the non-terminal symbol \( \text{id} \) is simply defined by enumeration: \( \{x, y, z, \ldots \} \).

- The set of words \([\text{e}]\) associated with \( \text{e} \), is the least set of words such that \( [\text{e}] \supseteq [\text{id}] \) and:

\[
\begin{align*}
  w_1, w_2 & \in [\text{e}] \quad \text{implies} \quad (w_1 + w_2), (w_1 \ast w_2) \in [\text{e}].
\end{align*}
\]
Syntax (first-order language)

Reserve symbols as follows:

\[
\begin{align*}
\text{cid} & ::= c, c', \ldots \quad \text{constructor symbols} \\
\text{fid} & ::= f, f', \ldots \quad \text{function symbols} \\
\text{id} & ::= x, x'\ldots \quad \text{first-order variables}
\end{align*}
\]

and distinguish the following \textbf{syntactic categories}:

\[
\begin{align*}
\text{v} & ::= \text{cid}(v, \ldots, v) \quad \text{(values)} \\
\text{p} & ::= \text{id} \mid \text{cid}(p, \ldots, p) \quad \text{(patterns)} \\
\text{e} & ::= \text{id} \mid \text{cid}(e, \ldots, e) \mid \text{fid}(e, \ldots, e) \quad \text{(expressions)}.
\end{align*}
\]
Functions

A list of mutually recursive definitions by pattern matching.

\[
f = \\
\ldots \\
p_1, \ldots, p_n \Rightarrow e \\
\ldots
\]

• Patterns are (usually) supposed to be:

  Linear: a variable occurs at most once.

  Orthogonal: a value cannot match two patterns.

  Complete: each value matches a pattern.

• By convention, \_ is the default pattern (a fresh variable).

• We refer to \( f(p_1, \ldots, p_n) \Rightarrow e \) as a rule.
Examples of functions

\( \text{ite} = \)
\( t, y, - \quad \Rightarrow \quad y \)
\( f, -, z \quad \Rightarrow \quad z \)

\( \text{lesseq} = \)
\( z, - \quad \Rightarrow \quad t \)
\( s(x), z \quad \Rightarrow \quad f \)
\( s(x), s(y) \quad \Rightarrow \quad \text{lesseq}(x, y) \)
\begin{align*}
\text{insert} &= \\
x, \text{nil} &\Rightarrow \text{cons}(x, \text{nil}) \\
x, \text{cons}(y, l) &\Rightarrow \text{ite}(\text{lesseq}(x, y), \text{cons}(x, \text{cons}(y, l)), \\
&\quad \text{cons}(y, \text{insert}(x, l)))
\end{align*}
The same in ML

type bool = T | F;;
type tnat = Z | S of tnat;;
type tnatlist = Nil | C of tnat * tnatlist;;
type bnat = O of bnat | I of bnat | E;;

let ite p = match p with
  (T,y,z) -> y
| (F,y,z) -> z ;;

let rec lesseq p = match p with
  (Z,y) -> T
| (S(x),Z) -> F
| (S(x),S(y)) -> lesseq(x,y);;

let rec insert p = match p with
  (x,Nil) -> C(x,Nil)
| (x,C(y,l)) -> ite(lesseq(x,y),C(x,C(y,l)),C(y,insert(x,l)));
Call-by-value evaluation

Expressions are reduced to values before being passed as parameters. This leads to the following definition of evaluation.

\[ e_j \downarrow v_j \quad j = 1, \ldots, n \]
\[ c(e_1, \ldots, e_n) \downarrow c(v_1, \ldots, v_n) \]

\[ e_j \downarrow v_j, \quad f(p_1, \ldots, p_n) \Rightarrow e, \quad Sp_j = v_j, \quad j = 1, \ldots, n \quad S(e) \downarrow v \]
\[ f(e_1, \ldots, e_n) \downarrow v \]
Remark (first-order functional programs are TRS)

It is always possible to simulate the computations of a functional program with the computation of a corresponding TRS. This TRS has a special shape:

- The signature can be partitioned in constructors and functions.
- The left hand side of a rule has exactly one function symbol and is linear (a variable occurs at most once):
  \[ f(p_1, \ldots, p_n) \rightarrow e \]

Then it is easy to check that \( e \downarrow v \) implies \( e \rightarrow^* v \).
Exercise (first-order functional programming)

Define the functions for sorting a list of integers according to the following two methods:

**Insertion sort** To sort the list \( \text{cons}(y, l) \), first sort \( l \) then insert \( y \) in \( l \).

**Quick sort** To sort the list \( \text{cons}(y, l) \) split the list \( l \) in two according to the pivot \( y \), sort them, and then combine them.

Evaluate the functions insertion sort and quick sort on the argument:

\[
\text{cons}(s(s(z)), \text{cons}(z, \text{cons}(s(z), \text{nil})))
\]

Note that \( \text{ite} \) function always evaluates **both branches**.
Typing

Programs are supposed to be well typed.

• A constructor $c$ has type $t_1, \ldots, t_n \to t$ if it is declared (once) as $t = \ldots | c$ of $t_1, \ldots, t_n | \ldots$.

• To every function $f$ we must be able to assign a type $t_1, \ldots, t_n \to t$ so that all expressions in the program are well typed with respect to a suitable context $\Gamma \equiv x_1 : t_1, \ldots, x_n : t_n$, where:

\[
\begin{array}{c}
x : t \in \Gamma \\
\Gamma \vdash x : t
\end{array} \quad \quad \begin{array}{c}
\Gamma \vdash e_i : t_i \quad i = 1, \ldots, n \\
\Gamma \vdash c : t_1, \ldots, t_n \to t
\end{array} \quad \quad \begin{array}{c}
\Gamma \vdash e_i : t_i \\
\Gamma \vdash f(e_1, \ldots, e_n) : t
\end{array} \quad \quad \begin{array}{c}
\Gamma \vdash e_i : t_i \\
f : t_1, \ldots, t_n \to t
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash c(e_1, \ldots, e_n) : t
\end{array} \quad \quad \begin{array}{c}
\Gamma \vdash e_i : t_i \\
\Gamma \vdash f(e_1, \ldots, e_n) : t
\end{array}
\]
A rule \( f(p_1, \ldots, p_n) \Rightarrow e \) in a function definition \( f \) is well typed if:

1. \( f : t_1, \ldots, t_n \rightarrow t. \)
2. There are contexts \( \Gamma_i, i = 1, \ldots, n \) such that \( \Gamma_i \vdash p_i : t_i. \)
3. \( \Gamma = \Gamma_1, \ldots, \Gamma_n \) is well formed and \( \Gamma \vdash e : t. \)

**NB** As in ML, there is a notion of most general type of a function that can be automatically inferred.
Exercise (typing)

Type the functions for insertion sort and quick sort.
What we have done so far

• Introduced a first-order functional language.
• Defined a call-by-value, big-step, operational semantics.
• Defined simple typing rules.

The language is easily shown to be **Turing complete** (= can simulate any program).
Primitive recursion
Primitive recursion

We start looking at a fragment with guaranteed termination.

Tally numbers

\[ t\text{nat} = z \mid s \text{ of } t\text{nat} \]

Basic functions

\begin{align*}
z(x) &= z & \text{zero} \\
s(x) &= s(x) & \text{successor} \\
p_i(x_1, \ldots, x_n) &= x_i, i = 1, \ldots, n & \text{projections}
\end{align*}

Composition From \( f : t\text{nat}^k \to t\text{nat}, \) \( g_i : t\text{nat}^n \to t\text{nat}, \) \( i = 1, \ldots, k \) define \( f(g_1, \ldots g_k) : t\text{nat}^n \to t\text{nat} : \\
(f(g_1, \ldots, g_k))(\vec{x}) = f(g_1(\vec{x}), \ldots, g_k(\vec{x})) \)
**Primitive Recursion** From \( g : \text{tnat}^n \to \text{tnat} \), \( h : \text{tnat}^{n+2} \to \text{tnat} \)

define \( PR(g, h) : \text{tnat}^{n+1} \to \text{tnat} \) such that:

\[
PR(g, h)(\vec{z}, \vec{y}) = g(\vec{y})
\]

\[
PR(g, h)(s(x), \vec{y}) = h(f(x, \vec{y}), x, \vec{y})
\]

With the usual notation, \( PR(g, h) \) is a function \( f \) defined recursively as follows:

\[
f = \\
z, \vec{y} \quad \Rightarrow \quad g(\vec{y}) \\
s(x), \vec{y} \quad \Rightarrow \quad h(f(x, \vec{y}), x, \vec{y})
\]
Examples

We rely on the usual notation and write the constant $z$ rather than the zero function $z$.

$$add = \quad \text{(addition)}$$

$z, y \Rightarrow y$

$s(x), y \Rightarrow s(add(x, y))$

$$mul = \quad \text{(multiplication)}$$

$z, y \Rightarrow z$

$s(x), y \Rightarrow add(mul(x, y), y)$

$$exp = \quad \text{(exponentiation)}$$

$x, z \Rightarrow s(z)$

$x, s(n) \Rightarrow mul(exp(x, n), x)$

Can go on to describe towers of exponentials, ... Complexity of programmable functions still very high!
Exercise (primitive recursive programming)

Define primitive recursive functions on tally numbers:

1. to **decrement** by one (with $0 - 1 = 0$),
2. to **subtract** (with $x - y = 0$ if $y > x$),
3. to compute an **if-then-else**, 
4. to compute the **minimum** of two numbers.

**NB** There is a **trade-off**: termination is for free but some **functions** cannot be represented and some **algorithms** are more difficult or impossible to represent.
A limitation of primitive recursion

Primitive recursion is a bit of a straightjacket to guarantee termination. Consider:

\[
\begin{align*}
\min &= \\
z, y &\Rightarrow z \\
s(x), z &\Rightarrow z \\
s(x), s(y) &\Rightarrow s(\min(x, y))
\end{align*}
\]

L. Colson has shown that no primitive recursive definition of \( \min \) produces an algorithm reducing in time \( \min(|x|, |y|) \).
From unary to binary notation

• **Size:** $|c| = 0$, $|c(v_1, \ldots, v_n)| = 1 + \sum_{i=1,\ldots,n} |v_i|$.  

• From a practical point of view, unary notation requires **too much space**.  

• From a complexity point of view unary notation is rather odd. Because the input is so **large** complexities are unexpectedly **low** (*e.g.*, look at addition).  

• We consider a variant of primitive recursion on **binary words**:

  $$
type \ bnat = 0 \ of \ bnat \ | \ 1 \ of \ bnat \ | \ \epsilon
$$

• Then we try to **bound the complexity** of the definable functions.
Recursion on notation (RN)

Basic functions

\[ e(x) = \epsilon \]  \hspace{2cm} \text{empty word}

\[ s_i(x) = ix, i = 0, 1 \]  \hspace{2cm} \text{successors}

\[ p_i(x_1, \ldots, x_n) = x_i, i = 1, \ldots, n \]  \hspace{2cm} \text{projections}

Composition  As for primitive recursion,

\[ (f(g_1, \ldots, g_k))(\vec{x}) = f(g_1(\vec{x}), \ldots, g_k(\vec{x})) \]
Recursion on notation  From $g : bnat^n \rightarrow bnat$, \\
    $h_i : bnat^{n+2} \rightarrow bnat$, $i = 1, 2$ define \\
    $BRN(g, h_0, h_1) : bnat^{n+1} \rightarrow bnat$ such that:

\[
\begin{align*}
BRN(g, h_0, h_1)(\epsilon, \vec{y}) &= g(\vec{y}) \\
BRN(g, h_0, h_1)(0(x), \vec{y}) &= h_0(f(x, \vec{y}), x, \vec{y}) \\
BRN(g, h_0, h_1)(1(x), \vec{y}) &= h_1(f(x, \vec{y}), x, \vec{y})
\end{align*}
\]

In other words, we define a function $f$ by recursion as follows:

\[
\begin{align*}
f &= \\
\epsilon, \vec{y} &\Rightarrow g(\vec{y}) \\
0(x), \vec{y} &\Rightarrow h_0(f(x, \vec{y}), x, \vec{y}) \\
1(x), \vec{y} &\Rightarrow h_1(f(x, \vec{y}), x, \vec{y})
\end{align*}
\]
Exercise (programming by recursion on notation)

• Represent binary numbers as binary words where the least significative digit is on the left.

• Define a function that takes a binary word and removes all ‘0’ that do not occur on the left of a 1 (hence \(\epsilon\) can be taken as the canonical representation of zero).

• Show that the functions division by 2, modulo 2, successor, if-then-else, predecessor, number of digits can be defined by recursion on notation.
• Suppose *add* is a function that implements *addition* (we skip this because it is quite technical, for instance, Rose’s book takes 22 steps to define addition!). Define a function *mult* by recursion on notation to implement the multiplication.
Computing primitive recursion

• Suppose $v_k = i_k \ldots i_1 e$.

• Here is a simple for loop that computes $f(v_k, \vec{y})$:

\[
\begin{align*}
    r & := g(\vec{y}); \\
    \text{for } j = 1 \text{ to } k \text{ do} \\
    & \quad r := h_{i_j}(r, v_{j-1}, \vec{y}) \quad \text{od} \\
    \text{return } r
\end{align*}
\]

Problem Suppose that $h_i$ and $g$ can be computed in polynomial time. Can we conclude that $f$ can be computed in polynomial time?
What can go wrong

Consider first the function $d$ doubling the size of its input:

$$d =
\epsilon \Rightarrow 1\epsilon
i(x) \Rightarrow i(i(d(x)))$$

Then consider the function $e$

$$e =
\epsilon \Rightarrow 1\epsilon
i(x) \Rightarrow d(e(x))$$

These functions are definable by recursion on notation (exercise!) and the size of $|e(x)|$ is exponential in $|x|$. Iterating $|x|$ times polynomial time operations can generate data whose size is not polynomial in $|x|$. 
Restricting recursion to capture polynomial time
Bounded recursion on notation (BRN)

In the definition of recursion on notation

\[ f(x, \bar{y}) = \]

\[ \epsilon, \bar{y} \Rightarrow g(\bar{y}) \]

\[ 0(x), \bar{y} \Rightarrow h_0(f(x, \bar{y}), x, \bar{y}) \]

\[ 1(x), \bar{y} \Rightarrow h_1(f(x, \bar{y}), x, \bar{y}) \]

we require that a polynomial \( S_f \) with non-negative coefficients is given such that:

\[ |f(x, \bar{y})| \leq S_f(|x|, |\bar{y}|) . \]

**NB** \( |f(x, \bar{y})| \) is the size of the value to which \( |f(x, \bar{y})| \) reduces.
Cobham’s theorem (1964)

The **functions** computable by an algorithm in BRN are exactly those computable in PTIME.

- If we can define an **algorithm by BRN** then we can compute its result in PTIME.

- If there is a **PTIME algorithm** in some Turing-equivalent formalism then we can compile it to an **algorithm in BRN** that computes the same function.
Remark

• It is quite possible to program a function that takes \textit{exponential time} and runs in \textit{polynomial space} (never going twice through the same configuration!).

• For instance, take a function that \textit{counts} from $x = 0^n \epsilon$ to $1^n \epsilon$.

• Implicitly, Cobham’s theorem states that as long as we stick with BRN such function \textit{cannot} be programmed.

• The problem is \textit{not} the size of the data (the identity function gives the bound!) but the fact that the recursion mechanism is not compatible with primitive recursion on notation.
Algorithms in BRN can be computed in PTIME (1/4)

First prove by induction on the definition of a function $f$ in BRN that there is a polynomial $S_f$ such that for all $v_1, \ldots, v_n$:

$$|f(v_1, \ldots, v_n)| \leq S_f(|v_1|, \ldots, |v_n|).$$

- Clear for the basic functions and for BRN.
- Composition:

$$ (f \circ (g_1, \ldots, g_k))(\vec{x}) = f(g_1(\vec{x}), \ldots, g_k(\vec{x})) $$

By inductive hypothesis

$$|g_i(\vec{x})| \leq S_{g_i}(|\vec{x}|)$$

Let $S$ be a polynomial that bounds all $S_{g_i}$. 
Algorithms in BRN can be computed in PTIME (2/4)

Applying again the inductive hypothesis:

\[
|f(g_1(\vec{x}), \ldots, g_k(\vec{x}))| \leq S_f(|g_1(\vec{x})|, \ldots, |g_k(\vec{x})|)
\leq S_f(S_{g_1}(|\vec{x}|), \ldots, S_{g_k}(|\vec{x}|))
\leq S_f(S(|\vec{x}|), \ldots, S(|\vec{x}|))
\]

and the composition of polynomials is a polynomial.

NB Thus data computed by BRN has size polynomial in the size of the input.
Algorithms in BRN can be computed in PTIME (3/4)

Next, prove by induction on the definition of a function $f$ in BRN that there is a polynomial $T_f$ such that $f(u_1, \ldots, u_n)$ can be computed in time $T_f(|u_1|, \ldots, |u_n|)$.

**Interesting case recursion** Consider again the for loop that computes primitive recursion, where $v_j = i_j \cdots i_1 \epsilon$, $1 \leq j \leq k$:

$$
\begin{align*}
r &:= g(\vec{y}); \\
\text{for } j = 1 \text{ to } k \text{ do} \\
r &:= h_{i_j}(r, v_j, \vec{y}) \text{ od} \\
\text{return } r
\end{align*}
$$

For all steps $j$, we have that: $|r| \leq S_f(k, |\vec{y}|)$. 

Algorithms in BRN can be computed in PTIME (4/4)

• Let $T_h$ be a polynomial that bounds both $T_{h_0}$ and $T_{h_1}$.

• Then the computation of the $k$ steps is performed in at most:

$$k \cdot T_h(S_f(k, |\vec{y}|), k, |\vec{y}|) = |x| \cdot T_h(S_f(|x|, |\vec{y}|), |x|, |\vec{y}|)$$

which is a polynomial in $|x|, |\vec{y}|$. 
All TM running in PTIME can be simulated (1/7)

• Let $M = (\Sigma, Q, q_o, F, \delta)$ be a Turing machine with
  – $\Sigma$ alphabet,
  – $Q$ states, $q_o \in Q$ initial state,
  – $F \subseteq Q$ final states,
  – $\delta : \Sigma \times Q \to \Sigma \times Q \times \{L, R\}$ transition function.

• Obviously, elements in $\Sigma$ and $Q$ can be encoded as binary words of length $\lceil \lg(\#\Sigma) \rceil$ and $\lceil \lg(\#Q) \rceil$, respectively.
All TM running in PTIME can be simulated (2/7)

• The configuration of a TM can be described by a tuple $(q, h, l, r)$ where:
  – $q$ is the current state.
  – $h$ is the character read.
  – $l$ are the characters on the left hand side of the head.
  – $r$ are the characters on the right hand side of the head.
All TM running in PTIME can be simulated (3/7)

- Then we have to define a step function that simulates one step of a Turing machine while working on the encodings of states and characters.

- Informally, the function step is a case analysis corresponding to the finite table defining the transitions of the TM. E.g.

  \[ \text{step}(q, h, l, r) = \ldots \]

  \[ q = 01\epsilon, h = 1\epsilon, l = 0l', r = r \Rightarrow (11\epsilon, 0\epsilon, l', 0r) \]

describes the situation where being in state 01 and reading 1, we go in state 11, write 0, and move to the left.
All TM running in PTIME can be simulated (4/7)

- The only technical difficulty is that tuples are not a primitive data structures in our formalisation.
- However, using the arithmetic functions, we can program pairing of natural numbers and the related projections.
- Alternatively (and more naturally) one could extend the framework with a pairing constructor.
- We ignore these problems and assume a function step that takes a tuple \((q, h, l, r)\) and returns the tuple \((q', h', l', r')\) describing the following state.
All TM running in PTIME can be simulated (5/7)

Now comes a key idea:

- We have an initial configuration $x_0$
- We have a function $step$ such that
  \[ |step(x)| \leq |x| + 1 \]
- We want to iterate $step$ on $x_o$ at least $P(|x_o|)$ times where $P$ is a polynomial of degree $k$.

**NB** W.l.o.g., one may assume the TM loops after reaching the final state so that running it longer does not hurt.
All TM running in PTIME can be simulated (6/7)

- We assume an expansion function $exp$ such that for all $k$ there is an $m$ such that

$$|exp^m(x)| \geq |x|^k.$$  

E.g. it is enough to iterate a function that squares the size of its entry.

- Then we define a function $it$ as:

$$it(\epsilon, x) = x$$

$$it(i \cdot t, x) = \text{step}(it(t, x))$$

- This is a definition by bounded recursion on notation since writing $S_{it}(n, m) = n + m$ we have:

$$|\text{it}(t, x)| \leq S_{it}(|t|, |x|).$$
All TM running in PTIME can be simulated (7/7)

• In the framework of TM, we just have to replace it with an *exec* function of the following shape:

\[
exec = \\
\epsilon, q, h, l, r \Rightarrow (q, h, l, r) \\
i x, q, h, l, r \Rightarrow \text{step}(exec(x, q, h, l, r))
\]

• To summarize, given a TM running in \( P \) time, for any input \( x \) we:
  – initialize a **counter** to a value \( v \) such that \( |v| \geq P(|x|) \).
  – perform a BRN on the counter thus iterating the step function \( |v| \) times.

**NB** The iteration works on the **length of the counter** and not on its binary representation. O.w. the definition would not be by BRN and termination could take exponential time!
Remark 1 (PTIME by time-out)

The second part of the proof suggests that there is a trivial way of building PTIME algorithms.

- Take any program.
- Instrument it so that it keeps a counter that stops after a number of steps which is polynomial in the size of the input (for a fixed polynomial).

Of course, the problem with this ‘time-out’ approach is that we have no idea whether the program will produce interesting answers before running it.

NB While it is possible to build a programming language that computes exactly the PTIME functions it is not possible to build one that contains exactly the PTIME programs (this set is undecidable).
Remark 2 (polynomial bound?)

Cobham’s result restricts the programmer to primitive recursion and it provides no clue on how to find a polynomial bound on size.
Safe recursion on notation (SRN)

The high-complexity of programs defined by RN depends on the fact that we have nested recursions, i.e., the result of a RN can be used as the main argument of another RN as in:

\[
\begin{align*}
d &= \\
\epsilon &\Rightarrow 1\epsilon \\
i(x) &\Rightarrow i(i(d(x)))
\end{align*}
\]

\[
\begin{align*}
e &= \\
\epsilon &\Rightarrow 1\epsilon \\
i(x) &\Rightarrow d(e(x)) \Leftarrow \text{('bad' recursion)}
\end{align*}
\]

**Key insight:** if we forbid this by a **syntactic mechanism** then **data size stays polynomial** and it is still possible to define all functions computable in PTIME.
A syntactic mechanism (Bellantoni-Cook 1992)

- Functions’ arguments are partitioned into two zones (separated by ‘;’)

\[ f(x_1, \ldots, x_n; y_1, \ldots, y_m) \quad n, m \geq 0 \]

The ones on the left are called normal and those on the right safe.

- The invariant is that for all functions \( f \) defined by SRN there is a polynomial \( P_f \) such that:

\[ |f(x_1, \ldots, x_n; y_1, \ldots, y_m)| \leq P_f(|x_1|, \ldots, |x_n|) + \max(|y_1|, \ldots, |y_m|) \]

In particular, if \( f \) has no normal arguments then the size of its result is bound by the size of its arguments up to an additive constant.

NB A related mechanism called tiering was proposed by Leivant and Marion at about the same time.
Safe recursion and composition

Unlike in BRN, in SRN the existence of the polynomial is **guaranteed** by the way recursion and composition are restricted.

**Safe recursion**  Assuming $g, h_0, h_1$ are SRN functions, we can define $f$ as:

\[
\begin{align*}
  f &= \\
  (\epsilon, \vec{x}; \vec{y}) &\Rightarrow g(\vec{x}; \vec{y}) \\
  (0x, \vec{x}; \vec{y}) &\Rightarrow h_0(x, \vec{x}; \vec{y}, f(x, \vec{x}; \vec{y})) \\
  (1x, \vec{x}; \vec{y}) &\Rightarrow h_1(x, \vec{x}; \vec{y}, f(x, \vec{x}; \vec{y}))
\end{align*}
\]

Main argument:  Normal zone

Recursive call:  Safe zone
**Safe composition**  Assuming, \( f, g_1, \ldots, g_k, h_1, \ldots, h_l \) are SRN functions we can define their composition:

\[
f( g_1(\vec{x};), \ldots, g_k(\vec{x};) ; h_1(\vec{x};\vec{y}), \ldots, h_l(\vec{x};\vec{y}) )
\]

Expressions plugged in the normal zone do not depend on arguments in the safe zone.
Example and Non-Example of SRN

\[ d(x;) = \]
\[ \epsilon \quad \Rightarrow 1(;\epsilon) \]
\[ i(;x) \quad \Rightarrow i(;i(;d(x;)))) \]

\[ e(x;) = \]
\[ \epsilon \quad \Rightarrow 1\epsilon \]
\[ i(;x) \quad \Rightarrow d(e(x;)) \quad \Leftarrow \text{Not SRN!} \]

\( e(x;) \) should go to the safe zone, while \( d \) is waiting for an argument in the normal zone; nested recursions do not compose.
Summary

• A function defined by primitive recursive or recursion on notation is guaranteed to terminate.

• A function defined by bounded (or safe) recursion on notation is guaranteed to terminate in polynomial time.

• Key insights:
  – \( PTIME \) = polynomial data size + restricted recursion mechanism.
  – The polynomial data size can be guaranteed by avoiding nested recursions.

• Trade off: Guaranteed bounds vs. Programming freedom.
References (optional)

If you want to know more.


λ-calculus
History

• The $\lambda$-calculus was introduced around 1930 by Church as part of an investigation in the formal foundations of mathematics and logic.

• The related formalism of combinatory logic had been introduced some years earlier by Schönfinkel and Curry.

• While the foundational program was later relativized by such results as Gödel’s incompleteness theorem, $\lambda$-calculus nevertheless provided one of the concurrent formalizations of partial recursive functions (= computable functions).

• Logical interest in the $\lambda$-calculus was resumed by Girard’s discovery of the second order $\lambda$-calculus in the early seventies.
• In computer science, the interest in $\lambda$-calculus goes back to Landin (1966) and Reynolds (1970).

• The $\lambda$-notation is also important in LISP, designed around 1960 by MacCarthy.

• These pioneering works have eventually led to the development of functional programming languages like Scheme, ML or Haskell.

• In parallel, Scott and Strachey used $\lambda$-calculus as a meta-language for the description of the denotational semantics of programming languages.
Syntax

- We consider ‘pure’ functional expressions

\[ M ::= id \mid (\lambda id.M) \mid (MM) \]

where \( id ::= x \mid y \mid \ldots \)

- This is a minimal language where the only operations allowed are abstraction \( \lambda x.M \) and application \( MN \).

- In ML, one would write \( \lambda x.M \) as

  \[ \text{function } x \rightarrow M \]

NB One can write all \( \lambda \)-terms in ML-syntax, however only some of them will be typable, and moreover the reduction of the typable ones will follow a particular strategy (call-by-value).
• A number of programming operations can be introduced as **syntactic sugar**. For instance, the operation \( \text{let } x = M \text{ in } N \) can be represented as \((\lambda x. N)M\).

• The abstraction \( \lambda x. M \) **binds** the variable \( x \) in the term \( M \) just as the quantified first-order formula \( \forall x. \phi \) binds \( x \) in \( \phi \).

• We denote with \( FV(M) \) the variables occurring **free** in the term \( M \).
Substitution

Because of bound variables, the substitution $[N/x]M$ must be defined with some care (on the size $|M|$ of $M$):

\[
\begin{align*}
[N/x]x &= N \\
[N/x]y &= y \text{ if } y \neq x \\
[N/x](M_1 M_2) &= [N/x]M_1 [N/x]M_2 \\
[N/x](\lambda y.M) &= \lambda z.[N/x][z/y]M \text{ if } z \notin FV(MN)
\end{align*}
\]

**NB** This is well defined because $|[z/y]M| = |M|$ !
Contexts

A (one-hole) context \( C \) is defined by:

\[
C ::= \square \mid \lambda id.C \mid CM \mid MC
\]

We write \( C[N] \) for the term obtained by replacing the hole \( \square \) with the term \( N \) without paying attention to the potential capture of variables. Formally:

\[
[N] = N \\
(\lambda x.C)[N] = \lambda x.C[N] \\
(CM)[N] = C[N]M \\
(MC)[N] = MC[N]
\]
\[ \alpha \text{-renaming} \]

- \( \lambda \)-terms (like formulae or integrals!) are always manipulated up to the \textit{renaming of bound variables}.

- Formally, we define a binary relation \( \equiv \) on terms called \( \alpha \text{-conversion} \) which is the least equivalence relation containing the following pairs:

\[
C[\lambda x.M] \equiv C[\lambda y.[y/x]M] \quad \text{if } y \notin FV(M)
\]
\textbf{\(\beta\)-reduction and \(\beta\)-conversion}

The \textbf{\(\beta\)-rule} is the following relation between \(\lambda\)-terms:

\[(\beta) \quad C[(\lambda x. M)N] \rightarrow C[[N/x]M],\]

where \(C\) is an occurrence context and \(M, N\) are arbitrary terms.

The subterm \((\lambda x. M)N\) is called a \textbf{\(\beta\)-redex} (the subterm which is transformed).

We denote with \(\equiv_\beta\) the associated \textbf{conversion} relation, \textit{i.e.}, \(\leftrightarrow_\beta\).

\textbf{NB} This is \textbf{NOT} a TRS but there are proposals to turn it into a TRS by making the substitution operation explicit.
Examples

Here are some **basic λ-terms**:

\[
\begin{align*}
I &= \lambda x.x \\
K &= \lambda x, y.x \\
S &= \lambda x, y, z.xz(yz) \\
\Delta &= \lambda x.xx \\
\Delta_f &= \lambda x.f(xx)
\end{align*}
\]

Here are some **examples of β-reduction** (up to α-renaming!):

\[
\begin{align*}
II &\rightarrow I \\
SKK &\rightarrow I \\
\Delta\Delta &\rightarrow \Delta\Delta \\
\Delta_f\Delta_f &\rightarrow f(\Delta_f\Delta_f)
\end{align*}
\]

**NB** \(\lambda x_1, \ldots, x_n.M \equiv \lambda x_1 \ldots \lambda x_n.M\) and application associates to the left, *i.e.*, \(M_1M_2\ldots M_n \equiv (\cdots(M_1M_2)\cdots M_n)\).
Exercise (characterisation normal forms)

Let \( NF \) be the smallest set of \( \lambda \)-terms such that:

\[
M_i \in NF \quad i = 1, \ldots, k
\]

\[
\lambda x_1 \ldots x_n . x M_1 \ldots M_k \in NF
\]

Show that \( NF \) is exactly the set of \( \lambda \)-terms in \( \beta \)-normal form.
Exercise (fixed-point combinator)

Let $Y = \lambda f. \Delta_f \Delta_f$. Show that

$$YM =_{\beta} M(YM)$$

This is known as Curry’s fixed point combinator.
Exercise (another fixed point combinator)

Turing’s fixed point combinator is defined by:

\[
Y_T = (\lambda xy. y(xx))(\lambda xy. y(xx))
\]

Show that that \(Y_T f\) is not only convertible to, but reduces to, \(f(Y_T f)\).
Overview of results on the $\lambda$-calculus to come

- The $\lambda$-calculus is **confluent**.

- Every program can be **represented** in the $\lambda$-calculus.

- Typing is **preserved** by reduction.

- The typed $\lambda$-calculus **terminates**.

- Simple and stratified polymorphic types can be **automatically inferred**.
Confluence of the $\lambda$-calculus
Exercise (local confluence)

First show that:

1. If $M \rightarrow M'$ then $[N/x]M \rightarrow [N/x]M'$.
2. If $N \rightarrow N'$ then $[N/x]M \xrightarrow{*} [N'/x]M$.

Then conclude that $\beta$-reduction is **locally confluent**:

$$\forall M, N, P \ (M \rightarrow N, \ M \rightarrow P)$$

$$\exists Q \ (N \xrightarrow{*} Q, \ P \xrightarrow{*} Q)$$
Church-Rosser theorem

\(\beta\)-reduction is **confluent**. That is:

\[
\forall M, N, P \ (M \xrightarrow{\ast} N \quad M \xrightarrow{\ast} P) \\
\exists Q \ (N \xrightarrow{\ast} Q \quad P \xrightarrow{\ast} Q)
\]
Towards a proof

• Let $\leftrightarrow$ be the reflexive closure of $\rightarrow$. Notice that a strong confluence property such as:

$$
\forall M, N, P \ (M \rightarrow N, \ M \rightarrow P) \\
\exists Q \ (N \leftrightarrow Q, \ P \leftrightarrow Q)
$$

fails because as a result of $\beta$-reduction redexes can be duplicated.

• A key insight due Tait and Martin-Löf amounts to define a parallel reduction relation $\Rightarrow$ with the following properties:
  1. $\rightarrow \subset \Rightarrow \subset \ *$.
  2. The strong confluence property holds for $\Rightarrow$ (and $\Rightarrow$ is simple enough for a proof to go through!)
Definition Parallel Reduction

\[
\begin{align*}
M \Rightarrow M' & \quad N \Rightarrow N' \\
\hline
M \Rightarrow M & \quad (\lambda x.M)N \Rightarrow [N'/x]M'
\end{align*}
\]

\[
\begin{align*}
M \Rightarrow M' & \quad N \Rightarrow N' \\
\hline
MN \Rightarrow M'N' & \quad \lambda x.M \Rightarrow \lambda x.M'
\end{align*}
\]
Exercise (on parallel reduction)

- Let $M = (\lambda x.Ix)(II)$ where $I = \lambda z.z$.
- What is the minimum number of parallel reductions needed to reduce $M$ to $I$?
Confluence of parallel reduction (1/3)

First observe the following structural properties:

\[ \begin{align*} 
\lambda x. M &\Rightarrow N \\
N \equiv \lambda x. M', &\quad M \Rightarrow M'
\end{align*} \]

\[ \begin{align*} 
MN &\Rightarrow L \\
(L \equiv M'N', &\quad M \Rightarrow M', \quad N \Rightarrow N') \quad \text{or} \\
(M \equiv \lambda x. P, &\quad P \Rightarrow P', \quad N \Rightarrow N', \quad L \equiv [N'/x]P')
\end{align*} \]
Confluence of parallel reduction (2/3)

Next prove the following substitution property by induction on the definition of $M \Rightarrow M'$:

\[
\frac{M \Rightarrow M' \quad N \Rightarrow N'}{[N/x]M \Rightarrow [N'/x]M'}
\]
Confluence of parallel reduction (3/3)

Finally, show a strong confluence property:

\[
\forall M, N_1, N_2 \ (N_1 \leftrightarrow M \Rightarrow N_2) \\
\exists P \ (N_1 \Rightarrow P \Leftarrow N_2)
\]

One can proceed by induction on \( M \Rightarrow N_1 \) and case analysis on \( M \Rightarrow N_2 \) to close the diagram.

Finally, from

\[
\rightarrow_\beta \subset \Rightarrow \subset \rightarrow^*_\beta
\]

conclude that \( \rightarrow_\beta \) is confluent.
\( \eta \)-rule

In addition to \( \beta \), another rule is often considered:

\[
(\eta) \quad C[\lambda x. M x] \rightarrow C[M] \quad (x \not\in FV(M)).
\]

This is an extensionality rule, asserting that ‘every term is a function’ (if it is read backwards).
Exercise (confluence for $\beta\eta$)

1. Show that $\eta$ reduction is strongly **confluent** in the following sense:

$$\frac{M \xrightarrow{\eta} N_i \quad i = 1, 2 \quad N_1 \not\equiv N_2}{\exists P \ (N_i \xrightarrow{\eta} P, \ i = 1, 2)}$$

2. Show that $\beta$ and $\eta$ reductions **commute** in the following sense:

$$\frac{M \xrightarrow{\beta} N_1 \quad M \xrightarrow{\eta} N_2 \quad N_1 \not\equiv N_2}{\exists P \ (N_1(\xrightarrow{\eta})^* P, \ N_2 \xrightarrow{\beta} P)}$$

3. Show that $(\xrightarrow{\beta})^*$ and $(\xrightarrow{\eta})^*$ reductions commute in the following sense:

$$\frac{M(\xrightarrow{\beta})^* N_1 \quad M(\xrightarrow{\eta})^* N_2}{\exists P \ (N_1(\xrightarrow{\eta})^* P, \ N_2(\xrightarrow{\beta})^* P)}$$

4. Conclude that $\beta\eta$ reduction is **confluent**.
Example (non-confluence)

- Here is an extension of the $\lambda$-calculus that does not preserve confluence (proof omitted). We add to the language a constant $D$ and the rule

\[ Dxx \rightarrow x \]

- This is actually a simplification of a natural rule called surjective paring which also leads to a non-confluent system:

\[ D(Fx)(Sx) \rightarrow x \]

Here $D$, $F$, $S$ are constants where intuitively $D$ is the pairing while $F$ and $S$ are the first and second projection.

- The problem is confluence (not local confluence). Indeed a surjective pairing rule is introduced in terminating typed $\lambda$-calculi.
Combinatory Logic

• We consider a binary operation @ and two constants $K$ and $S$.

• As in the $\lambda$ calculus, we write $MN$ for @($M$, $N$) and associate to the left.

• We have two term rewriting rules:

  $Kxy \rightarrow x$

  $Sxyz \rightarrow xz(yz)$

• We abbreviate $SKK$ as $I$. 
• One can simulate the $\lambda$-abstraction of a term $M$ of combinatory logic as follows:

\[
\begin{align*}
\lambda x.x &= I \\
\lambda x.M &= KM \quad \text{if } x \notin \text{Var}(M) \\
\lambda x.MN &= S(\lambda x.M)(\lambda x.N)
\end{align*}
\]

• By induction on $M$ verify that:

\[
(\lambda x.M)N \xrightarrow{*} [N/x]M
\]
• Keeping in mind that CL is a term-rewriting system, show that it is **locally confluent**.

• Further, adapt the **method of parallel reduction** to show that the system is **confluent**.

• CL induces a **weaker notion of equality** than the one induced by $\beta$ conversion. For instance, consider the translations in CL of $(\lambda z. (\lambda x. x) z)$ and $\lambda z. z$. Check that the translations do not have a common reduct.
Programming in the $\lambda$-calculus
The λ-calculus is ‘Turing equivalent’

- All partial recursive functions can be represented in the (type free) λ-calculus. Thus the formalism is Turing equivalent.

- Or one could say that Turing machines are Church equivalent. Indeed one speaks of the Turing-Church thesis.

- The simplest proof of this fact relies on the following definition of the partial recursive functions:

  The partial recursive functions is the smallest set of functions on (vectors of) natural numbers which contains the basic functions (zero, successor, projections) and is closed under composition, primitive recursion, and minimalisation
Minimalisation (fact or reminder)

Given a total function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ one defines a partial function $\mu(f) : \mathbb{N}^k \to \mathbb{N}$ by minimalisation as:

$$\mu(f)(x_1, \ldots, x_k) = \begin{cases} 
    x_0 & \text{if } x_0 = \min\{x \in \mathbb{N} \mid f(x, x_1, \ldots, x_n) = 0\} \\
    \uparrow & \text{if } \forall x \ f(x, x_1, \ldots, x_n) > 0
\end{cases}$$

NB Recursive definitions provide a direct mechanism to mimic definitions by minimalisation.
Number representation

- **Numbers** (a.k.a. Church numerals) are represented as follows:

\[ \overline{n} = \lambda f.\lambda x. (f \cdots (fx) \cdots) \]

where \( f \) is applied \( n \) times.

- In some sense this is similar to **tally natural numbers**. When working with second order types (system F) the inductive definition of natural numbers actually suggests their representation in the \( \lambda \)-calculus.
Function representation

A term $F$ represents a partial function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ if for all $n_1, \ldots, n_k \in \mathbb{N}$:

$$f(n_1, \ldots, n_k) = m \quad \text{iff} \quad Fn_1 \cdots n_k = m$$

$$f(n_1, \ldots, n_k) \uparrow \quad \text{iff} \quad Fn_1 \cdots n_k \text{ has no (head) normal form.}$$
Exercise (representation of basic functions)

Define terms $S$, $A$, and $M$ that represent the *successor*, *addition*, and *multiplication* functions.
Booleans and pairing

• We can also define booleans $T$, $F$ and an if-then-else $ITE$ as follows:

$$T = \lambda x.\lambda y.x \quad F = \lambda x.\lambda y.y \quad ITE = \lambda x.\lambda y.\lambda z.xyz$$

• Further, one can represent pairing $P$ and projections $P_1$, $P_2$:

$$P = \lambda x.\lambda y.\lambda z.zxy \quad P_1 = \lambda p.p(\lambda x,y.x) \quad P_2 = \lambda p.p(\lambda x,y.y)$$
Recursive definitions

In general a **recursive definition** of a function such as

\[
\text{letrec } f(x) = M \text{ in } N
\]

where \( f \) may appear in \( M \) and \( N \) is coded as:

\[
(\lambda f. N)(Y(\lambda f. \lambda x. M))
\]

It follows that one can represent definitions by **primitive recursion** and **minimalisation**.
Summary (and some perspectives)

• The λ-calculus is a (relatively) simple and neat theory of effective (computable) functions which plays a central role in the formal study of programming languages.

• As a matter of fact, it embodies many concepts which arise in high-level programming languages such as:
  – Notion of (higher-order) procedure.
  – Static scoping.
  – Recursive definitions.
  – Evaluation strategies.
• This foundational character is even stronger when the calculus is enriched with types which are another important structuring feature of programming languages (data types, modules,…)

• (Typed) λ-calculi have been extensively used to provide semantics to a variety of programming constructs. This approach is called denotational semantics (ML belongs to this tradition).
Finally, for suitable choices of the types, the λ-calculus is actually a \textbf{logical system} (two examples are Martin-Löf type theory and the Calculus of Constructions) following the so-called \textbf{Curry-Howard correspondance} which goes as follows:

<table>
<thead>
<tr>
<th>λ-calculus</th>
<th>proof system</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>proposition</td>
</tr>
<tr>
<td>λ-term</td>
<td>proof</td>
</tr>
<tr>
<td>reduction</td>
<td>proof normalisation</td>
</tr>
</tbody>
</table>
References


This is **the** standard reference for the ‘pure’ (type-free) $\lambda$-calculus. You can just skim through the first introductory chapters (chapter 3 in particular) to have an idea of the variety of results.
Simple types
Syntax: types and contexts

• We define the collection of simple types as follows

\[ A ::= b \mid tid \mid (A \to A) \]

where \( b \) is a basic type (there can be more) and \( tid ::= t \mid s \mid \ldots \) are type variables.

• An environment \( \Gamma \) is a set of pairs \( \{x_1 : A_1, \ldots, x_n : A_n\} \) where all variables \( x_1, \ldots, x_n \) are distinct.

• We use \( \Gamma, x : A \) as an abbreviation for \( \Gamma \cup \{x : A\} \) where \( x \) is not in \( \Gamma \).
Terms and typing rules

\[(\text{asmp})\]

\[
\frac{x : A \in \Gamma}{\Gamma \vdash x : A}
\]

\[(\rightarrow_I)\]

\[
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A. M : A \rightarrow B}
\]

\[(\rightarrow_E)\]

\[
\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}
\]

**NB** We have: (1) a structural/identity rule: \((\text{asmp})\), (2) an introduction rule: \((\rightarrow_I)\), and (3) an elimination rule: \((\rightarrow_E)\). This presentation style comes from logic where it is called **natural deduction** (Dag Prawitz).
Exercise (types and propositions)

1. Show that if $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ is derivable then 
   $(A_1 \to \cdots (A_n \to B) \cdots)$ is a tautology of propositional logic 
   where we interpret $\to$ as implication and atomic types as 
   propositional variables.

2. Conclude that there are types $A$ which are not inhabited, 
   i.e., there is no (closed) term $M$ such that $\emptyset \vdash M : A$. 

Exercise (inhabited types)

1. Show that there is no term $M$ such that:

   $\emptyset \vdash M : (b \rightarrow b) \rightarrow b$

2. Write $A \rightarrow b$ as $\neg A$. Show that there are terms $N_1$ and $N_2$ such that:

   $\emptyset \vdash N_1 : A \rightarrow (\neg\neg A)$
   $\emptyset \vdash N_2 : (\neg\neg\neg A) \rightarrow (\neg A)$
3. On the other hand, there are tautologies which are not inhabited! For instance, consider:

\[ A \equiv (((t \rightarrow s) \rightarrow t) \rightarrow t) \]

Show that there is no term \( M \) in normal form such that \( \emptyset \vdash M : A \) is derivable. This is enough because later we will show that all typable terms normalise to a term of the same type.

4. For another example, show that there is no term \( M \) in normal form such that \( \emptyset \vdash M : \neg\neg t \rightarrow t \) is derivable (the intuitionistic/constructive negation is not involutive!).
Alternative presentations: Curry style

We type pure (typeless) \(\lambda\)-terms. Thus the rule \((\rightarrow_I)\) becomes:

\[
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \rightarrow B}
\]

In this presentation, a term may have more than one type. Typically, type inference problems are studied in this framework.
Alternative presentations: Variable labelling

We label every variable with its type and we drop the context:

\[
\begin{align*}
\text{id} & ::= x \mid y \mid \ldots \\
M & ::= id^A \mid \lambda id^A.M \mid MM
\end{align*}
\]

\[
\begin{array}{ccc}
\rule{0pt}{25pt} x^A : A & \quad & M : A \to B & \quad & N : A \\
\hline
\end{array}
\quad
\begin{array}{c}
M N : B
\end{array}
\quad
\begin{array}{c}
\lambda x^A.M : A \to B
\end{array}
\]

\[\text{274}\]
Alternative presentations: Term labelling

We label every term (not just the variables) with its type and we drop the context and the type:

\[
\begin{align*}
\text{id} & ::= x \mid y \mid \ldots \\
M & ::= \text{id}^A \mid (\lambda \text{id}^A.M)^A \mid (M M)^A
\end{align*}
\]

\[
\begin{array}{ccc}
x^A & M^{A\rightarrow B} & N^A \\
\hline
(A^{A\rightarrow B} \quad N^A)^B & (\lambda x^A.M^B)^{(A\rightarrow B)}
\end{array}
\]
Logical extensions of natural deduction

\[(\times I) \quad \frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2}\]

\[(\times E, 1) \quad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_1(M) : A_1} \quad (\times E, 2) \quad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_2(M) : A_2}\]

\[(+ I, 1) \quad \frac{\Gamma \vdash M : A_1}{\Gamma \vdash \text{inl}_{A_1 + A_2}(M) : A_1 + A_2} \quad (+ I, 2) \quad \frac{\Gamma \vdash M : A_2}{\Gamma \vdash \text{inr}_{A_1 + A_2}(M) : A_1 + A_2}\]

\[(+ E) \quad \frac{\Gamma \vdash M : (A_1 + A_2) \quad \Gamma \vdash N_i : A_i \to B \quad i = 1, 2}{\Gamma \vdash \text{case } M \ N_1 \ N_2 : B}\]
Non-Logical extensions of natural deduction

\[(Z) \quad \frac{}{\Gamma \vdash Z : nat}\]

\[(S) \quad \frac{\Gamma \vdash M : nat}{\Gamma \vdash SM : nat}\]

\[(Y) \quad \frac{\Gamma \vdash M : (A \to A)}{\Gamma \vdash YM : A}\]

**NB** With the rule (Y), every type \(A\) is **inhabited** by the closed term \(Y(\lambda x.x)\). Thus \(Y\) leads to logical inconsistency!
Invariance of typing under reduction (and progress)
Subject Reduction (motivations)

• If a term is well-typed, then by inspection of the rules we see, \textit{e.g.}, that the term cannot contain the application of a natural number to a function.

• However, to get static guarantees we must make sure that \textit{typing is invariant under reduction,}

\textit{i.e.}, if a term is well-typed and we reduce it then we still get a well-typed term.
Substitution lemma

To establish this property we note the following lemma.

If $\Gamma, x : A \vdash M : B$ and $\Gamma \vdash N : A$ then $\Gamma \vdash [N/x]M : B$. 
Proof of the substitution lemma

- By induction on the **height of the proof** of $\Gamma, x : A \vdash M : B$.
- For instance, suppose the **root of the proof** has the shape:

  $$
  \Gamma, x : A, y : B' \vdash M : B''
  $$

  $\Gamma, x : A \vdash \lambda y. M : (B' \rightarrow B'')$

  with $x \neq y$.

- Then by inductive hypothesis, $\Gamma, y : B' \vdash [N/x]M : B''$ and we conclude by ($\rightarrow_I$).
Subject Reduction (formal statement)

If $\Gamma \vdash M : A$ and $M \rightarrow_{\beta} N$ then $\Gamma \vdash N : A$. 
Proof of subject reduction

- Recall, that \( M \rightarrow_{\beta} N \) means:

\[
M =_{\alpha} C[(\lambda x. M_1)M_2] \quad N =_{\alpha} C[[M_2/x]M_1]
\]

- To prove subject reduction we proceed by induction on the structure of \( C \).

- The **basic case** follows directly from the substitution lemma.

- For the **inductive case** consider in turn the cases where: (1) \( C = \lambda y.C' \), (2) \( C = C'P \), and (3) \( C = PC' \).
Outcomes of a program

- In the ‘pure’ λ-calculus, we assume:
  
  \[ \text{normal forms} = \text{results} \]

- In applications, one often decomposes the results as follows:
  
  \[ \text{normal forms} = \text{values} \cup \text{stuck/erroneous config.} \]

- Thus a program (a closed term) has three possible outcomes:
  - It **returns a value.**
  - It reaches an **erroneous configuration.**
  - It **diverges.**
Progress

• Besides being invariant by reduction, a desirable property for a type system is that:

   Well-typed programs cannot go wrong

or at least that they go wrong in some expected way (e.g., division by zero).

• This property is often called progress, because in its simple form it requires that if a program is not a value then it can reduce (progress).

• The following exercise elaborates on this point.
Exercise (progress)

• Suppose we reconsider the non-logical extension of the simply typed λ-calculus with a basic type \textit{nat}, constants \textit{Z}, \textit{S}, \textit{Y}, and with the following fixed-point rule:

\[
C[YM] \rightarrow C[M(YM)]
\]

• Let a \textbf{program} be a closed typable term of type \textit{nat} and let a \textbf{value} be a term of the shape \((S \cdots (SZ) \cdots)\).

• Show that if \(P\) is a program in \textbf{normal form} (cannot reduce), then \(P\) is a \textbf{value}.
Termination of the simply typed $\lambda$-calculus
Degrees

We introduce some measure towards the goal of proving (weak) termination of the typable terms.

- The degree of a type is defined as follows:
  \[ \delta(t) = 1 \quad \delta(A \to B) = 1 + max(\delta(A), \delta(B)) \]

- The degree of a redex
  \[ \delta_r((\lambda x : A.M)N), \]
  is the degree of the type associated with the redex \((\lambda x : A.M)\).

- The degree of a term
  \[ \delta_t(M), \]
  is 0 if \(M\) is in normal form and the maximum of the degrees of the redexes contained in \(M\) otherwise.

NB A redex \(R\) is also a term and we have \(\delta_r(R) \leq \delta_t(R)\).
Degrees and substitution

If $x$ is of type $A$ then

$$\delta_t([N/x]M) \leq \max(\delta(A), \delta_t(M), \delta_t(N))$$

**Proof** The redexes in $[N/x]M$ fall in the following categories:

- The redexes already in $M$.
- The redexes already in $N$.
- New redexes arising by the substitution if $N \equiv \lambda y.N'$ and $M = C[xM']$. These redexes have degree $\delta(A)$. 
Degrees and reduction

If $M \rightarrow N$ then $\delta_t(N) \leq \delta_t(M)$.

**Proof** We apply the previous analysis.
Reduction strategy: maximal degree

• Consider a term $M$ which is not in normal form.

• Let $R$ be a redex of maximal degree $n$ in $M$ and such that all redexes contained in $R$ have lower degree.

• Then by reducing $R$ we obtain a term with strictly less redexes of degree $n$.

• We prove normalization by taking as measure:

\[ \mu(M) = (n, m) \]

with the lexicographic order (from left to right) where $n = \delta_t(M)$ and $m$ is the number of redexes of degree $n$. 
Towards strong normalization

- A $\lambda$-term $M$ is called strongly normalizable if all $\beta$-reductions starting from $M$ terminate (thus strong-normalisation is just a synonymous for termination!).

- We denote with $SN$ the set of strongly normalizable terms (cf. set $WF$ for RPO termination!).

- The size of a term $M$ is defined as usual:

  \[
  \begin{align*}
  size(x) &= 1 \\
  size(MN) &= size(M) + size(N) + 1 \\
  size(\lambda x. M) &= size(M) + 1
  \end{align*}
  \]

- If $M \in SN$, the maximal length of a derivation starting from $M$ is called the reduction depth of $M$, and is denoted $depth(M)$.

  **NB** This is well-defined because the reduction tree is finitely branching (König lemma!).
Strong normalization

**Theorem**  All simply typed $\lambda$-terms are $\rightarrow_{\beta}$-strongly normalizable.

**Proof**  The **key idea** is to interpret types as follows:

$$
\begin{align*}
[t] &= SN \\
[A \to B] &= \{ M \mid \forall N \in [A] \ (MN \in [B]) \}
\end{align*}
$$

**NB** Recall that for RPO, we showed:

$$s_1, \ldots, s_n \in WF \text{ implies } f(s_1, \ldots, s_n) \in WF$$
Properties of the interpretation

1. \([A] \subseteq SN\).

2. If \(N_i \in SN\) for \(i = 1, \ldots, k\) then \(xN_1 \cdots N_k \in [A]\).

3. If \([N/x]MM_1 \cdots M_k \in [A]\) and \(N \in SN\) then 
   \((\lambda x.M)NM_1 \cdots M_k \in [A]\).

NB These interpretations of types are called reducibility candidates. These are sets of of strongly normalisable terms (property 1) which contain at least the variables (and more) (property 2), and are closed under head expansions (property 3).
Proof of the properties

By induction on $A$.

Atomic types

1. By definition.

2. The reductions of $xN_1 \ldots N_k$ are just an interleaving of the reductions of $N_1, \ldots N_k$.

3. We have:

$$\text{depth}((\lambda x.M)NM_1 \ldots M_k) \leq \text{depth}(N) + \text{depth}([N/x]MM_1 \ldots M_k) + 1.$$
Functional types $A \to B$  Suppose $M \in \mathbb{[A \to B]}$.

1. By ind. hyp. $x \in \mathbb{[A]}$. Hence $Mx \in \mathbb{[B]} \subseteq SN$, by ind. hyp. This entails $M \in SN$.

2. Take $M = xN_1 \ldots N_k$ with $N_i \in SN$. Take $N_{k+1} \in \mathbb{[A]} \subseteq SN$. By ind. hyp. $xN_1 \ldots N_k N_{k+1} \in \mathbb{[B]}$.

3. Then:

$$
\frac{\mathbb{[N/x] MN_1 \ldots N_k} \in \mathbb{[A \to B]}
\forall N_{k+1} \in \mathbb{[A]} \Rightarrow \mathbb{[N/x] MN_1 \ldots N_k N_{k+1} \in [B]}
\forall N_{k+1} \in \mathbb{[A]} \Rightarrow (\lambda x.M)NN_1 \ldots N_k N_{k+1} \in \mathbb{[B]} \quad \text{(by ind. hyp.)}
(\lambda x.M)NN_1 \ldots N_k \in \mathbb{[A \to B]}
$$
Soundness of the interpretation

If $x_1 : A_1, \ldots, x_k : A_k \vdash M : B$ (in Curry style) and $N_i \in \llbracket A_i \rrbracket$ for $i = 1, \ldots, k$ then $[N_1/x_1, \ldots, N_k/x_k]M \in \llbracket B \rrbracket$. 
Corollary of soundness: Strong normalisation

• Suppose $x_1 : A_1, \ldots, x_k : A_k \vdash M : B$.

• We know $x_i \in \llbracket A_i \rrbracket$.

• By the soundness $M \in \llbracket B \rrbracket$.

• We know $\llbracket B \rrbracket \subseteq SN$. 
Proof of soundness

By induction on the typing proof.

\((asmp)\) Immediate by definition.

\((\rightarrow_E)\) By the interpretation of \(\rightarrow\).

\((\rightarrow_I)\) By the closure under head expansions of the interpretation.

\[
x : A \vdash M : B
\]
\[
\forall N \in \llbracket A \rrbracket \ [N/x]M \in \llbracket B \rrbracket \quad \text{(ind. hyp.)}
\]
\[
\forall N \in \llbracket A \rrbracket \ (\lambda x.M)N \in \llbracket B \rrbracket \quad \text{(saturation)}
\]
\[
(\lambda x.M) \in \llbracket A \rightarrow B \rrbracket
\]
Exercise (recursive types)

Assume a type $t$ satisfying the equation $t = t \rightarrow b$ and suppose we add a rule for typing up to type equality:

\[
\Gamma \vdash M : A \quad A = B
\]
\[
\Gamma \vdash M : B
\]

Show that in this case the following $\lambda$-term (Curry’s fixed point combinator) is typable (e.g., in Curry’s style):

\[
Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))
\]

Are terms typable in this system terminating?
Summary

• We regard Types as Propositions and \( \lambda \)-terms as their (constructive) Proofs.

• Typing is invariant under reduction: Subject Reduction and Progress properties.

• Typing guarantees termination: Weak and Strong Normalisation proofs (interpretation of types as reducibility candidates).
Recommended reading


You may skim through the first 6 chapters. Beware that the book includes a full treatment of logic (natural deduction and sequent calculus) which is beyond the scope of this course.
Type inference and its reduction to syntactic unification
The type inference problem

• Given a (pure) term $M$ and a context $\Gamma$, the **type inference problem** is the problem of checking whether there is a type $A$ such that $\Gamma \vdash M : A$.

• Given a (pure) term $M$, a **variant of the problem** is to look for a type $A$ and a context $\Gamma$ such that $\Gamma \vdash M : A$.

• Connected to the type inference problem is the problem of actually producing an **informative output**.

• Typically, if a term $M$ is typable, we are interested in a **synthetic representation of its types**, and if it is not, we look for an **informative error message**.
Overview of the reduction

- We present a polynomial time reduction of the type inference problem for the simple system à la Curry to the syntactic unification problem (as noted by Hindley in 1969).

- The existence of a most general unifier for the unification problem leads to the existence of a most general type for the type inference problem.
Reduction (formal definitions)

- A goal is a finite set $G$ of triples $(\Gamma, M, A)$ where $\Gamma$ is a context, $M$ a $\lambda$-term, and $A$ a simple type.

- We assume that all bound variables in $M$ are distinct, that all free variables occur in the context $\Gamma$, and that for every variable $x$ we have a type variable $t_x$.

- We define a reduction relation on pairs $(G, E)$. 
• Assuming $G = \{g\} \cup G'$ and $g \equiv (\Gamma, M, A) \notin G'$ all the rules produce a pair $(G' \cup G_g, E \cup E_g)$ where $G_g$ and $E_g$ are defined as follows:

<table>
<thead>
<tr>
<th>$g$</th>
<th>$G_g$</th>
<th>$E_g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\Gamma, x, A)$</td>
<td>$\emptyset$</td>
<td>${t_x = A}$</td>
</tr>
<tr>
<td>$(\Gamma, M_1 M_2, A)$</td>
<td>${(\Gamma, M_1, t_1 \to A), (\Gamma, M_2, t_1)}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$(\Gamma, \lambda x . M_1, A)$</td>
<td>${(\Gamma, x : t_x, M_1, t)}$</td>
<td>${A = t_x \to t}$</td>
</tr>
</tbody>
</table>
Termination

It is enough to notice that every reduction step replaces a triple \((\Gamma, M, A)\) by a finite number of triples \((\Gamma', M', A')\) where \(M'\) is structurally smaller than \(M\).
Some notation

• In the following, we consider substitutions $S$ that act on the algebra $T_{\Sigma}(Tvar)$ where $\Sigma = \{b, \rightarrow\}$.

• We define:

\[
S \models E \quad \text{if } S \text{ unifies } E \\
S \models (\Gamma, M, A) \quad \text{if } S\Gamma \vdash M : S(A) \text{ is derivable} \\
S \models G \quad \text{if } \forall g \in G \ S \models g \\
S \models (G, E) \quad \text{if } S \models G \text{ and } S \models E
\]

• Given a term $M_0$ with free variables $x_1, \ldots, x_n$ we set the initial pair to $(G_0, \emptyset)$ with $G_0 = \{(\Gamma_0, M_0, t_0)\}$ and $\Gamma_0 = x_1 : t_{x_1}, \ldots, x_n : t_{x_n}$. 

309
Soundness and Completeness

If \((G_0, \emptyset) \rightarrow^* (G, E)\) then:

1. If \(S \models (G, E)\) then \(S \Gamma_0 \vdash M_0 : S t_0\).

2. If \(\Gamma \vdash M_0 : A\) then \(\exists S(S \models (G, E), S \Gamma_0 \subseteq \Gamma, \text{ and } A = S t_0)\).

(1) and (2) hold for the initial goal and every application of the rules maintains these properties.
Remark (on soundness)

- (1) entails the **soundness** of the method.
- Suppose from the initial goal we derive a set of equations $E$ and a substitution $S$ such that $S \models E$ (a **unifier**).
- Then we derive a **correct typing**

$$ST_0 \vdash M_0 : St_0$$
Remark (on completeness)

• On the other hand, (2) proves the completeness of the method.

• Suppose $\Gamma \vdash M_0 : A$ is a valid typing.

• Then we can reduce $(\Gamma_0, M_0, t_0)$ to $(\emptyset, E)$ and find a unifier $S$ for $E$ such that $S\Gamma_0$ is contained in $\Gamma$ and $St_0 = A$.

In particular, if we take the most general unifier $S$ of $E$ and we apply it to $t_0$ we obtain the most general type: every other type is an instance of $St_0$. 
Example

The most general type of the term

$$\lambda f. \lambda x. f(f(x))$$

is

$$(t \rightarrow t) \rightarrow (t \rightarrow t)$$

Note that the most general type is not unique. For instance,

$$(s \rightarrow s) \rightarrow (s \rightarrow s)$$

is also a most general type of the term considered.
Proof of soundness

• For instance, suppose (1) true for

\[(G \cup \{\,(\Gamma, MN, A)\,\}, E)\]

• The rule for application produces the pair \((G', E)\) with

\[G' = G \cup \{(\Gamma, M, t_1 \rightarrow A), (\Gamma, N, t_1)\}\].

• Suppose \(S \models (G', E)\). This means \(S \models (G, E)\),

\[ST \vdash M : S(t_1 \rightarrow A), \text{ and } ST \vdash N : St_1.\]

• By \((\rightarrow E)\), we conclude \(ST \vdash MN : SA\).

• Thus \(S \models (G \cup \{\,(\Gamma, MN, A)\,\}, E)\), and by hypothesis

\[ST_0 \vdash M_0 : St_0.\]
Proof of completeness

For instance, suppose

\[ \Gamma \vdash M_0 : A, \quad S \models (G \cup \{(\Gamma', \lambda x.M, A')\}, E), \]
\[ ST \Gamma_0 \subseteq \Gamma \quad A = St_0 \]

This implies:

\[ ST \Gamma' \vdash \lambda x.M : S(A') \]

which entails:

\[ ST \Gamma', x : A_1 \vdash M : A_2, \quad SA' = A_1 \rightarrow A_2, \text{ for some } A_1, A_2. \]

Suppose we reduce to the pair:

\[ (G \cup \{(\Gamma', x : t_x, M, t)\}, E \cup \{A' = t_x \rightarrow t\}) \]

Then take \( S' = S[A_1/t_x, A_2/t] \).
An equivalent ‘graphical’ presentation

- Rename bound variables so that they are all distinct and different from the free ones.

- Draw the tree associated with the \( \lambda \)-term.

- Associate a distinct type variable with every internal node.

- Associate a type variable \( t_x \) with a leaf node corresponding to the variable \( x \).

- For every abstraction node \((\lambda x. M^{t'})^t\) generate the equation \( t = t_x \rightarrow t' \).

- For every application node \((M^{t'} N^{t''})^t\) generate the equation \( t' = t'' \rightarrow t \).
Exercise

Compute, if they exist, the most general types of the following terms:

\[ \lambda x.\lambda y.\lambda z.xz(yz) \]
\[ \lambda x.\lambda y.x(yx) \]
\[ \lambda k.(k(\lambda x.\lambda h.hx)) \]
Summary

• The type inference problem is (efficiently) reduced to a syntactic unification problem.

• The existence of a most general unifier is reflected back in the existence of a most general type.

• This ‘reductionistic approach’ is typical of many program analysis techniques. E.g. the data flow analyses performed by optimising compilers are reduced to systems of monotonic boolean equations.
Recommended reading


**Note**  The approach to type inference described here has been generalised by Robin Milner et al. to a slightly more general type system with **predicative/stratified polymorphism** (ML language).
Predicative polymorphic types and type inference
Plan

• Predicative (or stratified) universal quantification in types.
• Polymorphic functions and universal quantification.
• ML-polymorphism.
• A syntax-oriented type system.
• A characterisation by let-expansion.
• A type-inference algorithm.
• Hints to the complexity of type inference.
• Generalisation to ML programs with side effects.
Second order quantification in logic

\[(P \rightarrow P) \rightarrow (P \rightarrow P)\]  \hspace{1cm} \text{(Propositional calculus)}

\[\forall x \ (P(x) \rightarrow P(S(x)))) \rightarrow (P(Z) \rightarrow \forall x \ P(x))\]  \hspace{1cm} \text{(First-order quantification)}

\[\forall P \ (P \rightarrow P) \rightarrow (P \rightarrow P)\]  \hspace{1cm} \text{(Second-order quantification)}
Second order quantification in types

• We consider the possibility of quantifying over types.

• In particular, we take a predicative approach to quantification. E.g.

\[ A \equiv (t \to t) \to (t \to t) \]  is a type

\[ \sigma \equiv \forall t \ (t \to t) \to (t \to t) \]  is a type schema

**Advantage** we stay close to simple types and can generalise the type inference techniques.

**Incovenience** we do not have the full power of second order quantification at our disposal (cf. Girard’s system F).
Syntax types and contexts, revisited

\( A ::= b \mid tid \mid (A \to A) \)  \hspace{0.5cm} \text{(types)}

\( \sigma ::= A \mid \forall tid.\sigma \)  \hspace{0.5cm} \text{(type schema)}

\( \Gamma ::= id : \sigma, \ldots, id : \sigma \)  \hspace{0.5cm} \text{(type environments)}

NB \( \forall t.(t \to t) \) is not a type and \( \forall t.t \to \forall t.t \) is not a type schema.
Typing rules, revisited (Curry style)

\[
\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad (\text{asmp})
\]

\[
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \to B} \quad (\to_I)
\]

\[
\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \quad (\to_E)
\]

\[
\frac{\Gamma \vdash M : \sigma \quad t \notin FV_t(\Gamma)}{\Gamma \vdash M : \forall t. \sigma} \quad (\forall_I)
\]

\[
\frac{\Gamma \vdash M : \forall t. \sigma}{\Gamma \vdash M : [A/t] \sigma} \quad (\forall_E)
\]

NB In \((\to_I)\) and \((\to_E)\) we handle **types** (not type schema).
Polymorphism and Universal Type Quantification

• Sometimes, the same code/function can be applied to different data-types. E.g. the functional that iterates twice a function

\[ D \equiv \lambda f.\lambda x. f(fx) \]

will work equally well on a function over booleans or integers.

• In the context of simple types, we have already seen that we can automatically infer for \( M \) the most general type:

\[ D : (t \to t) \to (t \to t) \]

• You might be under the impression that this type is good enough to represent the fact that \( D \) will work on any argument of type \((A \to A)\). Almost but not quite...
• Suppose:

\[ F_1 : (\text{bool} \to \text{bool}) \]
\[ F_2 : (\text{int} \to \text{int}) \]

• Consider the term \( P \):

\[
\text{let } f = D \text{ in } \langle fF_1, fF_2 \rangle
\]

where as usual:

\[
\text{let } x = M \text{ in } N \equiv (\lambda x.N)M
\]
\[
\langle M, N \rangle \equiv \lambda z.zMN
\]

**Exercise:** Check that the term \( P \) has no simple type. Also the example can be rephrased in the pure \( \lambda \)-calculus without appealing to the basic types \text{bool} and \text{int}. 
• A possible **way out** is to consider that $D$ has a **type schema**:

$$\sigma \equiv \forall t.(t \rightarrow t) \rightarrow (t \rightarrow t)$$

and then to **specialise** it just before it is applied to $F_1$ and $F_2$.

• This is **almost** what we can do with the predicative type system. The **problem** that remains is that we cannot really type the term

$$\lambda f.\langle fF_1, fF_2 \rangle$$

as expected because $\sigma \rightarrow \cdots$ is **not** even a type schema according to our definitions.
• One could allow **more complex types** . . . , but there is a more conservative solution which consists in taking the let-definition as a **primitive** and giving the following typing rule for it:

\[
\frac{\Gamma, x : \sigma \vdash N : A \quad \Gamma \vdash M : \sigma}{\Gamma \vdash \text{let } x = M \text{ in } N : A}
\]

This is a **first formalisation** of the **ML type system** (more to come).
Running example

- Consider $M \equiv \lambda y.\text{let } x = \lambda z. z$ in $y(xx)$ which is not typable in the simple type system but has type $A \equiv ((t \to t) \to t') \to t'$ in the ML type system.

- The main difference is that we give to the variable $x$ the type schema:
  \[ \sigma \equiv \forall s. (s \to s) \]

- Then taking $B \equiv (t \to t) \to t'$ we can derive:
  \[
  \begin{align*}
  y : B, z : s & \vdash z : s \\
  \hline
  y : B & \vdash \lambda z. z : (s \to s) \quad s \notin FV_t(B) \\
  \hline
  y : B & \vdash \lambda z. z : \sigma
  \end{align*}
  \]

- On the other hand, one can derive:
  \[
  y : B, x : \sigma \vdash (xx) : (t \to t)
  \]
Church-style type system

• The rules $(\forall_I)$ and $(\forall_E)$ are not syntax-directed. When do we apply them?

• One possibility is to ask the programmer to specify when this must be done.

• This requires an enriched syntax for terms including type abstraction and type application and leads to a Church-style formulation of the type system.

\[
(\forall_I) \quad \frac{\Gamma \vdash M : \sigma \quad t \notin FV_t(\Gamma)}{\Gamma \vdash \lambda t. M : \forall t. \sigma} \quad (\forall_E) \quad \frac{\Gamma \vdash M : \forall t. \sigma}{\Gamma \vdash MA : [A/t]\sigma}
\]
Towards a syntax-directed Curry-style system

- Due to the simplicity of the ML system, it is actually possible to foresee the points where the rules ($\forall_I$) and ($\forall_E$) need to be applied.

- This leads to a Curry-style and syntax directed type system: the shape of the term determines the rule to apply.

- First, we define $G(\Gamma, A)$ ($G$ for generalisation) as the type schema that results by quantifying the type variables which occur in $A$ but do not occur free in $\Gamma$.

- **Example:** if $\Gamma = x : \forall t. (s \to t)$ and $A = s \to (t \to r)$ then $G(A) = \forall t. \forall r. A$. 
Syntax directed ML system

The idea is to **generalise as much as possible** let-variables and **instantiate once** type schema in the context.

\[(asmp)\]
\[
\frac{x : \forall \vec{t}. A \in \Gamma}{\Gamma \vdash^{syn} \ x : [\vec{B}/\vec{t}]A}
\]

\[(\to_I)\]
\[
\frac{\Gamma, x : A \vdash^{syn} M : B}{\Gamma \vdash^{syn} \lambda x. M : A \to B}
\]

\[(\to_E)\]
\[
\frac{\Gamma \vdash^{syn} M : A \to B \quad \Gamma \vdash^{syn} N : A}{\Gamma \vdash^{syn} MN : B}
\]

\[(let)\]
\[
\frac{\Gamma, x : G(\Gamma, B) \vdash^{syn} N : A \quad \Gamma \vdash^{syn} M : B}{\Gamma \vdash^{syn} \text{let } x = M \text{ in } N : A}
\]

**NB** We use \(\vdash^{syn}\) for this system.
Exercise (running example, continued)

Consider again the term:

\[ M \equiv \lambda y. \text{let } x = \lambda z. z \text{ in } y(xx) \]

and check that we can derive:

\[ \emptyset \vdash^{\text{syn}} M : ((t \to t) \to t') \to t' \]
A type inference algorithm (1/4)

- We use the following notation:
  
  \( M, N \) type free \( \lambda \)-terms with let-definitions
  
  \( \Gamma \) type context with simple types
  
  \( \Theta \) partial function from identifiers to pairs \((\Gamma, A)\)

- The (partial) function

  \[
  PT(M, \Theta)
  \]

  tries to infer a principal typing judgment \( \Gamma \vdash M : A \) for \( M \).
  The search is \textbf{driven by} \( M \) while \( \Theta \) keeps track of the type schema assigned to let-bound variables.

- We assume: (i) all bound variables are renamed so as to be distinct and different from the free variables, (ii) in all subterms let \( x = N \) in \( M \) we have \( x \in FV(M) \).
A type inference algorithm (2/4)

- Given two typing judgments \( J_i \equiv \Gamma_i \vdash M_i : A_i \) we denote by \( \text{UnifyApl}(J_1, J_2) \) a triple \((S, t, J'_2)\) obtained as follows:
  
  1. obtain \( J'_2 \equiv \Gamma'_2 \vdash M_2 : A'_2 \) by **renaming** the type variables of \( J_2 \) so that they are **disjoint** from those in \( J_1 \).
  
  2. select a **fresh** type variable \( t \)
  
  3. build the **system of equations**:

\[
E = \{A_1 = A'_2 \rightarrow t\} \cup \{A = A' \mid x : A \in \Gamma_1, x : A' \in \Gamma'_2\}
\]

  4. compute (if it exists) a **mgu** \( S \) of \( E \).
A type inference algorithm (3/4)

\[
PT(M, \Theta) = \text{case } M
\]

\[
x: \quad \text{case } \Theta(x)
\]

\[
(\Gamma, A) \quad : \Gamma \vdash x : A
\]

\[
_{-} \quad : x : t_x \vdash x : t_x
\]

\[
\lambda x. M : \quad \text{let } (\Gamma \vdash M : A) = PT(M, \Theta) \text{ in}
\]

\[
\text{case } x : A' \in \Gamma
\]

\[
\text{true} \quad : \Gamma \backslash (x : A') \vdash \lambda x. M : A' \to A
\]

\[
_{-} \quad : \Gamma \vdash \lambda x. M : t \to A, t \text{ fresh}
\]
A type inference algorithm (4/4)

\[
M_1 M_2 : \begin{aligned}
\text{let } J_i \equiv (\Gamma_i \vdash M_i : A_i) &= PT(M_i, \Theta) \quad i = 1, 2 \text{ in} \\
\text{let } (S, t, \Gamma'_2 \vdash M_2 : A'_2) &= UnifyApl(J_1, J_2) \text{ in} \\
S(\Gamma_1 \cup \Gamma'_2 \vdash M_1 M_2 : t)
\end{aligned}
\]

\[
\begin{aligned}
\text{let } x = M_1 \text{ in } M_2 : \\
\text{let } (\Gamma_1 \vdash M_1 : A_1) &= PT(M_1, \Theta) \text{ in} \\
\text{let } \Theta' = \Theta[(\Gamma_1, A_1)/x] \text{ in} \\
\text{let } (\Gamma_2 \vdash M_2 : A_2) &= PT(M_2, \Theta') \text{ in} \\
\Gamma_2 \vdash \text{let } x = M_1 \text{ in } M_2 : A_2
\end{aligned}
\]
Exercise (running example, continued)

Consider again the term:

\[ M \equiv \lambda y. \text{let } x = \lambda z. z \text{ in } y(xx) \]

and check that \( PT(M, \emptyset) = \emptyset \vdash M : ((t \rightarrow t) \rightarrow t') \rightarrow t' \).
Reduction of ML typing to simple typing

• We may consider a typing system where to type let \( x = M \) in \( N \) we actually type \( M \) and \([N/x]M\), where by typing we mean simple typing.

• This way of proceeding is not particularly efficient because the term expansion might take exponential time.

• However, the interesting point is that the terms typable in this way are exactly those typable in the original ML system. Therefore we have the following intuitive characterisation:

\[
\text{ML typing} = \text{Simple typing} + \text{let-expansion}
\]

• The type inference algorithm we have presented is a way to keep implicit the let-expansion (but type renaming still forces a type expansion as we will see shortly!).
Simple typing with let-expansion

\[(asmp)\]
\[\begin{align*}
  x : A & \in \Gamma \\
  \Gamma & \vdash \text{let } x : A
\end{align*}\]

\[(\to I)\]
\[\begin{align*}
  \Gamma, x : A & \vdash \text{let } M : B \\
  \Gamma & \vdash \text{let } \lambda x. M : A \to B
\end{align*}\]

\[(\to E)\]
\[\begin{align*}
  \Gamma & \vdash \text{let } M : A \to B \\
  \Gamma & \vdash \text{let } N : A \\
  \Gamma & \vdash \text{let } MN : B
\end{align*}\]

\[(\text{let})\]
\[\begin{align*}
  \Gamma & \vdash \text{let } [M/x]N : A \\
  \Gamma & \vdash \text{let } M : B \\
  \Gamma & \vdash \text{let } \text{let } x = M \text{ in } N : A
\end{align*}\]

**NB** We use \(\vdash \text{let}\) to distinguish this system from \(\vdash\) and \(\vdash^{syn}\).
Exercise (running example, end)

Consider again the term:

\[ M \equiv \lambda y. \text{let } x = \lambda z. z \text{ in } y(xx) \]

and check that we can derive:

\[ \emptyset \vdash \text{let } M : ((t \rightarrow t) \rightarrow t') \rightarrow t' \]
Upper and lower bounds on the ML type inference problem

• How hard is it to decide if a term is typable in the ML system?

• Well, in theory it is hard but in practice it is easy!

• The characterisation via simple typing with let expansion shows that the problem can be solved in exponential time:
  – let-expand the term (exponential penalty).
  – reduce the simple type-inference problem to a unification problem (efficient).
  – solve the unification problem (efficient).
• Infact one can show that any decision problem that runs in exponential time can be coded as an ML type inference problem.

• Hence any algorithm (including the symbolic one) that solves the problem will run in at least exponential time.

• The good news are that the complexity is exponential in the let-depth (example next) of the term and that deeply nested chains of let-definitions do not seem to appear in practice.
Example (how to blow up ML type inference)

let pair = function x ->
  function y ->
    function z -> z x y ;;
val pair : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c = <fun>

let x1 = function y -> pair y y ;;
val x1 : 'a -> ('a -> 'a -> 'b) -> 'b = <fun>

let x2 = function y -> x1 (x1 y);;
val x2 : a -> ((('a -> 'a -> 'b) -> 'b) -> 'b) -> (('a -> 'a -> 'b) -> 'b) -> 'c) -> 'c = <fun>
...

The number of type variables and the size of the type doubles at each step. H. Mairson reports that Standard ML of New Jersey, Version 0.24, 1988 gets stuck at x5. Performance has not improved a lot since then...
Generalisation to programs with side effects

Side-effects can be added to functional languages (cf. ML) but their non-logical nature can easily produce paradoxical situations.

Example 1  One can simply type a looping computation (so typing does not guarantee termination):

let loop = ref(function x -> x ) in
loop:= (function x -> !loop ()) ; !loop ();;

NB The example can be generalised to ‘implement’ a fix-point combinator.
Example 2 Polymorphic generalisation of a reference expression may lead to type errors:

```ml
let x = ref (function x -> x) in
x := (function x -> x+1); (!x) true;;
```

ML-like languages avoid these problems by allowing polymorphic generalisation only on values. For instance, OCAML accepts

```ml
let x = (function x -> x) in
x (); x true ;;
```

but rejects the dangerous expression above with references as well as the following innocuous one:

```ml
let x = (function y -> (function x -> x))2 in
x (); x true ;;
```

NB In practice, most ML programs seem to meet this restriction.
Summary

- **Universally quantified** types are the types of **polymorphic** terms.

- In particular we have considered a **predicative/stratified** form of universal quantification (as used in ML).

- The type inference techniques can be **extended** to predicative polymorphism.

- Complexity of type inference is **exponential** in the number of nested let-definitions. In **practice**, it works well.

- To handle side effects, give **polymorphic types** only to **values**.
Recommended reading

Conclusion
Two models of functional computation

First-order Term Rewriting Systems.

Higher-order (Typed) $\lambda$-calculi.
Recurring themes

**Confluence** local confluence, critical pairs, completion, parallel reduction.

**Termination** interpretation method, recursive path-ordering, reducibility candidates.

**Expressive power** PTIME, primitive recursive, partial recursive functions.

**Constraint solving** unification, type inference.
Approach

- Reduce preliminaries.
- Give rather complete proofs.
- Only hint to the applications.

Some of the courses of the second semester will go deeper into the theory while others will focus more on the applications to programming languages design.
Exercise (revision)

Attach a name and a concept and/or a theorem to the following pictures: