

Foundations of Rewriting and Functional Programming

Cours Programmation 1.2, Licence d'Informatique
ENS Cachan, AA 2011-2012

Roberto M. Amadio

Université Paris Diderot (Paris 7)

1

Plan (preliminary)

- Rewriting Systems.
- First-order functional programs.
- Terms: rewriting and unification.
- Termination.
- Confluence and Completion.
- The λ -calculus (higher-order rewriting).
- Simply-typed λ -calculus.
- Type inference.
- Polymorphic λ -calculus (system F).
- Optional topics (usually not enough time):
 - Weak reduction and Environment machines.
 - Predicative polymorphic λ -calculus (ML polymorphism).
 - Type inference revisited.
 - Categorical Interpretations.

3

The setting

An introduction to the **foundations of rewriting systems**, **functional programming**, and **type theory** oriented towards **syntactic/proof-theoretic** methods.

2

Strongly related courses

Semester 1

- Programming 1.1.
- Computability and Logic.

Semester 2

- Programming 2.
- λ -calculus and Computer Science Logic.
- Logic programming.

4

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:

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

- Les transparents sont en anglais mais les sujets seront en français et les réponses pourront être rédigées 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 sont notés avec le projet de compilation.

5

Plan

- A few basic definitions: termination, confluence.
- A few basic facts: Induction principle, Newman's lemma.

7

Reduction systems

6

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,...

8

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

$$S(x) + y = S(x + y) \quad \text{or} \quad (x + y) + z = x + (y + z) ?$$

9

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?

From equations to rewriting rules

Suppose we come out with the following orientation:

$$x + Z \rightarrow x$$

$$Z + x \rightarrow x$$

$$S(x) + y \rightarrow S(x + y)$$

$$x + S(y) \rightarrow S(x + y)$$

$$(x + y) + z \rightarrow x + (y + z)$$

This is a (term) rewriting system. It is a special case of rewriting system where the relation \rightarrow is represented **schematically** by a finite set of rules.

10

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 simple a 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.

13

'Demo' for our example (confluence)

Tool used: <http://colo6-c703.uibk.ac.at/mkbt2/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.

14

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.

YES

Time: 0.040384

POLY:

[*](x0, x1) = 6x0 + x1 + 1,

[i](x0) = 8x0,

[e] = 0

POLY=Polynomial interpretation, another technique for proving termination.

15

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.

16

Terminology

- Fix a **reduction relation** (A, \rightarrow) .
- \rightarrow is **terminating** if there is no infinite sequence:

$$a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \dots$$

- 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:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow \dots \text{ and } i \rightarrow 0 \text{ for } i \geq 1$$

17

Exercise (confluent vs. Church-Rosser)

Show that a reduction relation \rightarrow is **Church-Rosser** iff it is **confluent**.

19

- \rightarrow is **confluent** if

$$\frac{\forall b, c (b \xrightarrow{*} a \xrightarrow{*} c)}{\exists d (b \xrightarrow{*} d \xrightarrow{*} c)}$$

We also write $b \downarrow c$ if $\exists d (b \xrightarrow{*} d \xrightarrow{*} c)$.

- 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 λ -calculus: two λ -terms are ‘equivalent’ iff they can be rewritten to a common λ -term.

18

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**.

20

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.

21

The principle of well-founded induction

The following reasoning principle holds in a well-founded partial order.

$$\frac{\forall x \in W (\forall y < x P(y)) \rightarrow P(x)}{\forall x \in W P(x)}$$

Question Explain why the principle fails if $W = \{*\}$ is a singleton and $>$ is reflexive.

23

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 admits no infinite chain:

$$x_0 > x_1 > x_2 > \dots$$

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** $(W, \overset{+}{\rightarrow})$ where $\overset{+}{\rightarrow}$ is the transitive closure of \rightarrow .

22

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 on to build:

$$x_0 > x_1 > x_2 > \dots$$

which **contradicts** the hypothesis that W is well-founded.

24

Remark (special case)

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

$$\frac{\forall n (\forall n' < n P(n')) \rightarrow P(n)}{\forall n P(n)}$$

or equivalently:

$$\frac{P(0) \wedge (\forall n (\forall n' < n + 1 P(n')) \rightarrow P(n + 1))}{\forall n P(n)}$$

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

$$\frac{P(0) \wedge (\forall n (P(n) \rightarrow P(n + 1)))}{\forall n P(n)}$$

25

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 **induction principle** holds on a partial order $(W, >)$ then $(W, >)$ is **well-founded**.

27

Exercise (on well-founded orders)

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

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

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

Prove or disprove the assertion that $>$ is a well-founded order.

26

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

$$\frac{\forall b, c (b \leftarrow a \rightarrow c)}{\exists d (b \xrightarrow{*} d \xleftarrow{*} c)}$$

Theorem (Newman) If a reduction system (A, \rightarrow) is **locally confluent** and **terminating** then it is confluent.

28

Proof of Newman's lemma

- We apply the principle of **well-founded induction** to $(A, \overset{\pm}{\rightarrow})$!
- Suppose

$$c_1 \overset{*}{\leftarrow} b_1 \leftarrow a \rightarrow b_2 \overset{*}{\rightarrow} c_2$$

29

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 $\exists d (b \overset{*}{\rightarrow}_2 d \text{ and } c \overset{*}{\rightarrow}_1 d)$.
- Show that:
 - if \rightarrow_1 and \rightarrow_2 are **confluent** and **commute**
 - then $\rightarrow_1 \cup \rightarrow_2$ is also **confluent**.

31

- By **local confluence**, $\exists d (b_1 \overset{*}{\rightarrow} d \overset{*}{\leftarrow} b_2)$.
- By **induction hypothesis** on b_1 , $\exists d' (c_1 \overset{*}{\rightarrow} d' \overset{*}{\leftarrow} d)$.
- By **induction hypothesis** on b_2 , $\exists d'' (d' \overset{*}{\rightarrow} d'' \overset{*}{\leftarrow} c_2)$.
- But then $c_1 \downarrow c_2$.
- Thus by the **principle of well-founded induction**, the reduction relation is **confluent**.

30

A fundamental result in combinatorics

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 $f : \mathbf{N}^{[k]} \rightarrow \{1, \dots, r\}$ then there is an **infinite subset** X of \mathbf{N} such that f is constant over $X^{[k]}$.

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

32

Example

- An **undirected countable graph** can be described by a function $f : \mathbf{N}^{[2]} \rightarrow \{1, 2\}$ 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**.

33

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 $f : \mathbf{N}^{[2]} \rightarrow \{1, \dots, r\}$. Show that there is an infinite sequence of infinite sets $X_0 \supset X_1 \supset X_2 \supset \dots$ such that:

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

2. For $x \in X_{i+1}$, let $c_i = f(\{\min(X_i), x\}) \in \{1, \dots, r\}$. Show that for some $c \in \{1, \dots, r\}$ 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 $\forall i < j \quad f(x_{\sigma(i)}, x_{\sigma(j)}) = c$.

35

Finite version of Ramsey theorem

Let $[n] = \{1, \dots, n\}$. 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.

34

Exercise (modular termination)

Let \rightarrow_1 and \rightarrow_2 be two reduction relations such that $\rightarrow_1 \cup \rightarrow_2$ is **transitive**. Show that if \rightarrow_1 and \rightarrow_2 are terminating then their union is terminating too.

36

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_Σ be the set of **closed terms** over Σ (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_Σ such that:

$$\begin{aligned} f(g_1(t)) &\rightarrow g_1(g_1(f(f(t)))) && \text{for all } t \in T_\Sigma \\ f(g_2(t)) &\rightarrow g_2(f(t)) && \text{for all } t \in T_\Sigma \\ f(a) &\rightarrow a \end{aligned}$$

and such that if $t \rightarrow s$ and h is a unary symbol in Σ then $h(t) \rightarrow h(s)$.

37

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.
- Ramsey theorem:

In an infinite graph there is a an infinite subgraph where all elements are connected or all elements are disconnected.

39

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 $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \dots$ 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.

38

- Some sufficient conditions for modular confluence and termination:

$\rightarrow_1, \rightarrow_2$	condition	$\rightarrow_1 \cup \rightarrow_2$
Confluent	$\xrightarrow{*}_1, \xrightarrow{*}_2$ commute	Confluent
Terminate	$(\rightarrow_1 \cup \rightarrow_2)$ transitive	Terminates

40

Recommended reading

Baader, Nipkow, *Term rewriting and all that*, Cambridge University Press, chapters 1 and 2.

41

Plan

- We introduce a simple first-order **functional language** (a fragment of ML).
- First we present a definition mechanism that guarantees termination: **primitive recursion** (cf. Logic course).
- Then we see under which conditions, we can guarantee **termination in polynomial time**.

43

Functional programming, primitive recursion, and polynomial time

42

A first-order functional language

- To define **types**, a system of mutually recursive equations:

$$t = \dots \mid c \text{ of } t_1, \dots, t_n \mid \dots$$

- The symbol c is a **constructor** of the data type t .
- Elements of type t can be defined **inductively**: constants, constructors applied to constants, ...

44

Examples of types

- Booleans.

$$bool = t \mid f$$

- Tally numbers.

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

- Lists of tally numbers.

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

- Binary numbers (words).

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

45

Functions

A list of mutually recursive definitions by [pattern matching](#).

$$\begin{aligned} f &= \\ \dots & \\ p_1, \dots, p_n &\Rightarrow e \\ \dots & \end{aligned}$$

- 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 $f(p_1, \dots, p_n) \Rightarrow e$ as a **rule**.

47

Syntax

Reserve symbols as follows:

c, c', \dots constructor symbols

f, f', \dots function symbols

$x, x' \dots$ first-order variables

and distinguish the following syntactic categories:

$v ::= c(v, \dots, v)$ (values)

$p ::= x \mid c(p, \dots, p)$ (patterns)

$e ::= x \mid c(e, \dots, e) \mid f(e, \dots, e)$ (expressions).

46

Examples of functions

$$\begin{aligned} ite &= \\ t, y, _ &\Rightarrow y \\ f, _, z &\Rightarrow z \end{aligned}$$

$$\begin{aligned} lesseq &= \\ z, _ &\Rightarrow t \\ s(x), z &\Rightarrow f \\ s(x), s(y) &\Rightarrow lesseq(x, y) \end{aligned}$$

48

The same in ML

```

insert =
x, nil      ⇒ cons(x, nil)
x, cons(y, l) ⇒ ite(lesseq(x, y), cons(x, cons(y, l)),
                    cons(y, insert(x, l)))

```

49

Exercise (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.

51

```

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)));;

```

50

Typing

Programs are supposed to be **well typed**.

- A **constructor** c has type $t_1, \dots, t_n \rightarrow t$ if it is declared (once) as $t = \dots \mid c \text{ of } t_1, \dots, t_n \mid \dots$
- To every **function** f we must be able to assign a type $t_1, \dots, t_n \rightarrow t$ so that all expressions in the program are well typed with respect to a suitable context $\Gamma \equiv x_1 : t_1, \dots, x_n : t_n$, where:

$$\frac{x : t \in \Gamma \quad \Gamma \vdash x : t}{\Gamma \vdash x : t} \quad \frac{\Gamma \vdash e_i : t_i \quad i = 1, \dots, n \quad c : t_1, \dots, t_n \rightarrow t}{\Gamma \vdash c(e_1, \dots, e_n) : t}$$

$$\frac{\Gamma \vdash e_i : t_i \quad i = 1, \dots, n \quad f : t_1, \dots, t_n \rightarrow t}{\Gamma \vdash f(e_1, \dots, e_n) : t}$$

52

A rule $f(p_1, \dots, p_n) \Rightarrow e$ in a function definition f is **well typed** if:

1. $f : t_1, \dots, t_n \rightarrow t$.
2. There are contexts $\Gamma_i, i = 1, \dots, n$ such that $\Gamma_i \vdash p_i : t_i$.
3. $\Gamma = \Gamma_1, \dots, \Gamma_n$ 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.

53

Call-by-value evaluation

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

$$\frac{e_j \Downarrow v_j \quad j = 1, \dots, n}{c(e_1, \dots, e_n) \Downarrow c(v_1 \dots, v_n)}$$

$$\frac{e_j \Downarrow v_j, f(p_1, \dots, p_n) \Rightarrow e, Sp_j = v_j, j = 1, \dots, n \quad S(e) \Downarrow v}{f(e_1, \dots, e_n) \Downarrow v}$$

55

Exercise (typing)

Type the functions for insertion sort and quick sort.

54

Exercise (evaluation)

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

`cons(s(s(z)), cons(z, cons(s(z), nil)))`

Note that *ite* function always evaluates **both branches**.

56

What we have done so far

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

The language is easily shown to be **Turing complete**. Exercise!

57

Primitive Recursion From $g : \mathit{tnat}^n \rightarrow \mathit{tnat}$, $h : \mathit{tnat}^{n+2} \rightarrow \mathit{tnat}$ define $f : \mathit{tnat}^{n+1} \rightarrow \mathit{tnat}$ such that:

$$\begin{aligned}
 f &= \\
 z, \vec{y} &\Rightarrow g(\vec{y}) \\
 s(x), \vec{y} &\Rightarrow h(f(x, y), x, \vec{y})
 \end{aligned}$$

59

Primitive recursion

We start looking at a fragment with **guaranteed termination**.

Tally numbers

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

Basic functions

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

Composition

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

58

Examples

We take some freedom with the notation. E.g. we write the constant z rather than the zero function z .

$$\begin{aligned}
 add &= && \text{(addition)} \\
 z, y &\Rightarrow y \\
 s(x), y &\Rightarrow s(add(x, y)) \\
 \\
 mul &= && \text{(multiplication)} \\
 z, y &\Rightarrow z \\
 s(x), y &\Rightarrow add(mul(x, y), y) \\
 \\
 exp &= && \text{(exponentiation)} \\
 x, z &\Rightarrow s(z) \\
 x, s(n) &\Rightarrow mul(exp(x, n), x)
 \end{aligned}$$

Can go on to describe towers of exponentials, ... Complexity of programmable functions still very high!

60

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.

61

Recursion on notation

Basic functions

$e(x) = \epsilon$	empty word
$s_i(x) = ix, i = 0, 1$	successors
$p_i(x_1, \dots, x_n) = x_i, i = 1, \dots, n$	projections

Composition As for primitive recursion

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

63

From unary to binary notation

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

$$\text{type } \text{bnat} = 0 \text{ of } \text{bnat} \mid 1 \text{ of } \text{bnat} \mid \epsilon$$

- Then we try to **bound the complexity** of the definable functions.

62

Recursion on notation From $g : \text{bnat}^n \rightarrow \text{bnat}$,

$h_i : \text{bnat}^{n+2} \rightarrow \text{bnat}, i = 1, 2$ define $f : \text{bnat}^{n+1} \rightarrow \text{bnat}$ such that:

$$\begin{aligned}
 f &= \\
 \epsilon, \vec{y} &\Rightarrow g(\vec{y}) \\
 0(x), \vec{y} &\Rightarrow h_0(f(x, \vec{y}), x, \vec{y}) \\
 1(x), \vec{y} &\Rightarrow h_1(f(x, \vec{y}), x, \vec{y})
 \end{aligned}$$

64

Exercise (programming by recursion on notation)

- Represent binary numbers as binary words where the **least significant 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 ϵ 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.

65

Computing primitive recursion

- Suppose $v_k = i_k \dots i_1 \epsilon$.
- Here is a simple for loop that computes $f(v_k, \vec{y})$:

```
r := g( $\vec{y}$ );  
for j = 1 to k do  
  r :=  $h_{i_j}(r, v_{j-1}, \vec{y})$  od  
return r
```

Problem Suppose that h_i and g can be computed in **polynomial time**. Can we conclude that f can be computed in **polynomial time**?

67

- 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.

66

What can go wrong

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

$$\begin{aligned}d &= \\ \epsilon &\Rightarrow 1\epsilon \\ d(ix) &\Rightarrow i(i(d(x)))\end{aligned}$$

Then consider the function e

$$\begin{aligned}e &= \\ \epsilon &\Rightarrow 1\epsilon \\ i(x) &\Rightarrow d(e(x))\end{aligned}$$

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 $|x|$.

68

Bounded recursion on notation (BRN)

In the definition of recursion on notation

$$\begin{aligned} f(x, \vec{y}) = \\ \epsilon, \vec{y} &\Rightarrow g(\vec{y}) \\ 0(x), \vec{y} &\Rightarrow h_0(f(x, \vec{y}), x, \vec{y}) \\ 1(x), \vec{y} &\Rightarrow h_1(f(x, \vec{y}), x, \vec{y}) \end{aligned}$$

we require that a **polynomial** S_f with non-negative coefficients is given such that:

$$|f(x, \vec{y})| \leq S_f(|x|, |\vec{y}|) .$$

69

Remark

- It is quite possible to program a function that takes **exponential time** and runs in **polynomial space** (never going twice through the same configuration!).
- For instance, take a function that **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 **cannot** be programmed.
- The problem is **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.

71

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**.

70

All 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, \dots, v_n :

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

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

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

By inductive hypothesis

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

Let S be a polynomial that bounds all S_{g_i} .

72

All algorithms in BRN can be computed in PTIME (2/4)

Applying again the inductive hypothesis:

$$\begin{aligned} |f(g_1(\vec{x}), \dots, g_k(\vec{x}))| &\leq S_f(|g_1(\vec{x})|, \dots, |g_k(\vec{x})|) \\ &\leq S_f(S_{g_1}(|\vec{x}|), \dots, S_{g_k}(|\vec{x}|)) \\ &\leq S_f(S(|\vec{x}|), \dots, S(|\vec{x}|)) \end{aligned}$$

and the composition of polynomials is a polynomial.

NB Thus data computed by BRN has **size polynomial in the size of the input**.

73

All 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 $f(u_1, \dots, u_n)$ can be computed in time $T_f(|u_1|, \dots, |u_n|)$.

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

```

r := g(y);
for j = 1 to k do
  r := hij(r, vj, y) od
return r

```

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

74

All 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 $|x|, |\vec{y}|$.

75

All TM running in PTIME can be simulated (1/7)

- Let $M = (\Sigma, Q, q_o, F, \delta)$ be a **Turing machine** with
 - Σ alphabet,
 - Q states, $q_o \in Q$ initial state,
 - $F \subseteq Q$ final states,
 - $\delta : \Sigma \times Q \rightarrow \Sigma \times Q \times \{L, R\}$ transition function.
- Obviously, elements in Σ and Q can be encoded as **binary words** of length $\lceil \lg(\#\Sigma) \rceil$ and $\lceil \lg(\#Q) \rceil$, respectively.

76

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**.

77

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.

79

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.

$$step(q, h, l, r) = \dots$$
$$q = 01\epsilon, h = 1\epsilon, l = 0l', r = r \Rightarrow (11\epsilon, 0\epsilon, l', 0r)$$

describes the situation where being in state 01 and reading 1, we go in state 11, write 0, and move to the left.

78

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_0 at least $P(|x_0|)$ 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.

80

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:

$$\begin{aligned} it(\epsilon, x) &= x \\ it(i \cdot t, x) &= step(it(t, x)) \end{aligned}$$

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

$$|it(t, x)| \leq S_{it}(|t|, |x|) .$$

81

Remarks

- **Completeness results** are a kind of **sanitary check** (cf. Turing completeness, logic completeness, ...).
- In practice, there might be a big difference: assembler vs. ML, Hilbert system vs. Natural deduction, ... So there is still space for research but with a **different yard stick**.
- Cobham's result restricts the programmer to primitive recursion and it provides no clue on **how to find a polynomial bound** on size.
- Following work (Bellantoni-Cook 1992) introduces a simple **type discipline** that distinguishes two kinds of arguments of a function and shows that well-typed functions have a polynomial bound.

83

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:

$$\begin{aligned} exec &= \\ \epsilon, q, h, l, r &\Rightarrow (q, h, l, r) \\ ix, q, h, l, r &\Rightarrow step(exec(x, q, h, l, r)) \end{aligned}$$

- 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 p 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!

82

Summary

- A programming language with **general recursive first-order functions**.
- A function defined by **primitive recursive** or **recursion on notation** is guaranteed to terminate.
- A function defined by **bounded recursion on notation** is guaranteed to terminate in **polynomial time**.
- Key insight:
 $PTIME = \text{polynomial data size} + \text{restricted recursion mechanism}$

84

References (optional)

If you want to know more.

- Rose. *Subrecursion*, Clarendon Press (chapter 1 and section 5.4 in particular).
- Clote, *Computation models and function algebras*, In Proc. Logic and computational complexity, LNCS 960, 1995.
Warning: this is an advanced research survey.

85

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**.

87

Terms: rewriting rules and syntactic unification

86

Some motivation

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

$$\begin{aligned} \min &= \\ z, y &\Rightarrow z \\ s(x), z &\Rightarrow z \\ s(x), s(y) &\Rightarrow s(\min(x, y)) \end{aligned}$$

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

88

Term rewriting systems

We introduce term rewriting systems.

- The **signature** $\Sigma = \{f_1, \dots, f_n\}$ is a finite set of symbols f_i of given arity $ar(f_i) \in \mathbf{N}$.
- 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, \dots (respecting the arity).
- A **substitution** $S : V \rightarrow T_\Sigma(V)$ is a function which is the identity almost everywhere.

- A variety of methods have been developed to **prove termination**.
- We will introduce some of them in a more general framework: **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 the functional language** we have considered.

89

90

- The **substitution extension** to $T_\Sigma(V)$ is:

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

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 $Var(r) \subseteq Var(l)$.

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

$$t \rightarrow_R s \quad \text{if } t = C[Sl] \text{ and } s = C[Sr]$$

91

Example (rewrite relation induced by a TRS)

Take as TRS R :

$$f(x) \rightarrow g(f(s(x)))$$

$$i(0, y, z) \rightarrow y$$

$$i(1, y, z) \rightarrow z$$

Then, for instance:

$$f(s(y)) \rightarrow_R g(f(s(s(y)))) \rightarrow_R g(g(f(s(s(s(y)))))) \rightarrow_R \dots$$

$$i(0, 1, f(y)) \rightarrow_R 1$$

$$i(0, 1, f(0)) \rightarrow_R i(0, 1, g(f(s(0)))) \rightarrow_R \dots$$

92

All 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):

$$f(p_1, \dots, p_n) \rightarrow e$$

Then it is easy to check that $e \Downarrow v$ implies $e \xrightarrow{*} v$.

93

Exercise (initial algebra)

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

- A Σ -algebra is a vector $\mathcal{A} = (A, c_A, g_A)$ where A is a set, $c_A \in A$ and $g_A : A \times A \rightarrow A$.
- If $\mathcal{A} = (A, c_A, g_A)$ and $\mathcal{B} = (B, c_B, g_B)$ are two Σ -algebras, a *morphism* h from \mathcal{A} to \mathcal{B} is a function $h : A \rightarrow B$ 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 Σ that may contain variables in X .
- We note that $\mathcal{T}_\Sigma(X) = (T_\Sigma(X), c_T, g_T)$ where $c_T = c$ and $g_T(t, t') = g(t, t')$ is a Σ -algebra.

95

Exercise

Write down the TRS corresponding to the functional program for the insertion sort.

94

1. Show that if $\mathcal{A} = (A, c_A, g_A)$ is a Σ -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 Σ -algebra is uniquely determined by an assignment of algebra values to its variables.

96

Some notation and terminology

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).

97

- We write $t = s$ if the terms t and s are **syntactically equal**.
- We denote with $Var(t)$ 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 **finite list**

$$[t_1/x_1, \dots, t_n/x_n]$$

where $x_i \neq x_j$ if $i \neq j$.

- We also denote with id the **identity substitution**.

98

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

$$R \leq S \text{ iff } \exists T \ T \circ R = S$$

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

$$\{t_1 = s_1, \dots, t_n = s_n\} .$$

- 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).

99

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:

$$S \neq T, \quad S \leq T \text{ and } T \leq S .$$

100

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) $(E \cup \{x = x\}, S) \rightarrow (E, S)$
- (vt₁) $(E \cup \{x = t\}, S) \rightarrow ([t/x]E, [t/x] \circ S) \quad x \notin \text{Var}(t)$
- (vt₂) $(E \cup \{x = t\}, S) \rightarrow \perp \quad x \neq t, x \in \text{Var}(t)$
- (f₁) $(E \cup \{f(t_1, \dots, t_n) = f(s_1, \dots, s_n)\}, S) \rightarrow (E \cup \{t_1 = s_1, \dots, t_n = s_n\}, S)$
- (f₂) $(E \cup \{f(t_1, \dots, t_n) = g(s_1, \dots, s_m)\}, S) \rightarrow \perp \quad \text{if } f \neq g$

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).

101

Exercise (unification algorithm)

Apply the unification algorithm to the following systems of equations:

$$\{f(x, f(x, y)) = f(g(y), f(g(a), z))\}, \quad (a \text{ is a constant}).$$

$$\{f(x, f(y)) = f(y, f(f(x)))\}$$

103

Example (unification)

Applying the unification method to the system

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

leads to the substitution

$$S = [f(z)/y] \circ [f(z)/x]$$

102

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 $S \leq T$.

104

(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 $\mu(E, S) > \mu(U)$.

105

(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 $x \notin \text{Var}(t)$, $S \circ [t/x](x) = S(t) = S \circ [t/x](t)$.

107

Preliminary remark

- In (E, S) the second component of S is just used to **accumulate** the substitutions.
- Therefore:

$$(E, S) \rightarrow^m (\emptyset, S_n \circ \dots \circ S_1 \circ S) \text{ iff } (E, \emptyset) \rightarrow^m (\emptyset, S_n \circ \dots \circ S_1)$$

where $m \geq 1$, $n \geq 0$ and the S_i are the elementary substitutions $[t/x]$ introduced by rule (vt_1) .

106

(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.

108

- 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:

$$S \circ [t/x] \leq T \circ [t/x] = T .$$

109

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.

111

Exercise (size of solution)

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

Consider the following unification problem:

$$\{x_1 = f(x_0, x_0), x_2 = f(x_1, x_1), \dots, x_n = f(x_{n-1}, x_{n-1})\}$$

Compute the most general unifier S .

What is the size of $S(x_n)$ as a function of n ?

110

Summary

- Notion of **term rewriting system** generalizes that of first-order functional program.
- 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**.

112

Exercise (on filters)

Let t, s, \dots be terms over a signature Σ .

We say that t is a **filter** (or pattern) for s if there is a substitution S such that $St = s$. 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 $r \geq s$.

113

Termination of Term Rewriting Systems

115

Recommended reading

Baader, Nipkow, *Term rewriting and all that*, Cambridge University Press, chapter 3 and sections 4.5-4.6.

114

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 through the notion of **well-partial order** and **Kruskal's theorem**.

116

Reduction orders

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

$$\frac{t > s}{C[t] > C[s] \text{ and } St > Ss}$$

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 $l > r$.

117

Interpretation method

Well-founded Σ -algebras and strictly monotonic functions

Suppose the TRS is given over a **signature** Σ .

- Consider a Σ -**algebra** (cf. exercise on Σ -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**:

$$a_i > b_i \Rightarrow f^A(\dots, a_i, \dots) > f^A(\dots, b_i, \dots)$$

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

$$\begin{aligned} \llbracket x \rrbracket \theta &= \theta(x) \\ \llbracket f(t_1, \dots, t_n) \rrbracket \theta &= f^A(\llbracket t_1 \rrbracket \theta, \dots, \llbracket t_n \rrbracket \theta) \end{aligned}$$

This is the usual interpretation of terms in first-order logic.

119

118

Proposition If all functions f^A are **monotone**, then the following is a **reduction order** on $T_\Sigma(V)$:

$$t >_A s \text{ if } \forall \theta \llbracket t \rrbracket \theta >_A \llbracket s \rrbracket \theta$$

120

Proof(1/4)

Well-foundation Suppose by contradiction

$$t_0 >_A t_1 >_A \dots$$

Then taking an arbitrary assignment θ we have:

$$\llbracket t_0 \rrbracket \theta >_A \llbracket t_1 \rrbracket \theta >_A \dots$$

But this contradicts the hypothesis that $(A, >)$ is well-founded.

121

Proof (3/4)

- We note that:

$$\llbracket [s/x]t \rrbracket \theta = \llbracket t \rrbracket \theta \llbracket [s] \theta / x \rrbracket$$

- Thus taking $\theta' = \theta \llbracket [s] \theta / x \rrbracket$ we have:

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

123

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 $[s_1/x_1, \dots, s_n/x_n]$).
- That is for any θ ,

$$\llbracket [s/x]t \rrbracket \theta >_A \llbracket [s/x]t' \rrbracket \theta$$

122

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 $C = f(\dots, C', \dots)$.
- 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.

124

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.

125

Exercise

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

127

Example: polynomial interpretations

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

1. Coefficients range over the natural numbers: thus **no negative coefficients**, and the polynomials are **monotonic**.
2. $p_f(a, \dots, a) \in A$: 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**.

126

Example

Another definition of addition and multiplication over tally numbers (this time in TRS notation)

$$a(z, y) \rightarrow y \quad a(x, z) \rightarrow x \quad a(s(x), s(y)) \rightarrow s(a(x, y))$$

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

A polynomial interpretation is:

$$p_z = 1 \quad p_s = x + 2 \quad p_a = 2x + y + 1 \quad p_m = (x + 1)(y + 1)$$

128

Exercise (polynomial interpretation)

1. Find a polynomial interpretation for the TRS:

$$f(f(x, y), z) \rightarrow f(x, f(y, z)) \quad f(x, f(y, z)) \rightarrow f(y, y)$$

129

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 10th problem (proved by Matiyasevich).

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

$$\exists n_1, \dots, n_k \in \mathbf{Z} \ P(n_1, \dots, n_k) = 0$$

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.

131

2. Find a polynomial interpretation for the TRS

$$\begin{array}{lll} x + 0 \rightarrow x & x + \mathbf{s}(y) \rightarrow \mathbf{s}(x + y) & \text{(addition)} \\ d(0) \rightarrow 0 & d(\mathbf{s}(x)) \rightarrow \mathbf{s}(\mathbf{s}(d(x))) & \text{(double)} \\ q(0) \rightarrow 0 & q(\mathbf{s}(x)) \rightarrow q(x) + \mathbf{s}(d(x)) & \text{(square)} \end{array}$$

130

Simplification orders

132

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:

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

133

Exercise (subterm property)

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

$$C[t] > t$$

134

Homeomorphic embedding

Let \rightarrow be the TRS induced by the rules

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

Definition We write $t \succeq s$, read t embeds (**prolonge**) s , if $t \xrightarrow{*} s$, *i.e.*, if we can rewrite t in s in a finite number of steps (possibly 0).

135

Example (homeomorphic embedding)

$$f(f(h(a), h(x)), f(h(x), a)) \succeq f(f(a, x), x)$$

136

Exercise (alternative definition of homeomorphic embedding)

Here is another definition of the homeomorphic embedding:

$$\frac{}{x \succeq x}$$

$$\frac{s_i \succeq t_i, i = 1, \dots, n}{f(s_1, \dots, s_n) \succeq f(t_1, \dots, t_n)}$$

$$\frac{s_i \succeq t \text{ for some } i}{f(s_1, \dots, s_n) \succeq t}$$

Show that this definition is equivalent to the previous one.

137

Recursive path order (RPO)

139

Exercise (homeomorphic embedding vs. simplification order)

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

NB The reflexive closure of a simplification order may contain **strictly** the homeomorphic embedding.

138

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).

140

- We fix a pre-order \geq_Σ on the function symbols in Σ , where $f \approx_\Sigma g$ implies f, g have the **same** arity.
- To check $t = f(t_1, \dots, t_m) > g(s_1, \dots, s_n) = s$, check first whether s is a subterm of t . Otherwise, compare the top symbols f, g :
 - If $f >_\Sigma g$ then check recursively $t > s_i$ for $i = 1, \dots, n$.
 - If $f \approx g$ then fix a way to compare (t_1, \dots, t_m) with (s_1, \dots, s_m) .
 - If none of the above works, give up.

141

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.

143

Ordering tuples

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

$$\frac{s_1 \geq t_1, \dots, s_n \geq t_n, \exists j \ s_j > t_j}{(s_1, \dots, s_n) > (t_1, \dots, t_n)} \quad \text{product}$$

$$\frac{s_1 = t_1, \dots, s_{i-1} = t_{i-1}, s_i > t_i}{(s_1, \dots, s_n) > (t_1, \dots, t_n)} \quad \text{lexicographic}$$

142

(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 $M : A \rightarrow \mathbf{N}$.
- 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.

144

- Let $\mathcal{M}_{fin}(X)$ denote the **finite multi-sets** over a set X .
- Assume $(X, >)$ is an order and $M, N \in \mathcal{M}_{fin}(X)$. 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 $X = \mathbf{N}$ 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 $>_{1,m}$.

145

Formalisation of König lemma

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

147

König lemma

Every finitely branching tree with an infinite number of nodes admits an infinite path.

146

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 π 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 πi_{k+1} is infinite.
- Proceeding in this way, we can build an infinite path in D .

148

Exercise (multi-set)

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

149

Recursive path order

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

$$\frac{s \geq_r t}{f(\dots s \dots) >_r t}$$

$$\frac{f >_{\Sigma} g \quad f(s_1, \dots, s_m) >_r t_i \quad i = 1, \dots, n}{f(s_1, \dots, s_m) >_r g(t_1, \dots, t_n)}$$

$$\frac{f \approx g \quad \tau(f) = \tau(g) \quad (s_1, \dots, s_m) >_r^{\tau(f)} (t_1, \dots, t_m) \quad f(s_1, \dots, s_m) >_r t_i \quad i = 1, \dots, m}{f(s_1, \dots, s_m) >_r g(t_1, \dots, t_m)}$$

NB For the lexicographic order, the condition $f(s_1, \dots, s_m) >_r t_i$ is needed.

151

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.

$$\frac{\{\{s_1, \dots, s_n\} > \{t_1, \dots, t_n\}\}}{(s_1, \dots, s_n) > (t_1, \dots, t_n)} \quad \text{multiset}$$

150

RPO is a simplification order

Theorem The **recursive path order** is a **simplification order** on $T_{\Sigma}(V)$.

NB 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 λ -calculus).

152

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:

- $>$ is strict.
- $s > t$ implies $Var(s) \supseteq Var(t)$.
- Transitivity.
- Subterm property.
- Closure under substitution.
- Closure under context.

153

$>$ is strict

- By induction on s show that $s > s$ is impossible.
- Note in particular that $x > t$ and $f > f$ are impossible.

155

Specialised definition for product

It is convenient to have the specialised definition for product:

$$(R_1) \quad \frac{s \geq_r t}{f(\dots s \dots) >_r t}$$

$$(R_2) \quad \frac{f >_{\Sigma} g \quad f(s_1, \dots, s_m) >_r t_i \quad i = 1, \dots, n}{f(s_1, \dots, s_m) >_r g(t_1, \dots, t_n)}$$

$$(R_3) \quad \frac{f \approx g \quad s_i \geq t_i \text{ for } i \in 1..m, \text{ and } \exists j \in 1..m \quad s_j > t_j}{f(s_1, \dots, s_m) >_r g(t_1, \dots, t_m)}$$

154

$s > t$ implies $Var(s) \supseteq Var(t)$

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

156

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).

157

Subterm property

We check that $f(x_1, \dots, x_n) > x_i$ for $i = 1, \dots, n$.

158

Closure under substitution

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

159

Closure under context

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

160

Exercise

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

$$\begin{aligned} (x + y) + z &\rightarrow x + (y + z) \\ x * s(y) &\rightarrow x + (y * x) \end{aligned}$$

161

Remarks

- Ackermann function is an example of total recursive function that **cannot** be defined by primitive recursion.
- One shows that the function grows more quickly than any primitive recursive function.

```
#ack(2,2);;
- : int = 4
#ack(3,3);;
- : int = 16
#ack(4,4);;
Uncaught exception: Out_of_memory
```

- In particular, Ackermann cannot be shown to be terminating by a polynomial interpretation.

163

Exercise (Ackermann)

$$\begin{aligned} ack(z, n) &\rightarrow s(z) \\ ack(s(z), z) &\rightarrow s^2(z) \\ ack(s^2(m), z) &\rightarrow s^2(m) \\ ack(s(m), s(n)) &\rightarrow ack(ack(m, s(n)), n) \end{aligned}$$

Prove termination by RPO.

162

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?

164

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 .

165

RPO is a reduction order
via the reducibility candidates
method

167

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. Existence of a rpo for a TRS is an **NP-complete** problem (we guess an order).
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**.

166

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 λ -calculi.

168

Simplified version of RPO

Assume: (i) If $f >_{\Sigma} g$ and $g >_{\Sigma} f$ then $f = g$, and (ii) functions' arguments are always compared with the lexicographic order from left to right.

$$\frac{s \geq_r t}{f(\dots s \dots) >_r t}$$

$$\frac{f >_{\Sigma} g \quad f(s_1, \dots, s_m) >_r t_i \quad i = 1, \dots, n}{f(s_1, \dots, s_m) >_r g(t_1, \dots, t_n)}$$

$$\frac{(s_1, \dots, s_m) >_r^{lex} (t_1, \dots, t_m) \quad f(s_1, \dots, s_m) >_r t_i \quad i = 1, \dots, m}{f(s_1, \dots, s_m) >_r f(t_1, \dots, t_m)}$$

169

The key property

- Let $>_r^{lex}$ be the lexicographic order induced by $>_r$ on vectors of n terms in WF .

- The **key property** is:

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

Exercise Assuming the key property, derive by induction on the structure of the terms that all terms are in WF .

171

The well-founded set of terms WF

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

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

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

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 λ -calculus, the set WF (well-founded terms) will be called SN (strongly normalisable terms).

170

The induction principle at work

We prove the key property by **induction on the triple**:

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

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:

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

Two vectors of different lengths are **incomparable**.

Also $(WF, >_r)$ well-founded implies $(WF^n, >_r^{lex})$ is well-founded too.

172

Proof by induction of key property (2/3)

Proof by induction of key property (1/3)

Case $f(s_1, \dots, s_n) >_r t$ 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 $s_i \in WF$.

173

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

- We notice that $(f, (s_1, \dots, s_n), |t|) > (f, (s_1, \dots, s_n), |t_i|)$ for $i = 1, \dots, m$. Hence by inductive hypothesis, $t_i \in WF$.
- Suppose $g(t_1, \dots, t_m) >_r u$. We remark $(f, (s_1, \dots, s_n), |t|) > (g, (t_1, \dots, t_m), |u|)$. Hence by ind. hyp. $u \in WF$ and by a preliminary remark $g(t_1, \dots, t_m) \in WF$.

174

Proof by induction of key property (3/3)

Case $t = f(t_1, \dots, t_n)$, $f(s_1, \dots, s_n) >_r t_i$ for $i = 1, \dots, n$, $(s_1, \dots, s_n) >_r^{lex} (t_1, \dots, t_n)$.

This case is similar to the previous one.

- We remark that $(f, (s_1, \dots, s_n), |t|) > (f, (s_1, \dots, s_n), |t_i|)$ for $i = 1, \dots, n$. Hence by ind. hyp. $t_i \in WF$.
- Suppose $f(t_1, \dots, t_n) >_r u$. We notice $(f, (s_1, \dots, s_n), |t|) > (f, (t_1, \dots, t_n), |u|)$ (second component decreases!). By ind. hyp. $u \in WF$ and by a preliminary remark $f(t_1, \dots, t_n) \in WF$.

175

Well-partial orders Kruskal theorem and applications

176

Exercise (Dickson's lemma as a worm up)

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

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

1. Show that $>$ (the strict part of \geq) is well-founded.
2. Show by induction on k , that from every sequence $\{v_n\}_{n \in \mathbf{N}}$ in \mathbf{N}^k we can extract a **growing subsequence**, namely there is an injective function $\sigma : \mathbf{N} \rightarrow \mathbf{N}$ such that:

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

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

177

Well-partial order vs. well-founded order

Well partial orders are the **well-founded orders** that have **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.

179

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 $\{a_i \mid i \in \mathbf{N}\}$ in A ,

$$\exists i, j \in \mathbf{N} \ i < j \text{ and } a_i \leq a_j$$

Such a sequence is called **good**.

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

$$\forall i, j \in \mathbf{N} \ (i < j \text{ implies } a_i \not\leq a_j)$$

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**.

178

(\Leftarrow)

- 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.

180

Extraction of an ascending sequence in a wpo

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

$$i_1 < i_2 < i_3 < \dots \text{ and } a_{i_1} \leq a_{i_2} \leq a_{i_3} \leq \dots$$

- 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 $i_1 < i_2 < \dots$ such that $a_{i_1} \leq a_{i_2} \leq \dots$

181

Kruskal theorem

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

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

NB If Σ or V are **infinite** then $(T_\Sigma(V), \triangleright)$ contains an **infinite anti-chain** and the theorem does **not** hold.

183

Product of wpo's

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

- Consider $\{(a_i, b_i) \mid i \in \mathbf{N}\}$ and suppose $\{a_i \mid i \in \mathbf{N}\}$ and $\{b_i \mid i \in \mathbf{N}\}$ are both infinite (otherwise it is easy).
- Consider the subsequence $i_0 < i_1 < i_2 < \dots$ such that $a_{i_0} \leq a_{i_1} \leq a_{i_2} \leq \dots$ (cf. previous proposition).
- Find $l < k$ such that $b_{i_k} \leq b_{i_l}$.

182

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, \dots . This means that having built the sequence t_1, \dots, t_i , we pick a term t_{i+1} of minimal size among those that follow t_i .
- Define:

$$S_i = \begin{cases} \emptyset & \text{if } t_i \text{ variable} \\ \{s_1, \dots, s_n\} & \text{if } t_i = f(s_1, \dots, s_n) \end{cases} \quad \left| \quad S = \bigcup_{i \geq 0} S_i$$

- Show that (tricky point!): (S, \triangleright) is a **wpo**.

184

Proof Kruskal theorem continued

- Since Σ 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 $i_0 < i_1 < \dots$ with

$$t_{i_k} = f(s_1^{i_k}, \dots, s_n^{i_k})$$

- Now (S, \triangleright) is a wpo and the product of wpo's is a wpo. Therefore $\{(s_1^{i_k}, \dots, s_n^{i_k})\}_{k \geq 0}$ is a **good** sequence.
- So $\exists p, q$ $p < q$ and $s_l^{i_q} \triangleright s_l^{i_p}, l = 1, \dots, n$.
- And this entails $t_{i_q} \triangleright t_{i_p}$. **Contradiction**.

185

More on the tricky point (2/2)

- Since t_1, t_2, \dots and s_1, s_2, \dots are **bad**, this means

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

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

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

187

More on the tricky point (1/2)

(S, \triangleright) is a wpo.

- Suppose not, and let $s_1, s_2, s_3 \dots$ 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 $S_{<k} = S_1 \cup \dots \cup S_{k-1}$. There is an index l such that $s_i \notin S_{<k}$ for $i \geq l$.
- Consider

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

Since s_1 is smaller than t_k , **by minimality** (!) this sequence must be **good**.

186

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.

188

Exercise (Higman's lemma)

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

- Let Σ 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.

189

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**.

190

Proof (1/2)

By **contradiction**, suppose $t_1 > t_2 > \dots$

- First, we prove by contradiction that $Var(t_1) \supseteq Var(t_2) \supseteq \dots$:
 - Suppose $x \in Var(t_{i+1}) \setminus Var(t_i)$ 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.

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**.

191

192

Proof (2/2)

- Thus we can take $X = V(t_1)$ which is **finite**.
- Then we can **apply Kruskal theorem** to the sequence and conclude:

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

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

NB Note that we use $(T_\Sigma(X), \triangleright)$ **wpo**, not just well-founded...

193

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!

195

Application 2: Interpretation over the reals

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

With every $f^n \in \Sigma$ associate a multivariate polynomial

$p_f(x_1, \dots, x_n)$ such that:

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

194

Summary

1. $(T_\Sigma(V), \triangleright)$ is a **well partial order** (assuming Σ 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 **simplification orders**.

196

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>

197

- 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**.

199

Confluence and Completion

198

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 **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 **not** a variable.
- Let S be the **most general unifier** of l'_1 and l_2 (if it exists).
- Then $(S(r_1), S(C[r_2]))$ is a **critical pair**.

200

Preliminary remark

- The **domain** of a substitution S is the set $dom(S) = \{x \in V \mid S(x) \neq x\}$.
- Given two substitutions S_1, S_2 let us define their **union** as follows:

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

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

201

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

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

203

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).

202

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

204

Case 2

There is a variable x in l_1 such that S_2l_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 $f(g(x), g(x), g(x))$.

205

Exercise

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?

207

Case 3

We can decompose l_1 in $C[l'_1]$ so that:

$$l'_1 \text{ is not a variable and } S_1l'_1 = S_2l_2$$

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 $f(f(i(x_1), x_1), z)$.

206

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.

208

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,...

209

- Another critical pair is:

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

Again any simplification ordering satisfies:

$$(x * (x' * y')) * y' > (x' * y') .$$

- 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**.

211

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: $y' * ((y' * z') * z) > (y' * z')$.

210

Example (failure)

- The equations for left/right distributivity of $*$ over $+$ are:

$$x * (y + z) = (x * y) + (x * z), \quad (u + v) * w = (u * w) + (v * w)$$

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

$$(u+v)*(y+z) \rightarrow ((u+v)*y)+((u+v)*z), \quad (u*(y+z))+((v*(y+z)))$$

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

$$((u*y)+(v*y))+((u*z)+(v*z)), \quad ((u*y)+(u*z))+((v*y)+(v*z))$$

and there is no reasonable way to order them.

212

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:

$$x + s(s(z)) \rightarrow s(s(x))$$

213

Summary

1. **Critical pair test** to prove (**local**) **confluence** of TRS.
2. Idea of **Knuth-Bendix completion** to build **confluent and terminating** TRS.

215

- 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:

$$x + s(y) \rightarrow s(x + y)$$

214

Recommended reading

F. Baader, T. Nipkow, Term rewriting and all that, Cambridge University Press. Sections 6.1, 6.2, and 7.1.

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://col06-c703.uibk.ac.at/mkbtt/interface/index.php>

216

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?

217

λ -calculus

219

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 $g(f(x)) \rightarrow f(g(x))$.

218

History

- The λ -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, λ -calculus nevertheless provided one of the concurrent formalizations of **partial recursive functions** (= computable functions).
- Logical interest in the λ -calculus was resumed by Girard's discovery of the **second order** λ -calculus in the early seventies.

220

- In **computer science**, the interest in λ -calculus goes back to Landin (1966) and Reynolds (1970).
- The λ -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 λ -calculus as a meta-language for the description of the **denotational semantics** of programming languages.

221

- A number of programming operations can be introduced as **syntactic sugar**. For instance, the operation `let $x = M$ 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 ϕ .
- We denote with $FV(M)$ the variables occurring **free** in the term M .

223

Syntax

- We consider ‘pure’ functional expressions

$$M ::= id \mid (\lambda id.M) \mid (MM)$$

where $id ::= x \mid y \mid \dots$

- 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

```
function x -> M
```

NB One can write all λ -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**).

222

Substitution

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

$$\begin{aligned} [N/x]x &= N \\ [N/x]y &= y \text{ if } y \neq x \\ [N/x](M_1M_2) &= [N/x]M_1[N/x]M_2 \\ [N/x](\lambda y.M) &= \lambda z.[N/x][z/y]M \text{ if } z \notin FV(MN) \end{aligned}$$

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

224

Contexts

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

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

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

$$\begin{aligned} [N] &= N \\ (\lambda x.C)[N] &= \lambda x.C[N] \\ (CM)[N] &= C[N]M \\ (MC)[N] &= MC[N] \end{aligned}$$

225

β -reduction and β -conversion

The **β -rule** is the following relation between λ -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 **β -redex** (the subterm which is transformed).

We denote with $=_{\beta}$ the associated **conversion** relation, *i.e.*, $\overset{*}{\leftrightarrow}_{\beta}$.

NB This is **NOT** a TRS but there are proposals to turn it into a TRS by making the substitution operation explicit.

227

α -renaming

- λ -terms (like formulae or integrals!) are always manipulated up to the **renaming of bound variables**.
- Formally, we define a binary relation \equiv on terms called **α -conversion** which is the least equivalence relation containing the following pairs:

$$C[\lambda x.M] \equiv C[\lambda y.[y/x]M] \quad \text{if } y \notin FV(M)$$

226

Examples

Here are some **basic λ -terms**:

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

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

$$\begin{aligned} II &\rightarrow I \\ SKK &\rightarrow I \\ \Delta\Delta &\rightarrow \Delta\Delta \\ \Delta_f\Delta_f &\rightarrow f(\Delta_f\Delta_f) \end{aligned}$$

NB $\lambda x_1, \dots, x_n.M \equiv \lambda x_1 \dots \lambda x_n.M$ and application associates to the left, *i.e.*, $M_1M_2 \dots M_n \equiv (\dots(M_1M_2) \dots M_n)$.

228

Exercise (characterisation normal forms)

Let NF be the smallest set of λ -terms such that:

$$\frac{M_i \in NF \quad i = 1, \dots, k}{\lambda x_1 \dots x_n. x M_1 \dots M_k \in NF}$$

Show that NF is exactly the set of λ -terms in β -normal form.

Exercise (another fixed point combinator)

Turing's fixed point combinator is defined by:

$$Y_T = (\lambda xy. y(xxy))(\lambda xy. y(xxy))$$

Show that that $Y_T f$ is not only convertible to, but **reduces to**, $f(Y_T f)$.

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 (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 β -reduction is **locally confluent**:

$$\frac{\forall M, N, P (M \rightarrow N, \quad M \rightarrow P)}{\exists Q (N \xrightarrow{*} Q, \quad P \xrightarrow{*} Q)}$$

Church-Rosser theorem

β -reduction is **confluent**. That is:

$$\frac{\forall M, N, P (M \rightarrow^* N \quad M \rightarrow^* P)}{\exists Q (N \rightarrow^* Q \quad P \rightarrow^* Q)}$$

233

Definition Parallel Reduction

$$\frac{}{M \Rightarrow M} \quad \frac{M \Rightarrow M' \quad N \Rightarrow N'}{(\lambda x.M)N \Rightarrow [N'/x]M'}$$

$$\frac{M \Rightarrow M' \quad N \Rightarrow N'}{MN \Rightarrow M'N'} \quad \frac{M \Rightarrow M'}{\lambda x.M \Rightarrow \lambda x.M'}$$

235

Towards a proof

- Let \mapsto be the **reflexive** closure of \rightarrow . Notice that a strong confluence property such as:

$$\frac{\forall M, N, P (M \rightarrow N, \quad M \rightarrow P)}{\exists Q (N \mapsto Q, \quad P \mapsto Q)}$$

fails because as a result of β -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:
 - $\rightarrow \subset \Rightarrow \subset \rightarrow^*$.
 - The **strong confluence property** holds for \Rightarrow (and \Rightarrow is simple enough for a proof to go through!)

234

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 ?

236

Confluence of parallel reduction (1/3)

First observe the following **structural properties**:

$$\frac{\lambda x.M \Rightarrow N}{N \equiv \lambda x.M', \quad M \Rightarrow M'}$$

$$\frac{MN \Rightarrow L}{(L \equiv M'N', \quad M \Rightarrow M', \quad N \Rightarrow N') \text{ or } (M \equiv \lambda x.P, \quad P \Rightarrow P', \quad N \Rightarrow N', \quad L \equiv [N'/x]P')}$$

237

Confluence of parallel reduction (3/3)

Finally, show a **strong confluence** property:

$$\frac{\forall M, N_1, N_2 (N_1 \Leftarrow M \Rightarrow N_2)}{\exists P (N_1 \Rightarrow P \Leftarrow N_2)}$$

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

Finally, from

$$\rightarrow_\beta \subset \Rightarrow \subset \overset{*}{\rightarrow}_\beta$$

conclude that \rightarrow_β is confluent.

239

Confluence of parallel reduction (2/3)

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

$$\frac{M \Rightarrow M' \quad N \Rightarrow N'}{[N/x]M \Rightarrow [N'/x]M'}$$

238

η -rule

In addition to β , another rule is often considered:

$$(\eta) \quad C[\lambda x.Mx] \rightarrow C[M] \quad (x \notin FV(M)).$$

This is an **extensionality rule**, asserting that ‘every term is a function’ (if it is read backwards).

240

Exercise (confluence for $\beta\eta$)

1. Show that η reduction is strongly **confluent** in the following sense:

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

2. Show that β and η reductions **commute** in the following sense:

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

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

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

4. Conclude that $\beta\eta$ reduction is **confluent**.

241

Combinatory Logic

- We consider a binary operation $@$ and two constants K and S .
- As in the λ calculus, we write MN for $@(M, N)$ and associate to the left.
- We have two term rewriting rules:

$$\begin{aligned} Kxy &\rightarrow x \\ Sxyz &\rightarrow xz(yz) \end{aligned}$$

- We abbreviate SKK as I .

243

Example (non-confluence)

- Here is an extension of the λ -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 pairing** 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 λ -calculi.

242

- One can **simulate** the λ -abstraction of a term M of combinatory logic as follows:

$$\begin{aligned} \underline{\lambda x.x} &= I \\ \underline{\lambda x.M} &= KM \quad \text{if } x \notin \text{Var}(M) \\ \underline{\lambda x.MN} &= S(\underline{\lambda x.M})(\underline{\lambda x.N}) \end{aligned}$$

- By induction on M verify that:

$$(\underline{\lambda x.M})N \xrightarrow{*} [N/x]M$$

244

The λ -calculus is ‘Turing equivalent’

- 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 β 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.

245

Minimalisation (fact or reminder)

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

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

NB Recursive definitions provide a direct mechanism to mimick definitions by minimalisation.

247

- 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**

246

Number representation

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

$$\underline{n} = \lambda f. \lambda x. (f \cdots (f x) \cdots)$$

where f is applied n times.

- In some sense this is similar to **tally natural numbers** and we will see later in the course (system F) that the inductive definition of natural numbers actually suggests their representation in the λ -calculus.

248

Function representation

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

$$\begin{aligned} f(n_1, \dots, n_k) = m & \text{ iff } F\underline{n_1} \cdots \underline{n_k} = \underline{m} \\ f(n_1, \dots, n_k) \uparrow & \text{ iff } F\underline{n_1} \cdots \underline{n_k} \text{ has no (head) normal form.} \end{aligned}$$

249

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** $P1$, $P2$:

$$P = \lambda x.\lambda y.\lambda z.zxy \quad P1 = \lambda p.p(\lambda x, y.x) \quad P2 = \lambda p.p(\lambda x, y.y)$$

251

Exercise (representation of basic functions)

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

250

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:

$$(\lambda f.N)(Y(\lambda f.\lambda x.M(f)))$$

It follows that one can represent definitions by **primitive recursion** and **minimalisation**.

252

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.

253

- Finally, for suitable choices of the types, the λ -calculus is actually a **logical system** (two examples are Martin-Löf type theory and the Calculus of Constructions) following the so-called **Curry-Howard correspondance** which goes as follows:

λ -calculus	proof system
type	proposition
λ -term	proof
reduction	proof normalisation

255

- 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).

254

References

Barendregt. *The lambda calculus: its syntax and semantics*. Elsevier.

This is **the** standard reference for the ‘pure’ (type-free) λ -calculus. You can just skim through the first introductory chapters (chapter 3 in particular) to have an idea of the variety of results.

256

Simple types

257

Terms and typing rules

$$(\text{asm}) \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$(\rightarrow_I) \quad \frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x : \tau. M : \tau \rightarrow \tau'}$$

$$(\rightarrow_E) \quad \frac{\Gamma \vdash M : \tau \rightarrow \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'}$$

NB We have: (1) a structural/identity rule: (*asm*), (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).

259

Syntax: types and contexts

- We define the collection of **simple types** as follows

$$\tau ::= b \mid tid \mid (\tau \rightarrow \tau)$$

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

- An **environment** Γ is a set of pairs $\{x_1 : \tau_1, \dots, x_n : \tau_n\}$ where all variables x_1, \dots, x_n are **distinct**.
- We use $\Gamma, x : \tau$ as an **abbreviation** for $\Gamma \cup \{x : \tau\}$ where x is not in Γ .

258

Exercise (types and propositions)

1. Show that if $x_1 : \tau_1, \dots, x_n : \tau_n \vdash M : \tau$ is derivable then $(\tau_1 \rightarrow \dots (\tau_n \rightarrow \tau) \dots)$ is a **tautology** of propositional logic where we interpret \rightarrow as implication and atomic types as propositional variables.
2. Conclude that there are types τ which are **not inhabited**, *i.e.*, there is no (closed) term M such that $\emptyset \vdash M : \tau$.

260

Exercise (inhabited types)

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

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

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

$$\emptyset \vdash N_1 : \tau \rightarrow (\neg\neg\tau)$$

$$\emptyset \vdash N_2 : (\neg\neg\neg\tau) \rightarrow (\neg\tau)$$

On the other hand, there are types τ such that for no term N_3 we can derive $\emptyset \vdash N_3 : \neg\neg\tau \rightarrow \tau$. So not all tautologies are inhabited!

261

Alternative presentations: Variable labelling

We label every variable with its type and we drop the context:

$$id ::= x \mid y \mid \dots$$

$$lid ::= id^\sigma$$

$$M ::= lid \mid \lambda lid.M \mid MM$$

$$\frac{}{x^\sigma : \sigma} \quad \frac{M : \sigma \rightarrow \tau \quad N : \sigma}{MN : \tau} \quad \frac{M : \tau}{\lambda x^\sigma.M : \sigma \rightarrow \tau}$$

263

Alternative presentations: Curry style

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

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \rightarrow \tau}$$

In this presentation, a term may have more than one type. Typically, type inference problems are studied in this framework.

262

Alternative presentations: Term labelling

We label every term (not just the variables) with its type and we drop the context and the type:

$$id ::= x \mid y \mid \dots$$

$$M ::= id^\sigma \mid (\lambda id^\sigma.M)^\sigma \mid (MM)^\sigma$$

$$\frac{}{x^\sigma} \quad \frac{M^{\sigma \rightarrow \tau} \quad N^\sigma}{(M^{\sigma \rightarrow \tau} N^\sigma)^\tau} \quad \frac{M^\tau}{(\lambda x^\sigma.M^\tau)^{(\sigma \rightarrow \tau)}}$$

264

Logical extensions of natural deduction

$$\begin{array}{c}
 (\times_I) \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash \langle M_1, M_2 \rangle : \tau_1 \times \tau_2} \\
 \\
 (\times_{E,1}) \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1(M) : \tau_1} \quad (\times_{E,2}) \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2(M) : \tau_2} \\
 \\
 (+_{I,1}) \frac{\Gamma \vdash M : \tau_1}{\Gamma \vdash \text{inl}_{\tau_1 + \tau_2}(M) : \tau_1 + \tau_2} \quad (+_{I,2}) \frac{\Gamma \vdash M : \tau_2}{\Gamma \vdash \text{inr}_{\tau_1 + \tau_2}(M) : \tau_1 + \tau_2} \\
 \\
 (+_E) \frac{\Gamma \vdash M : (\tau_1 + \tau_2) \quad \Gamma \vdash N_i : \tau_i \rightarrow \sigma \quad i = 1, 2}{\Gamma \vdash (\text{case } M \ N_1 \ N_2) : \sigma}
 \end{array}$$

265

Subject Reduction (motivations)

- If a term is **well-typed**, then by inspection of the rules we see, *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 **typing is invariant under reduction**, *i.e.*, if a term is well-typed and we reduce it then we still get a well-typed term.

267

Non-Logical extensions of natural deduction

$$\begin{array}{c}
 (Z) \frac{}{\Gamma \vdash Z : \text{nat}} \\
 \\
 (S) \frac{\Gamma \vdash M : \text{nat}}{\Gamma \vdash SM : \text{nat}} \\
 \\
 (Y) \frac{\Gamma \vdash M : \tau \rightarrow \tau}{\Gamma \vdash YM : \tau}
 \end{array}$$

NB With the rule (Y), every type τ is **inhabited** by the closed term $Y(\lambda x : \tau. x)$. Thus Y leads to logical inconsistency!

266

Substitution lemma

To establish this property we note the following lemma.

If $\Gamma, x : \tau \vdash M : \tau'$ and $\Gamma \vdash N : \tau$ then $\Gamma \vdash [N/x]M : \tau'$.

268

Proof of the substitution lemma

- By induction on the **height of the proof** of $\Gamma, x : \tau \vdash M : \tau'$.
- For instance, suppose the **root of the proof** has the shape:

$$\frac{\Gamma, x : \tau, y : \tau' \vdash M : \tau''}{\Gamma, x : \tau \vdash \lambda y. M : \tau' \rightarrow \tau''}$$

with $x \neq y$.

- Then by inductive hypothesis, $\Gamma, y : \tau' \vdash [N/x]M : \tau''$ and we conclude by (\rightarrow_I) .

269

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) $C = PC'$.

271

Subject Reduction (formal statement)

If $\Gamma \vdash M : \tau$ and $M \rightarrow_\beta N$ then $\Gamma \vdash N : \tau$.

270

Outcomes of a program

- In the ‘pure’ λ -calculus, we assume:
normal forms = results
- In applications, one often decomposes the results as follows:
normal forms = values \cup stuck/erroneous config.
- Thus a program (a closed term) has three possible outcomes:
 - It **returns a value**.
 - It reaches an **erroneous configuration**.
 - It **diverges**.

272

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.

273

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:

$$\delta(t) = 1 \quad \delta(\sigma \rightarrow \tau) = 1 + \max(\delta(\sigma), \delta(\tau))$$

- The **degree of a redex**

$$\delta_r((\lambda x : \sigma.M)N),$$

is the degree of the type associated with the redex $(\lambda x : \sigma.M)$.

- 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 $\delta_r(R) \leq \delta_t(R)$.

275

Exercise (progress)

- Suppose we reconsider the non-logical extension of the simply typed λ -calculus with a **basic type** nat , **constants** Z, S, Y , and with the following fixed-point rule:

$$C[YM] \rightarrow C[M(YM)]$$

- Let a **program** be a closed typable term of type nat and let a **value** be a term of the shape $(S \cdots (SZ) \cdots)$.
- Show that if P is a program in **normal form** (cannot reduce), then P is a **value**.

274

Degrees and substitution

If x is of type σ then

$$\delta_t([N/x]M) \leq \max(\delta(\sigma), \delta_t(M), \delta_t(N))$$

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 $\delta(\sigma)$.

276

Degrees and reduction

If $M \rightarrow N$ then $\delta_t(N) \leq \delta_t(M)$.

Proof We apply the previous analysis.

277

Towards strong normalization

- A λ -term M is called **strongly normalizable** if all β -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:

$$\begin{aligned} size(x) &= 1 \\ size(MN) &= size(M) + size(N) + 1 \\ size(\lambda x.M) &= size(M) + 1 \end{aligned}$$

- 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!).

279

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 .

278

Strong normalization

Theorem All simply typed λ -terms are \rightarrow_β -strongly normalizable.

Proof The **key idea** is to interpret types as follows:

$$\begin{aligned} \llbracket t \rrbracket &= SN \\ \llbracket \sigma \rightarrow \tau \rrbracket &= \{M \mid \forall N \in \llbracket \sigma \rrbracket \ (MN \in \llbracket \tau \rrbracket)\} \end{aligned}$$

NB Recall that for RPO, we showed:

$$s_1, \dots, s_n \in WF \text{ implies } f(s_1, \dots, s_n) \in WF$$

280

Properties of the interpretation

1. $\llbracket \sigma \rrbracket \subseteq SN$.
2. If $N_i \in SN$ for $i = 1, \dots, k$ then $xN_1 \dots N_k \in \llbracket \sigma \rrbracket$.
3. If $[N/x]MM_1 \dots M_k \in \llbracket \sigma \rrbracket$ and $N \in SN$ then $(\lambda x.M)NM_1 \dots M_k \in \llbracket \sigma \rrbracket$.

NB These interpretations of types are called **reducibility candidates**. These are sets of strongly normalisable terms (property 1) which contain at least the variables (and more) (property 2), and are closed under head expansions (property 3).

281

Functional types $\tau \rightarrow \sigma$ Suppose $M \in \llbracket \tau \rightarrow \sigma \rrbracket$.

1. By ind. hyp. $x \in \llbracket \tau \rrbracket$. Hence $Mx \in \llbracket \sigma \rrbracket \subseteq SN$, by ind. hyp. This entails $M \in SN$.
2. Take $M = xN_1 \dots N_k$ with $N_i \in SN$. Take $N_{k+1} \in \llbracket \tau \rrbracket \subseteq SN$.
By ind. hyp. $xN_1 \dots N_k N_{k+1} \in \llbracket \sigma \rrbracket$.
3. Then:

$$\begin{aligned}
 & [N/x]MN_1 \dots N_k \in \llbracket \tau \rightarrow \sigma \rrbracket \\
 & \forall N_{k+1} \in \llbracket \tau \rrbracket \quad [N/x]MN_1 \dots N_k N_{k+1} \in \llbracket \sigma \rrbracket \\
 & \forall N_{k+1} \in \llbracket \tau \rrbracket \quad (\lambda x.M)NN_1 \dots N_k N_{k+1} \in \llbracket \sigma \rrbracket \quad (\text{by ind. hyp.}) \\
 & (\lambda x.M)NN_1 \dots N_k \in \llbracket \tau \rightarrow \sigma \rrbracket
 \end{aligned}$$

283

Proof of the properties

By induction on σ .

Atomic types

1. By definition.
2. The reductions of $xN_1 \dots N_k$ are just an interleaving of the reductions of N_1, \dots, N_k .
3. We have:

$$depth((\lambda x.M)NM_1 \dots M_k) \leq depth(N) + depth([N/x]MM_1 \dots M_k) + 1.$$

282

Soundness of the interpretation

If $x_1 : \sigma_1, \dots, x_k : \sigma_k \vdash M : \tau$ (in Curry style) and $N_i \in \llbracket \sigma_i \rrbracket$ for $i = 1, \dots, k$ then $[N_1/x_1, \dots, N_k/x_k]M \in \llbracket \tau \rrbracket$.

284

Corollary of soundness: Strong normalisation

- Suppose $x_1 : \sigma_1, \dots, x_k : \sigma_k \vdash M : \tau$.
- We know $x_i \in \llbracket \sigma_i \rrbracket$.
- By the soundness $M \in \llbracket \tau \rrbracket$.
- We know $\llbracket \tau \rrbracket \subseteq SN$.

285

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:

$$\frac{\Gamma \vdash M : \sigma \quad \sigma = \tau}{\Gamma \vdash M : \tau}$$

Show that in this case the following λ -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?

287

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.

$$\begin{aligned} x : \sigma \vdash M : \tau \\ \forall N \in \llbracket \sigma \rrbracket \quad [N/x]M \in \llbracket \tau \rrbracket \quad (\text{ind. hyp.}) \\ \forall N \in \llbracket \sigma \rrbracket \quad (\lambda x. M)N \in \llbracket \tau \rrbracket \quad (\text{saturation}) \\ (\lambda x. M) \in \llbracket \sigma \rightarrow \tau \rrbracket \end{aligned}$$

286

Summary

- We regard **Types as Propositions** and λ -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).

288

Recommended reading

Jean-Yves Girard et al. Proofs and Types. Cambridge University Press.

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.

289

The type inference problem

- Given a (pure) term M and a context Γ , the **type inference problem** is the problem of checking whether there is a type τ such that $\Gamma \vdash M : \tau$.
- Given a (pure) term M , a **variant of the problem** is to look for a type τ and a context Γ such that $\Gamma \vdash M : \tau$.
- 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**.

291

Type inference and its reduction to syntactic unification

290

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.

292

Reduction (formal definitions)

- A **goal** is a finite set G of triples (Γ, M, τ) where Γ is a **context**, M a **λ -term**, and τ a **simple type**.
- We assume that all bound variables in M are **distinct**, that all free variables occur in the context Γ , and that for every variable x we have a type variable t_x .
- We define a **reduction relation** on pairs (G, E) .

293

Termination

It is enough to notice that every reduction step replaces a triple (Γ, M, τ) by a finite number of triples (Γ', M', τ') where M' is structurally smaller than M .

295

- Assuming $G = \{g\} \cup G'$ and $g \equiv (\Gamma, M, \tau) \notin G'$ all the rules produce a pair $(G' \cup G_g, E \cup E_g)$ where G_g and E_g are defined as follows:

g	G_g	E_g
(Γ, x, τ)	\emptyset	$\{t_x = \tau\}$
$(\Gamma, M_1 M_2, \tau)$	$\{(\Gamma, M_1, t_1 \rightarrow \tau), (\Gamma, M_2, t_1)\}$	\emptyset (t_1 fresh)
$(\Gamma, \lambda x.M_1, \tau)$	$\{(\Gamma, x : t_x, M_1, t)\}$	$\{\tau = t_x \rightarrow t\}$ (t fresh)

294

Some notation

- In the following, we consider substitutions S that act on the algebra $T_\Sigma(Tvar)$ where $\Sigma = \{b, \rightarrow\}$.
- We define:

$$\begin{aligned}
 S \models E & && \text{if } S \text{ unifies } E \\
 S \models (\Gamma, M, \tau) & && \text{if } S\Gamma \vdash M : S\tau \text{ is derivable} \\
 S \models G & && \text{if } \forall g \in G \ S \models g \\
 S \models (G, E) & && \text{if } S \models G \text{ and } S \models E
 \end{aligned}$$

- Given a term M_0 with free variables x_1, \dots, x_n we set the initial pair to (G_0, \emptyset) with $G_0 = \{(\Gamma_0, M_0, t_0)\}$ and $\Gamma_0 = x_1 : t_{x_1}, \dots, x_n : t_{x_n}$.

296

Soundness and Completeness

If $(G_0, \emptyset) \rightarrow^* (G, E)$ then:

1. If $S \models (G, E)$ then $S\Gamma_0 \vdash M_0 : St_0$.
2. If $\Gamma \vdash M_0 : \tau$ then $\exists S(S \models (G, E), S\Gamma_0 \subseteq \Gamma, \text{ and } \tau = St_0)$.

(1) and (2) hold for the initial goal and every application of the rules maintains these properties.

297

Remark (on completeness)

- On the other hand, (2) proves the **completeness** of the method.
- Suppose $\Gamma \vdash M_0 : \tau$ is a **valid typing**.
- Then we can reduce (Γ_0, M_0, t_0) to (\emptyset, E) and find a **unifier** S for E such that $S\Gamma_0$ is contained in Γ and $St_0 = \tau$.

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 .

299

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**

$$S\Gamma_0 \vdash M_0 : St_0$$

298

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.

300

Proof of soundness

- For instance, suppose (1) true for

$$(G \cup \{(\Gamma, MN, \tau)\}, E)$$
- The rule for application produces the pair (G', E) with

$$G' = G \cup \{(\Gamma, M, t_1 \rightarrow \tau), (\Gamma, N, t_1)\}.$$
- Suppose $S \models (G', E)$. This means $S \models (G, E)$,
 $S\Gamma \vdash M : S(t_1 \rightarrow \tau)$, and $S\Gamma \vdash N : St_1$.
- By (\rightarrow_E) , we conclude $S\Gamma \vdash MN : S\tau$.
- Thus $S \models (G \cup \{(\Gamma, MN, \tau)\}, E)$, and by hypothesis
 $S\Gamma_0 \vdash M_0 : St_0$.

301

Exercise

Compute, if they exist, the most general types of the following terms:

$$\lambda x. \lambda y. \lambda z. xz(yz)$$

$$\lambda x. \lambda y. x(yx)$$

$$\lambda k. (k(\lambda x. \lambda h. hx))$$

303

Proof of completeness

For instance, suppose

$$\begin{aligned} \Gamma \vdash M_0 : \tau, \quad S \models (G \cup \{(\Gamma', \lambda x.M, \tau')\}, E), \\ S\Gamma_0 \subseteq \Gamma \quad \tau = St_0 \end{aligned}$$

This implies:

$$S\Gamma' \vdash \lambda x.M : S\tau'$$

which entails:

$$S\Gamma', x : \tau_1 \vdash M : \tau_2, \quad S\tau' = \tau_1 \rightarrow \tau_2, \text{ for some } \tau_1, \tau_2.$$

Suppose we reduce to the pair:

$$(G \cup \{(\Gamma', x : t_x, M, t)\}, E \cup \{\tau' = t_x \rightarrow t\})$$

Then take $S' = S[\tau_1/t_x, \tau_2/t]$.

302

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.

304

Recommended reading

John Mitchell. Foundations of programming languages. Chapter 11 (Type inference).

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).

305

History

- The system F was introduced by **Girard** in his PhD thesis (1972) as a tool for the study of the **cut-elimination procedure** in **second order arithmetic** (PA_2).
- More precisely the **normalization of system F** implies the **termination of the cut-elimination procedure** in PA_2 (and thus the consistency of analysis!).
- By relying on this strong connection between system F and PA_2 it was proven that all functions that can be shown to be total in PA_2 are **representable** in system F.
- This is a **huge** collection of total recursive functions that goes well beyond the primitive recursive functions.
- System F was later rediscovered by **Reynolds** (1974) as a concise calculus of *type parametric* functions.

307

Polymorphic λ -calculus (system F)

306

Second order quantification in logic

$$\begin{array}{ll} (P \rightarrow P) \rightarrow (P \rightarrow P) & \text{(Propositional calculus)} \\ \forall x (P(x) \rightarrow P(S(x))) \rightarrow (P(Z) \rightarrow \forall x P(x)) & \text{(First-order quantification)} \\ \forall P (P \rightarrow P) \rightarrow (P \rightarrow P) & \text{(Second-order quantification)} \end{array}$$

308

The origin of System F

- System F is a **logical system** obtained from the propositional intuitionistic system (simple types as far as we are concerned) by introducing second order quantification.
- At the **type level**, this means that we can quantify over type variables:

$$t_0 \equiv \forall t (t \rightarrow t)$$

- At the **term level**, this means that we can abstract with respect to a type and apply a term to a type:

$$pid = \lambda t. \lambda x : t. x \quad \text{(a 'polymorphic' identity)}$$

$$pid \text{ nat} \quad : (nat \rightarrow nat) \quad \text{(a specialisation)}$$

$$pid \ t_0 \quad : (t_0 \rightarrow t_0) \quad \text{(another specialisation)}$$

NB In System F, the type quantification in t_0 quantifies on **all types** including t_0 itself. More predicative/stratified definitions are possible.

309

Typing rules (à la Church)

$$(asm) \quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$(\rightarrow_I) \quad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau} \quad (\rightarrow_E) \quad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$(\forall_I) \quad \frac{\Gamma \vdash M : \sigma \quad t \notin FV_t(\Gamma)}{\Gamma \vdash \lambda t. M : \forall t. \sigma} \quad (\forall_E) \quad \frac{\Gamma \vdash M : \forall t. \sigma}{\Gamma \vdash M\tau : \sigma[\tau/t]}$$

NB We just add the rules (\forall_I) and (\forall_E) to the system for simple types.

311

Syntax: types and terms

$$tid \quad ::= t \mid s \mid \dots \quad \text{(type variables)}$$

$$\sigma \quad ::= tid \mid \sigma \rightarrow \sigma \mid \forall tid. \sigma \quad \text{(types)}$$

$$id \quad ::= x \mid y \mid \dots \quad \text{(variables)}$$

$$M \quad ::= id \mid \lambda id : \sigma. M \mid MM \mid \lambda tid. M \mid M\sigma \quad \text{(terms)}$$

$$\Gamma \quad ::= id : \sigma, \dots, id : \sigma \quad \text{(contexts)}$$

We denote with $FV_t(\Gamma)$ the collection of type variables that occur free in types occurring in Γ .

310

Exercise (on the side condition for (\forall_I))

Show that without the side condition in rule (\forall_I) one can build a closed term of type σ , for any type σ . In other terms without the side condition the system is inconsistent!

312

Reduction rules

The redexes of system F are:

$$\begin{aligned}(\lambda x : \sigma.M)N &\rightarrow [N/x]M & (\beta) \\ (\lambda t.M)\sigma &\rightarrow [\sigma/t]M & (\beta_t)\end{aligned}$$

and optionally:

$$\begin{aligned}(\lambda x : \sigma.Mx) &\rightarrow M & x \notin FV(M) & (\eta) \\ (\lambda t.Mt) &\rightarrow M & t \notin FV_t(M) & (\eta_t)\end{aligned}$$

313

Main results

We will focus on the following two main results:

1. Second-order quantification can be used to represent **inductive data types** and **iterative functions** on them.
2. The typing discipline guarantees **termination**.

315

Exercise (subject reduction and local confluence)

Show that in system F with $\beta\eta$ -reduction the following properties hold:

1. **Subject reduction** (typing is preserved by reduction).
2. **Local confluence**.

NB Confluence follows once we have shown that all typable terms are **strongly normalisable**.

314

Exercise (iterative vs. primitive recursive)

- Iterative functions are defined on the ground (closed) terms of an arbitrary Σ -algebras.
- The basic idea is to define a function **by induction on the structure of a ground term**, hence we have as many cases as function symbols in Σ -algebras.

316

- Let us consider the Σ -algebra of **tally natural numbers** $\Sigma = \{S^1, Z^0\}$ and let $T = T_\Sigma(\emptyset)$ be the set of ground terms.
- Given $g : T^n \rightarrow T$ and $h : T^{n+1} \rightarrow T$ the function $f : T^{n+1} \rightarrow T$ is defined by **iteration** as follows:

$$\begin{aligned} f &= \\ Z, \vec{y} &\Rightarrow g(\vec{y}) \\ S(x), \vec{y} &\Rightarrow h(f(x, \vec{y}), \vec{y}) \end{aligned}$$

At first sight this is less powerful than primitive recursive definitions because the function h **does not** depend directly on x .

317

One checks by induction on $x \in T$ that:

$$f'(x, \vec{y}) = \langle f(x, \vec{y}), x \rangle$$

and from f' one obtains f by projection.

319

- However one can first define **pairing** and **projections** and then show that a function f defined by **primitive recursion** such as:

$$\begin{aligned} f &= \\ Z, \vec{y} &\Rightarrow g(\vec{y}) \\ S(x), \vec{y} &\Rightarrow h(f(x, \vec{y}), x, \vec{y}) \end{aligned}$$

can also be defined by **iteration** as follows:

$$\begin{aligned} f' &= \\ Z, \vec{y} &\Rightarrow \langle g(\vec{y}), Z \rangle \\ S(x), \vec{y} &\Rightarrow h'(f'(x, \vec{y}), \vec{y}) \\ \\ h'(x, \vec{y}) &= \langle h(\pi_1(x), \pi_2(x), \vec{y}), S(\pi_2(x))) \rangle \end{aligned}$$

318

Iterative functions

- Let \mathcal{S} be a Σ -algebra with function symbols (constructors) $c_i^{n_i}$, $i = 1, \dots, k$, $n_i \geq 0$. Let $T = T_\Sigma(\emptyset)$.
- The collection of **iterative functions** $f : T^n \rightarrow T$ is the smallest set such that:
 - The **basic functions** $c_i^{n_i}$, constant functions, and projection functions are iterative functions.
 - The set is closed under **composition**.

$$\frac{f_1 : T^m \rightarrow T, \dots, f_n : T^m \rightarrow T, g : T^n \rightarrow T \text{ iterative}}{g(f_1, \dots, f_n) \text{ iterative}}$$

320

– The set is closed under **iteration**.

$$\frac{h_i : T^{n_i+m} \rightarrow T, i = 1, \dots, k \text{ iterative}}{f : T^{m+1} \rightarrow T \text{ iterative}}$$

where for $i = 1, \dots, k$:

$$f(c_i(x_1, \dots, x_{n_i}), \vec{y}) = h_i(f(x_1, \vec{y}), \dots, f(x_{n_i}, \vec{y}), \vec{y})$$

321

From algebras to types

We associate:

- With a Σ -algebra \mathcal{S} a type σ of system F .
- With a ground term a of the algebra a closed term \underline{a} of system F of type σ .

322

Given $c_i^{n_i} : \underbrace{T \times \dots \times T}_{n_i \text{ times}} \rightarrow T, i = 1, \dots, k$

Let $\sigma \equiv \forall t. \tau_1 \rightarrow \dots \rightarrow \tau_k \rightarrow t$

Where $\tau_i \equiv \underbrace{t \rightarrow \dots \rightarrow t}_{n_i \text{ times}} \rightarrow t$

Then:

$$\begin{aligned} \underline{c_i^{n_i}} &= \lambda y_1 : \sigma \dots \lambda y_{n_i} : \sigma. \\ &\quad \lambda t. \lambda x_1 : \tau_1 \dots \lambda x_k : \tau_k. \\ &\quad x_i(y_1 t x_1 \dots x_k) \dots (y_{n_i} t x_1 \dots x_k) \\ &: \underbrace{\sigma \rightarrow \dots \rightarrow \sigma}_{n_i \text{ times}} \rightarrow \sigma \end{aligned}$$

323

Example (tally natural numbers)

- If we apply the coding method to the Σ -algebra of **tally natural numbers** we obtain the type:

$$\forall t. (t \rightarrow t) \rightarrow (t \rightarrow t) .$$

The term $S(\dots(SZ)\dots)$ is represented (up to conversion) by the term:

$$\lambda t. \lambda f : t \rightarrow t. \lambda x : t. f(\dots(fx)\dots)$$

which are the so-called **Church numerals**.

324

Exercise (more Σ -algebras encodings)

Make explicit the coding of the following Σ -algebras:

1. The algebra with **no operation**.
2. The algebra with **two 0-ary operations** (booleans!).
3. The algebra of **binary words**.
4. The algebra of **binary trees**.

325

Proof data representation (1/4)

- Let M be a closed term of system F in β -normal form of type σ , where σ is defined according to the rules.
- The existence of the β -normal form will be proved next.
- M has to have the shape:

$$M \equiv \lambda t. \lambda x_1 : \tau_1 \dots \lambda x_i : \tau_i. M' \quad i \leq k .$$

327

Data Representation

There is a bijective correspondence between the **ground terms** of a Σ -algebra \mathcal{S} and the **closed terms** of system F of the corresponding type σ modulo $\beta\eta$ -conversion.

326

Proof data representation (2/4)

- If $i < k$ and M' is not a λ -abstraction then M' has the shape $(\dots (x_j M_1) \dots M_h)$ and so we can η -expand M' without introducing a β -redex.
- By iterated η -expansions we arrive at a term in β normal form of the shape:

$$\lambda t. \lambda x_1 : \tau_1 \dots \lambda x_k : \tau_k. M'' ,$$

where M'' has type t , it is in β normal form, and may include free variables x_1, \dots, x_k .

- We note that the types of the variables x_i do not contain second order quantifications.

328

Proof data representation (3/4)

- We claim that M'' cannot contain a λ -abstraction:
 - A λ -abstraction on the left of an application would contradict the hypothesis that M is in β normal form.
 - A λ -abstraction on the right of an application is incompatible with the ‘first order’ types of the variables τ_i .

329

Function representation

Definition A function $f : T^n \rightarrow T$ is representable (with respect to the proposed coding) if there is a closed term $M : \sigma^n \rightarrow \sigma$, such that for any vector of ground terms \vec{a} ,

$$M\vec{a} =_{\beta\eta} \underline{f(\vec{a})} .$$

Theorem All **iterative functions** over an algebra \mathcal{S} are representable.

331

Proof data representation (4/4)

- We have shown that a closed term of type σ is determined up to $\beta\eta$ conversion by a term M'' which is a well-typed combination of the variables x_i , for $i = 1, \dots, k$.
- Since each variable corresponds to a constructor of the algebra we can conclude that there is a unique ground term of the algebra which corresponds to M'' .

330

Example: typing the iterator on tally natural numbers

- Suppose T is the set of **tally numbers** and $g : T \rightarrow T$ and $h : T^2 \rightarrow T$. The iteration $it(h, g)$ of h and g must satisfy:

$$\begin{aligned} it(h, g)(Z, y) &= g(y) \\ it(h, g)(S(x), y) &= h(it(h, g)(x, y), y) \end{aligned}$$

- In the **pure** λ -calculus we would define:

$$it = \lambda h. \lambda g. \lambda x. \lambda y. x (\lambda z. h z y) (g y)$$

- For $\sigma = \forall t. (t \rightarrow t) \rightarrow (t \rightarrow t)$, this is **typable** as follows:

$$\begin{aligned} it &= \lambda h : \sigma \rightarrow (\sigma \rightarrow \sigma). \lambda g : \sigma \rightarrow \sigma. \lambda x : \sigma. \lambda y : \sigma. \\ &\quad x \sigma (\lambda z : \sigma. h z y) (g y) \\ &: (\sigma \rightarrow (\sigma \rightarrow \sigma)) \rightarrow (\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow \sigma) \end{aligned}$$

NB This would **not work** with a simple type $\sigma = (\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$!

332

Another example: parametric data type

- One can also handle the case of algebras which are defined **parametrically** with respect to a collection of data.
- For instance $List(D)$ is the **algebra of lists** whose elements belong to the set D .
- This algebra is equipped with the constructors

$$\text{nil} : List(D), \quad \text{cons} : D \times List(D) \rightarrow List(D) .$$

- One can define **iterative functions** over $List(D)$ and show that these functions can be represented in system F for a suitable embedding of the closed terms in system F.
- The sort $List(D)$ is coded by the type

$$\forall t. t \rightarrow (r \rightarrow t \rightarrow t) \rightarrow t$$

where r is a type variable, and generic elements in D are represented by (free) variables of type r .

333

Type constructors

- In the simply typed system one has to extend the language in order to have familiar type/logical operators such as product/conjunction and sum/disjunction.
- In system F, it is possible to give (**weak**) representations of common type operators.
- For instance, to define the **product** set:

$$(\sigma \times \tau) \equiv \forall t. (\sigma \rightarrow \tau \rightarrow t) \rightarrow t .$$

334

- Then **pairing** and **projection** terms can be defined as follows:

$$\langle M, N \rangle = \lambda t. \lambda f : \sigma \rightarrow \tau \rightarrow t. fMN$$

$$\pi_1 M = M\sigma(\lambda x : \sigma. \lambda y : \tau. x)$$

$$\pi_2 M = M\tau(\lambda x : \sigma. \lambda y : \tau. y) .$$

- One can check:

$$\pi_i \langle M_1, M_2 \rangle =_{\beta\eta} M_i \quad i = 1, 2$$

- Note however the following weakness: pairing is **not surjective**

$$\langle \pi_1 M, \pi_2 M \rangle \neq_{\beta\eta} M$$

335

More examples: sum and existential

In a similar vein, one can encode sum and (second-order) existential as follows:

$$\sigma + \tau \quad \text{as} \quad \forall t. (\sigma \rightarrow t) \rightarrow (\tau \rightarrow t) \rightarrow t$$

$$\exists t. \sigma \quad \text{as} \quad \forall s. (\forall t. \sigma \rightarrow s) \rightarrow s .$$

336

Towards strong normalisation

- We now move towards a proof of Girard's celebrated result: all terms typable in system F **strongly normalize**.
- The proof is based on the notion of **reducibility candidate**.
- This is an **abstraction** of the notion already considered for the strong normalisation of the simply typed λ -calculus (and RPO).

337

Type erasure

In order to make **notation lighter** we will work with untyped terms obtained from the **erasure** of well-typed terms.

- The **erasure** function er takes a typed term and returns an untyped λ -term.
- It is defined by induction on the structure of the term as follows:

$$\begin{aligned}er(x) &= x \\er(\lambda x : \sigma.M) &= \lambda x.er(M) \\er(MN) &= er(M)er(N) \\er(\lambda t.M) &= er(M) \\er(M\tau) &= er(M) .\end{aligned}$$

339

Key problem

- We want to adapt the **semantic** method already used in the simply typed case.
- What is the interpretation of $\sigma = \forall t.(t \rightarrow t)$?
- We have to build first a **universe** U of type-interpretations.
- Then we could require:

$$\llbracket \sigma \rrbracket = \{M \mid \forall X \in U \ \forall N \in X \ (MN \in X)\}$$

338

Erasure vs typed normalisation

- In system F we distinguish **two flavours of β -reduction**: the one involving a redex $(\lambda x : \sigma.M)N$ which we call simply β and the one involving a redex $(\lambda t.M)\sigma$ which we call β_t .
- Erasing type information we eliminate reductions of the shape β_t . However this does **not** affect the strong normalization property as shown in the following.

340

Proposition Let M be a well-typed term in system F. Then:

1. If $M \rightarrow_\beta N$ then $er(M) \rightarrow_\beta er(N)$.
2. If $M \rightarrow_{\beta_t} N$ then $er(M) \equiv er(N)$.
3. If M diverges then $er(M)$ diverges.

341

Reducibility candidate

We are now ready to define a universe of type interpretations, the so-called reducibility candidates.

- Let SN be the collection of untyped $\lambda\beta$ -strongly normalizable terms.
- A set X of λ -terms is a **reducibility candidate** if:
 1. $X \subseteq SN$.
 2. $Q_i \in SN, i = 1, \dots, n, n \geq 0$ implies $xQ_1, \dots, Q_n \in X$.
 3. $[Q/x]PQ_1, \dots, Q_n \in X$ and $Q \in SN$ implies $(\lambda x.P)QQ_1, \dots, Q_n \in X$.
- We denote with RC the collection of reducibility candidates and we abbreviate Q_1, \dots, Q_n with \vec{Q} .

NB We have made into a **definition** the **properties** of the interpretation of simple types.

343

Exercise

Prove the previous proposition.

342

Properties of reducibility candidates

1. The set SN is a reducibility candidate.
2. If $X \in RC$ then $X \neq \emptyset$.
3. The collection RC is closed under arbitrary intersections.
4. If $X, Y \in RC$ then the following set is a reducibility candidate:

$$X \rightarrow Y = \{M \mid \forall N \in X (MN \in Y)\} .$$

344

Proof of the properties

1. As in the simple case, we observe that $[Q/x]P\vec{Q} \in SN$ and $Q \in SN$ implies $(\lambda x.P)Q\vec{Q} \in SN$.
2. By definition $x \in X$.
3. Immediate.
4. Here we see the use of the saturation condition.

345

Soundness of the interpretation

Let η be a type environment η , and $x_1 : \sigma_1, \dots, x_n : \sigma_n \vdash M : \tau$ a derivable judgement.

If $P_i \in \llbracket \sigma_i \rrbracket \eta$, for $i = 1, \dots, n$ then $[P_1/x_1, \dots, P_n/x_n]er(M) \in \llbracket \tau \rrbracket \eta$.

347

Type interpretation

Given a type environment $\eta : Tvar \rightarrow RC$ we interpret types as follows:

$$\begin{aligned} \llbracket t \rrbracket \eta &= \eta(t) \\ \llbracket \sigma \rightarrow \tau \rrbracket \eta &= \llbracket \sigma \rrbracket \eta \rightarrow \llbracket \tau \rrbracket \eta \\ \llbracket \forall t. \sigma \rrbracket \eta &= \bigcap_{X \in RC} \llbracket \sigma \rrbracket \eta[X/t] . \end{aligned}$$

346

Proof of soundness (1/3)

- We abbreviate $[P_1/x_1, \dots, P_n/x_n]$ with $[\vec{P}/\vec{x}]$.
- We proceed by **induction on the length of the typing proof**.

(*asmp*) follows by definition.

(\rightarrow_I) We have to show

$$\lambda x. [\vec{P}/\vec{x}]er(M) \in \llbracket \sigma \rightarrow \tau \rrbracket \eta .$$

By inductive hypothesis we know

$$[\vec{P}/\vec{x}][P/x]er(M) \in \llbracket \tau \rrbracket \eta$$

for all $P \in \llbracket \sigma \rrbracket \eta$.

We conclude by using the properties of reducibility candidates.

(\rightarrow_E) By the definition of \rightarrow .

348

Proof of soundness (2/3)

(\forall_I) We have to show

$$[\vec{P}/\vec{x}]er(M) \in \bigcap_{X \in RC} \llbracket \tau \rrbracket \eta[X/t] .$$

By the side condition on the typing rule, we know:

$$\llbracket \sigma_i \rrbracket \eta = \llbracket \sigma_i \rrbracket \eta[X/t]$$

for an arbitrary $X \in RC$.

By inductive hypothesis:

$$[\vec{P}/\vec{x}]er(M) \in \llbracket \tau \rrbracket \eta[X/t]$$

for an arbitrary $X \in RC$.

349

Corollary: strong normalisation of system F

If $\Gamma \vdash M : \sigma$ in system F, then M is strongly normalizing.

351

Proof of soundness (3/3)

(\forall_E) We have to show:

$$[\vec{P}/\vec{x}]er(M) \in \llbracket \tau \rrbracket \eta[\llbracket \sigma \rrbracket \eta/t]$$

By inductive hypothesis:

$$[\vec{P}/\vec{x}]er(M) \in \bigcap_{X \in RC} \llbracket \tau \rrbracket \eta[X/t]$$

Choose $X = \llbracket \sigma \rrbracket \eta$.

350

Proof of strong normalisation

- We note that $\forall \sigma, \eta, x (x \in \llbracket \sigma \rrbracket \eta)$.
- Then we apply the lemma with $P_i \equiv x_i$, and derive that

$$er(M) \in \llbracket \tau \rrbracket \eta \subseteq SN$$

- By proposition (erasure vs. typed) we conclude that M is strongly normalizing.

352

Exercise (alternative definition of reducibility candidate)

- We say that a term is **neutral** if it does not start with a λ -abstraction.
- The collection RC' is given by the sets X of strongly normalizing terms satisfying the following conditions:
 1. $M \in X$ and $M \rightarrow_{\beta} M'$ implies $M' \in X$.
 2. M neutral and $\forall M' (M \rightarrow_{\beta} M' \supset M' \in X)$ implies $M \in X$.
- Carry on the strong normalization proof using the collection RC' .

353

Recommended reading

Jean-Yves Girard et al. Proofs and Types. Cambridge University Press. Chapters 11 and 14.

355

Summary

- The introduction of second-order quantification **preserves the standard properties** of the simply typed calculus: subject reduction, strong normalisation, confluence ...
- While **increasing the expressivity** in a very significant way: one can encode inductive data types and iterative functions.
- There is one catch however, **type inference becomes undecidable** which is one reason why ML-like programming languages adopt a weaker/predicative form of polymorphism.
- When extended with first-order quantification, system F is the backbone of a higher-order constructive logic (the so called **calculus of constructions** on which the COQ system is built).

354

Conclusion

356

Two models of functional computation

First-order Term Rewriting Systems.

Higher-order (Typed) λ -calculi.

357

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.

359

Recurring themes

Confluence local confluence, critical pairs, completion, parallel reduction.

Termination interpretation method, recursive path-ordering, reducibility candidates.

Expressive power PTIME, primitive recursive, inductive data types and iterative functions, partial recursive functions.

Constraint solving unification, type inference.

358

Exercise (revision)

Attach a name and a concept and/or a theorem to the following pictures:

360

