Handout for lecture 9:

Term Rewriting

1. Motivation

2

Beyond predicate logic

Recall: in the previous lecture we discussed predicate logic

$$(\exists x[\mathtt{S}(x) \wedge \forall y[\mathtt{L}(y) \rightarrow \mathtt{A}(x,y)]]) \wedge$$
$$(\forall x[(\mathtt{L}(x) \wedge \mathtt{B}(x)) \rightarrow \neg\exists y[\mathtt{S}(y) \wedge \mathtt{A}(y,x)]]) \rightarrow$$
$$\neg\exists x[\mathtt{L}(x) \wedge \mathtt{B}(x)]$$

We studied how to bring such formulas into a kind of CNF, and to solve them using resolution.

In practice: we often want an *equality* operator when studying logic with quantifiers.

$$\forall x[\forall y[\mathtt{suc}(x) \mathrel{\mathtt{=}} \mathtt{suc}(y) \rightarrow x \mathrel{\mathtt{=}} y]]$$

$$\exists x[\exists y[x \neq y \wedge \mathtt{Favourite}(x) \mathrel{\mathtt{=}} \mathtt{AR} \wedge \mathtt{Favourite}(y) \mathrel{\mathtt{=}} \mathtt{AR}]]$$

This is called **equational logic**, and will be discussed in some detail later in the course.

As a prerequisite, we will need to understand **term rewriting**.

3

Program analysis

We find another motivation in the area of program analysis.

Recall: analysing a for loop

```
    $a := 0$;
    for $i := 1$ to $m$ do $a := a + k$
```

After running this piece of code, we have $a = m * k$, by unsatisfiability of the following formula:

$$\bigwedge_{j=1}^{n} \neg a_{0,j} \quad \wedge \quad \bigwedge_{i=0}^{m-1} \mathrm{plus}(\vec{a}_i, \vec{k}, \vec{a}_{i+1}) \quad \wedge \neg\mathrm{mul}([\vec{m}], \vec{k}, \vec{a}_m)$$

Challenge: analysing functional programs, such as the following OCaml program:

```
let rec conc xs ys =
    match xs with
        | [] ⇒ ys
        | h :: t ⇒ h :: (conc t ys)
;;
```

```
let rec rev xs =
    match xs with
        | [] ⇒ []
        | h :: t ⇒ conc (rev t) [h]
;;
```

Here, too, term rewriting is a promising avenue for deriving relevant properties.

2. Term rewriting

2.1 Definition

4

Defining terms

Given: a set $\mathcal{V}ar$ of variables: $\{x,\ y,\ z,\ \dots\}$

Given: a set Σ of **function symbols**: $\{\mathtt{f},\ \mathtt{g},\ \mathtt{h},\ \dots\}$

Given: an **arity** function, that maps each function symbol to a non-negative integer

- every variable is a term
- if $\mathtt{f} \in \Sigma$ and $\mathit{arity}(\mathtt{f}) = n$ and $s_1, \dots, s_n$ are terms,
  then $\mathtt{f}(s_1, \dots, s_n)$ is a term

5

Defining terms

Example:

$$\Sigma = \{\mathtt{0},\ \mathtt{s},\ \mathtt{add},\ \mathtt{mul}\}$$

$$\begin{array}{rcl} \mathit{arity}(\mathtt{0}) & = & 0 \\ \mathit{arity}(\mathtt{s}) & = & 1 \\ \mathit{arity}(\mathtt{add}) & = & 2 \\ \mathit{arity}(\mathtt{mul}) & = & 2 \end{array}$$

Terms are for instance:

- $\mathtt{0}$, $\mathtt{s}(\mathtt{0})$, $\mathtt{s}(\mathtt{s}(\mathtt{0}))$, $\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{0})))$, …
- $\mathtt{s}(\mathtt{s}(\mathtt{s}(x)))$
- $\mathtt{add}(\mathtt{s}(\mathtt{0}), \mathtt{add}(y, \mathtt{s}(\mathtt{add}(\mathtt{0}, x))))$
- $\mathtt{mul}(\mathtt{add}(\mathtt{0}, \mathtt{0}), \mathtt{s}(y))$

6

Class exercise

Design a *signature* (Σ, arity) that contains lists of natural numbers.

Alter the signature to handle lists of *integers*.

Example answer:

$\Sigma = \{\texttt{0}, \texttt{s}, \texttt{nil}, \texttt{cons}\}$ for the first question, extended with $\{\texttt{min}\}$ for the second question.

$$\begin{array}{rcl} \mathit{arity}(\texttt{0}) & = & 0 \\ \mathit{arity}(\texttt{s}) & = & 1 \\ \mathit{arity}(\texttt{nil}) & = & 0 \\ \mathit{arity}(\texttt{cons}) & = & 2 \\ \mathit{arity}(\texttt{min}) & = & 1 \end{array}$$

The idea is that for instance $\texttt{s}(\texttt{s}(\texttt{0}))$ represents 2, and $\texttt{min}(\texttt{s}(\texttt{s}(\texttt{0})))$ represents -2. There are also meaningless terms like $\texttt{min}(\texttt{0})$ or $\texttt{s}(\texttt{cons}(\texttt{nil}, \texttt{min}(\texttt{0})))$, but their existence is usually not a problem for reasoning. In cases where limiting the set of possible terms becomes relevant, we can always add a type restriction, but for now we will not do so.

7
Rules

A rule is a pair of terms, denoted $\ell \Rightarrow r$, such that:

- ℓ is not a variable
- all variables in r occur also in ℓ

Examples:

- $\texttt{add}(x, \texttt{0}) \Rightarrow x$
- $\texttt{add}(x, \texttt{s}(y)) \Rightarrow \texttt{s}(\texttt{add}(x, y))$
- $\texttt{min}(\texttt{min}(x)) \Rightarrow x$

Non-examples

- $x \Rightarrow \texttt{min}(\texttt{min}(x))$ (but $\texttt{min}(\texttt{min}(x)) \Rightarrow x$ is fine!)
- $\texttt{mul}(x, \texttt{0}) \Rightarrow \texttt{mul}(y, \texttt{0})$

8
Reducing terms

Intuition: We want to use rules to rewrite terms. For example: $\texttt{min}(\texttt{add}(\texttt{s}(\texttt{0}), \texttt{s}(\texttt{s}(\texttt{0}))))$ *rewrites* to $\texttt{min}(\texttt{s}(\texttt{add}(\texttt{s}(\texttt{0}), \texttt{s}(\texttt{0}))))$ using the rule $\texttt{add}(x, \texttt{s}(y)) \Rightarrow \texttt{s}(\texttt{add}(x, y))$.

Ingredient: a **substitution** is a function from variables to terms. As we saw in the lecture on predicate logic, a substitution γ is *applied* on a term s, notation $s\gamma$, by replacing all variables x in the term by $\gamma(x)$.

$$\begin{array}{rcl} x\gamma & = & \gamma(x) \\ \texttt{f}(s_1, \ldots, s_n)\gamma & = & \texttt{f}(s_1\gamma, \ldots, s_n\gamma) \end{array}$$

Given: a set of rules $\mathcal{R}$

Define:

- $\ell\gamma \Rightarrow_{\mathcal{R}} r\gamma$ for all $\ell \Rightarrow r \in \mathcal{R}$, all γ
- $\mathtt{f}(s_1, \dots, s_i, \dots, s_n) \Rightarrow_{\mathcal{R}} \mathtt{f}(s_1, \dots, t_i, \dots, s_n)$ if $s_i \Rightarrow_{\mathcal{R}} t_i$

Example:

$$\begin{array}{ccc} \mathtt{min}(\underbrace{\mathtt{add}(\mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{s}(\mathtt{0})))}) & \Rightarrow_{\mathcal{R}} & \mathtt{min}(\mathtt{s}(\mathtt{add}(\mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{0})))) \\ \mathtt{add}(\ x\ , \mathtt{s}(\ y\ )) & \Rightarrow & \mathtt{s}(\mathtt{add}(\ x\ ,\ y\ )) \end{array}$$

We use a substitution γ with both $\gamma(x) = \mathtt{s}(\mathtt{0})$ and $\gamma(y) = \mathtt{s}(\mathtt{0})$.

9

Class exercise

Let

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathtt{add}(x, \mathtt{s}(y)) & \Rightarrow & \mathtt{s}(\mathtt{add}(x, y)) \\ \mathtt{add}(x, \mathtt{p}(y)) & \Rightarrow & \mathtt{p}(\mathtt{add}(x, y)) \\ \mathtt{add}(x, \mathtt{0}) & \Rightarrow & x \\ \mathtt{s}(\mathtt{p}(x)) & \Rightarrow & x \\ \mathtt{p}(\mathtt{s}(x)) & \Rightarrow & x \end{array} \right\}$$

Question: what can we reduce the following term to?

$$\mathtt{s}(\mathtt{add}(\mathtt{0}, \mathtt{p}(\mathtt{s}(\mathtt{0}))))$$

Answer: two options! (Rewriting is not necessarily deterministic.)

- $\mathtt{s}(\mathtt{add}(\mathtt{0}, \mathtt{0}))$
- $\mathtt{s}(\mathtt{p}(\mathtt{add}(\mathtt{0}, \mathtt{s}(\mathtt{0}))))$

10

Normal form

We are often interested in a reduction to **normal form**: a term that cannot be reduced anymore.

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathtt{add}(x, \mathtt{s}(y)) & \Rightarrow & \mathtt{s}(\mathtt{add}(x, y)) \\ \mathtt{add}(x, \mathtt{p}(y)) & \Rightarrow & \mathtt{p}(\mathtt{add}(x, y)) \\ \mathtt{add}(x, \mathtt{0}) & \Rightarrow & x \\ \mathtt{s}(\mathtt{p}(x)) & \Rightarrow & x \\ \mathtt{p}(\mathtt{s}(x)) & \Rightarrow & x \end{array} \right\}$$

Example:

$$\begin{array}{rcl} \mathtt{s}(\mathtt{add}(\mathtt{0}, \underline{\mathtt{p}(\mathtt{s}(\mathtt{0}))})) & \Rightarrow_{\mathcal{R}} & \mathtt{s}(\mathtt{add}(\mathtt{0}, \underline{\mathtt{0}})) \\ & \Rightarrow_{\mathcal{R}} & \mathtt{s}(\mathtt{0}) \end{array}$$

Exercise: find a *different* reduction to normal form for $\mathtt{s}(\mathtt{add}(\mathtt{0}, \mathtt{p}(\mathtt{s}(\mathtt{0}))))$

Possible answer:

$$\begin{array}{rcl} \mathtt{s}(\mathtt{add}(\mathtt{0}, \mathtt{p}(\mathtt{s}(\mathtt{0})))) & \Rightarrow_{\mathcal{R}} & \mathtt{s}(\mathtt{p}(\mathtt{add}(\mathtt{0}, \mathtt{s}(\mathtt{0})))) \\ & \Rightarrow_{\mathcal{R}} & \mathtt{add}(\mathtt{0}, \mathtt{s}(\mathtt{0})) \\ & \Rightarrow_{\mathcal{R}} & \mathtt{s}(\mathtt{add}(\mathtt{0}, \mathtt{0})) \\ & \Rightarrow_{\mathcal{R}} & \mathtt{s}(\mathtt{0}) \end{array}$$

After the first step, you could alternatively reduce the subterm $\mathtt{add}(\mathtt{0}, \mathtt{s}(\mathtt{0}))$ first. This again leads to multiple choices.

2.2 Uses

11

Peek-ahead: equational logic

Rules can be seen as **oriented equations**.

The rules

$$\begin{array}{rcl} \mathtt{add}(\mathtt{0}, y) & \Rightarrow & y \\ \mathtt{add}(\mathtt{s}(x), y) & \Rightarrow & \mathtt{s}(\mathtt{add}(x, y)) \end{array}$$

define the equations:

$$\begin{array}{rrcl} \forall y. & \mathtt{add}(\mathtt{0}, y) & = & y \\ \forall x \forall y. & \mathtt{add}(\mathtt{s}(x), y) & = & \mathtt{s}(\mathtt{add}(x, y)) \end{array}$$

We can see that for any model $(M, [\![\cdot]\!]_\alpha)$ that makes the given equalities true: if $s \Rightarrow_{\mathcal{R}} t$ then $[\![s]\!]_\alpha = [\![t]\!]_\alpha$ for all α. (By induction on the size of s.)

Hence, if $s \Leftrightarrow^*_{\mathcal{R}} t$, this proves that $s = t$ follows from equations in $\mathcal{R}$.

So, the reduction we saw before is a **proof** that $2 + 2 = 4$!

This will be properly defined and discussed in the lecture on equational logic.

12

Term rewriting and Prover9

We can also reason about reduction in Prover9:

```
formulas(assumptions).
R(a(0,x),x).
R(a(s(x),y),s(a(x,y))).
R(x,y) -> R(a(x,z),a(y,z)).
R(x,y) -> R(a(z,x),a(z,y)).
R(x,y) -> R(s(x),s(y)).
RR(x,x).
(RR(x,y) & R(y,z)) -> RR(x,z).
end_of_list.
formulas(goals).
```

```
RR(a(s(s(0)),s(s(0))),s(s(s(s(0))))).
end_of_list.
```

Here, we let:

`a` = plus operator
`R` = single rewrite step
`RR` = zero or more rewrite steps

---

13 Functional programming

Computation in a term rewriting system typically means: reduction of a term to normal form.

Rewriting to normal form is the basic formalism in several kinds of computation.

In particular, it is the underlying formalism for both semantics and implementation of **functional programming**, in which the function definitions are interpreted as rewrite rules.

For example, let us consider the code we saw before.

```
let rec conc xs ys =
    match xs with
        | [] ⇒ ys
        | h :: t ⇒ h :: (conc t ys)
;;
```

```
let rec rev xs =
    match xs with
        | [] ⇒ []
        | h :: t ⇒ conc (rev t) [h]
;;
```

Reformulated in Haskell, this same pair of functios looks as follows:

```
rev nil      = nil
rev (a:x)    = conc (rev x (a:nil))
conc nil x   = x
conc (a:x) y = a:(conc x y)
```

Here `a`, `x` and `y` are variables, and = corresponds to ⇒ in rewrite rules.

Hence, this corresponds to the following term rewriting system (still denoting the list constructor $\mathtt{:}$ in right-associative infix notation):

$$\begin{array}{rcl}
\mathtt{rev}(\mathtt{nil}) & \Rightarrow & \mathtt{nil} \\
\mathtt{rev}(a\ \mathtt{:}\ \mathtt{nil}) & \Rightarrow & \mathtt{conc}(\mathtt{rev}(x, a\ \mathtt{:}\ \mathtt{nil})) \\
\mathtt{conc}(\mathtt{nil}, x) & \Rightarrow & x \\
\mathtt{conc}(a\ \mathtt{:}\ x, y) & \Rightarrow & a\ \mathtt{:}\ \mathtt{conc}(x, y)
\end{array}$$

Then we have a reduction to normal form:

$$\begin{array}{ll}
\mathtt{rev}(x\ \mathtt{:}\ y\ \mathtt{:}\ \mathtt{nil}) & \Rightarrow \\
\mathtt{conc}(\mathtt{rev}(y\ \mathtt{:}\ \mathtt{nil}), x\ \mathtt{:}\ \mathtt{nil}) & \Rightarrow \\
\mathtt{conc}(\mathtt{conc}(\mathtt{rev}(\mathtt{nil}), y\ \mathtt{:}\ \mathtt{nil}), x\ \mathtt{:}\ \mathtt{nil}) & \Rightarrow \\
\mathtt{conc}(\mathtt{conc}(\mathtt{nil}, y\ \mathtt{:}\ \mathtt{nil}), x\ \mathtt{:}\ \mathtt{nil}) & \Rightarrow \\
\mathtt{conc}(y\ \mathtt{:}\ \mathtt{nil}, x\ \mathtt{:}\ \mathtt{nil}) & \Rightarrow \\
y\ \mathtt{:}\ \mathtt{conc}(\mathtt{nil}, x\ \mathtt{:}\ \mathtt{nil}) & \Rightarrow \\
y\ \mathtt{:}\ x\ \mathtt{:}\ \mathtt{nil} &
\end{array}$$

This corresponds to the computation done by (for instance) Ocaml.

2.3 Properties of term rewriting systems

14

Some nice properties

- $\mathcal{R}$ is **weakly normalising** (WN):

  every term has a normal form

- $\mathcal{R}$ is **terminating** (= strongly normalising, SN):

  no infinite sequence of terms $t_1, t_2, t_3, \ldots$ exists such that $t_i \Rightarrow_{\mathcal{R}} t_{i+1}$ for all i

- $\mathcal{R}$ is **confluent** (= Church-Rosser, CR):

  if $s \Rightarrow^*_{\mathcal{R}} t$ and $s \Rightarrow^*_{\mathcal{R}} q$ then a term u exists satisfying $t \Rightarrow^*_{\mathcal{R}} u$ and $q \Rightarrow^*_{\mathcal{R}} u$

- $\mathcal{R}$ is **locally confluent** (= weak Church-Rosser, WCR):

  if $s \Rightarrow_{\mathcal{R}} t$ and $s \Rightarrow_{\mathcal{R}} q$ then a term u exists satisfying $t \Rightarrow^*_{\mathcal{R}} u$ and $q \Rightarrow^*_{\mathcal{R}} u$

Here $\Rightarrow^*_{\mathcal{R}}$ is the reflexive transitive closure of $\Rightarrow_{\mathcal{R}}$, i.e., $t \Rightarrow^*_{\mathcal{R}} u$ if and only if t can be rewritten to u in zero or more steps.

15

Strong normalisation implies weak normalisation

Theorem
If a TRS is terminating, then every term has at least one normal form.

Proof: rewriting as long as possible does not go on forever due to termination.
So it ends in a normal form. □

The converse is not true: the TRS over the two constants `a`,`b` consisting of the two rules `a` $\Rightarrow$ `a` and `a` $\Rightarrow$ `b` is weakly normalising since the two terms `a` and `b` both have `b` as a normal form, but it is not terminating due to:

$$\mathtt{a} \Rightarrow_{\mathcal{R}} \mathtt{a} \Rightarrow_{\mathcal{R}} \mathtt{a} \Rightarrow_{\mathcal{R}} \mathtt{a} \Rightarrow_{\mathcal{R}} \cdots$$

16

Confluence implies uniqueness of normal forms

Theorem

If a TRS is confluent, then every term has at most one normal form.

Proof: Assume t has two normal forms u, u'.
Then by confluence there is a v such that $u \Rightarrow^*_{\mathcal{R}} v$ and $u' \Rightarrow^*_{\mathcal{R}} v$.
Since u, u' are normal forms we have $u = v = u'$. $\square$

17

Termination + confluence implies existence of unique normal forms

We conclude:

Theorem

If a TRS is terminating and confluent, then every term has exactly one normal form.

This normal form can be found just by rewriting the term until no further rewriting step is possible.

This is a very useful combination! After all, it allows us to essentially define a computable function using a term rewriting system: to calculate the result of the function on a given input, we simply reduce the term to normal form in whichever way we choose. In a later lecture, we will also discuss why this combination is so useful in the context of equational logic.

3. Termination

3.1 Undecidability

Note: this subsection is included for those wanting to know more, and will not be on the exam. If you want to focus on the material that was in the lecture, please feel free to skip to the section on Reduction Orders.

18

Termination of term rewriting is **undecidable**, i.e., there is no algorithm that can decide for every finite TRS whether it is terminating.

This can be proved by transforming an arbitrary Turing machine to a TRS and proving that the TRS is terminating if and only if the Turing machine is halting from every initial configuration.

A Turing machine (Q, S, δ) consists of

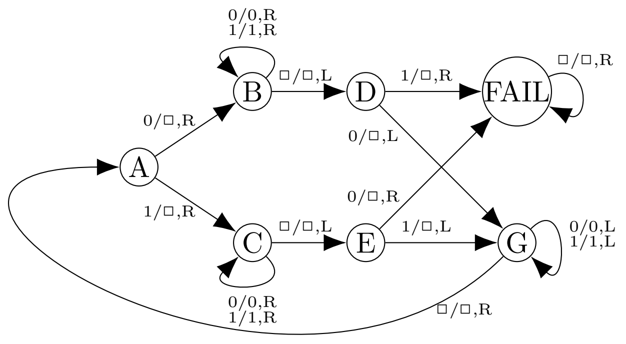
- a finite set Q of machine states
- a finite set S of tape symbols, including $\Box \in S$ representing the blank symbol
- the transition function $\delta : Q \times S \to Q \times S \times \{L, R\}$

Here $\delta(q, s) = (q', s', L)$ means that if the machine is in state q and reads s, this s is replaced by s', the machine shifts to the left, and the new machine state is q'.

Similar for $\delta(q, s) = (q', s', R)$: then the machine shifts to the right.

19

Example:



Here:

- $Q = \{$ A,B,C,D,E,FAIL,G $\}$

- $S = \{0, 1, \Box\}$
- $\delta(q, s) = (q', s', d)$ if there is an arrow marked $s/s', d$ from q to q'

20

This Turing machine behaviour can be simulated by a TRS: for a Turing machine $M = (Q, S, \delta)$ we define a TRS over the signature $Q \cup S \cup \{$ `:`, `b` $\}$ where

- `:` and symbols from Q have arity 2
- `b` and symbols from S have arity 0

The configuration with tape

$$\cdots \Box\Box\Box s'_m s'_{m-1} \cdots s'_1 s_1 s_2 \cdots s_{n-1} s_n \Box\Box\Box \cdots$$

in which the Turing machine is in state q and reads symbol s_1, is represented by the term

$$\mathtt{q}(\mathtt{s}_1\texttt{'} \mathbin{\texttt{:}} \mathtt{s}_2\texttt{'} \mathbin{\texttt{:}} \cdots \mathbin{\texttt{:}} \mathtt{s}_m\texttt{'} \mathbin{\texttt{:}} \mathtt{b}\ ,\ \mathtt{s}_1 \mathbin{\texttt{:}} \mathtt{s}_2 \mathbin{\texttt{:}} \cdots\ \mathtt{s}_n \mathbin{\texttt{:}} \mathtt{b})$$

Hence, we consider two lists of tape symbols, ending in `b` which represents an infinite sequence of blanks.

21

The rules for this TRS are:

$$\mathtt{q}(x, \mathtt{s} \mathbin{\texttt{:}} y) \to \mathtt{q'}(\mathtt{s'} \mathbin{\texttt{:}} x, y)$$

for all $(q, s) \in Q \times S$ with $\delta(q, s) = (q', s', R)$, and

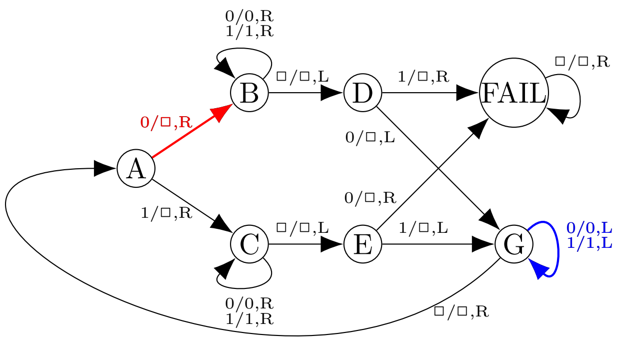
$$\mathtt{q}(c \mathbin{\texttt{:}} x, \mathtt{s} \mathbin{\texttt{:}} y) \to \mathtt{q'}(x, c \mathbin{\texttt{:}} \mathtt{s'} \mathbin{\texttt{:}} y)$$

for all $(q, s) \in Q \times S$ with $\delta(q, s) = (q', s', L)$, for all $t \in S$

and some extra rules representing that `b` consists of infinitely many blank symbols.

22

Example:



Step from A to B:

$$\mathtt{A}(x, \mathtt{0} : y) \rightarrow \mathtt{B}(\square : x, y)$$

Steps from G to G:

$$\begin{aligned} \mathtt{G}(c : x, \mathtt{0} : y) &\rightarrow \mathtt{G}(x, c : \mathtt{0} : y) \\ \mathtt{G}(c : x, \mathtt{1} : y) &\rightarrow \mathtt{G}(x, c : \mathtt{1} : y) \\ \mathtt{G}(\mathtt{b}, \mathtt{1} : y) &\rightarrow \mathtt{G}(\mathtt{b}, \square : \mathtt{1} : y) \end{aligned}$$

23

Theorem
M halts on every configuration if and only if $R(M)$ is terminating.

Consequence: TRS termination is undecidable.

Despite this, in many special cases it is possible to prove termination of a TRS.

3.2 Reduction orders

24

Well-founded ordering

Idea: find a **well-founded ordering** $\succ$ and prove that $s \succ t$ whenever $s \Rightarrow_{\mathcal{R}} t$.

(Because then any infinite reduction $s_1 \Rightarrow_{\mathcal{R}} s_2 \Rightarrow_{\mathcal{R}} s_3 \Rightarrow_{\mathcal{R}} \dots$ is an infinite decreasing sequence $s_1 \succ s_2 \succ s_3 \succ \dots$, contradicting well-foundedness.)

There are many ways to find such an ordering! In this course, we will consider two of the most popular.

25

A finite definition for infinitely many terms?

Difficulty: how to prove $s \succ t$ whenever $s \Rightarrow_{\mathcal{R}} t$? There are infinitely many terms and possible reductions.

$$\begin{array}{rcl} \mathtt{add}(\mathtt{0}, y) & \Rightarrow & y \\ \mathtt{add}(\mathtt{s}(x), y) & \Rightarrow & \mathtt{s}(\mathtt{add}(x, y)) \end{array}$$

Needed: $\mathtt{add}(\mathtt{0}, \mathtt{0}) \succ \mathtt{0}$, $\mathtt{add}(\mathtt{0}, \mathtt{add}(x, y)) \succ \mathtt{add}(x, y), \dots$

Solution: it suffices to orient the **rules** if we have a well-founded ordering $\succ$ with:

- if $s \succ t$ then $s\sigma \succ t\sigma$ for all substitutions σ
  (we say: $\succ$ is **stable**)

- if $s \succ t$ then $\mathtt{f}(\dots, s, \dots) \succ \mathtt{f}(\dots, t, \dots)$ for all $\mathtt{f}$
  (we say: $\succ$ is **monotonic**)

Such an ordering is called a **reduction ordering**.

4. LPO

4.1 The Lexicographic Path Ordering: definition

26

LPO

A very powerful technique, with many extensions and variations, is the **lexicographic path ordering**.

Let $\rhd$ be a **total, well-founded ordering** on the function symbols.

We define: $\mathtt{f}(s_1, \dots, s_n) \succ_{\mathrm{LPO}} t$ if one of the following holds:

(sub) $s_i \succeq_{\mathrm{LPO}} t$ for some $i \in \{1, \dots, n\}$
(that is, $s_i \succ t$ or $s_i = t$)

(copy) $t = \mathtt{g}(t_1, \dots, t_m)$ and $\mathtt{f} \rhd \mathtt{g}$ and $\mathtt{f}(s_1, \dots, s_n) \succ_{\mathrm{LPO}} t_j$ for all $j \in \{1, \dots, m\}$

(lex) $t = \mathtt{f}(t_1, \dots, t_n)$ and $\mathtt{f}(s_1, \dots, s_n) \succ_{\mathrm{LPO}} t_i$ for all $i \in \{1, \dots, n\}$,
and $[s_1, \dots, s_n](\succ_{\mathrm{LPO}})_{\mathrm{lex}}[t_1, \dots, t_n]$;
that is, there is some $i \in \{1, \dots, n\}$ such that:

- $s_j = t_j$ for $j \in \{1, \dots, i-1\}$
- $s_i \succ_{\mathrm{LPO}} t_i$

27

LPO example

$$\begin{array}{rcl}
\mathtt{add}(\mathtt{0}, y) & \Rightarrow & y \\
\mathtt{add}(\mathtt{s}(x), y) & \Rightarrow & \mathtt{s}(\mathtt{add}(x, y)) \\
\mathtt{mul}(\mathtt{0}, y) & \Rightarrow & \mathtt{0} \\
\mathtt{mul}(\mathtt{s}(x), y) & \Rightarrow & \mathtt{add}(y, \mathtt{mul}(x, y))
\end{array}$$

We choose: $\mathtt{mul} \rhd \mathtt{add} \rhd \mathtt{s} \rhd \mathtt{0}$

We orient the last rule as follows:

A.	$\mathtt{mul}(\mathtt{s}(x), y)$	$\succ_{\mathrm{LPO}}$	$\mathtt{add}(y, \mathtt{mul}(x, y))$	by **(copy)**, B, D
B.	$\mathtt{mul}(\mathtt{s}(x), y)$	$\succ_{\mathrm{LPO}}$	y	by **(sub)**, C
C.	y	$\succeq_{\mathrm{LPO}}$	y	by definition
D.	$\mathtt{mul}(\mathtt{s}(x), y)$	$\succ_{\mathrm{LPO}}$	$\mathtt{mul}(x, y)$	by **(lex)**, E, B, F
E.	$\mathtt{mul}(\mathtt{s}(x), y)$	$\succ_{\mathrm{LPO}}$	x	by **(sub)**, F
F.	$\mathtt{s}(x)$	$\succ_{\mathrm{LPO}}$	x	by **(sub)**, G
G.	x	$\succeq_{\mathrm{LPO}}$	x	by definition

28

Soundness of LPO

Theorem

If $\ell \succ_{\text{LPO}} r$ for all rules in $\mathcal{R}$, then the TRS with rules $\mathcal{R}$ is terminating.

Proof. $\succ_{\text{LPO}}$ is:

- stable: if $s \succ_{\text{LPO}} t$ then $s\sigma \succ_{\text{LPO}} t\sigma$
  (by a simple induction on the definition: if $x \succeq_{\text{LPO}} t$ then $t = x$, so $x\sigma = t\sigma$ too)
- monotonic: if $s \succ_{\text{LPO}} t$ then $\mathtt{f}(\dots, s, \dots) \succ_{\text{LPO}} \mathtt{f}(\dots, t, \dots)$
  (by the **(lex)** rule)
- well-founded: there is no infinite decreasing sequence
  (this proof is a bit more complex; essentially, we prove that $\mathtt{f}(s_1, \dots, s_n)$ is well-founded (i.e., there is no infinite decreasing sequence starting in it) if all s_i are, by induction first on $\rhd$, second on $[s_1, \dots, s_n]$ ordered by the lexicographic extension of $\succ_{\text{LPO}}$) $\square$

29

Ackermann example

The lexicograph path ordering is a powerful technique, that allows you to prove termination of quite sophisticated systems!

Consider the **Ackermann function:**

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

This may look simple, but $A(4, 2)$ turns out to be an integer of 19,729 decimal digits.

It is the simplest function that is not **primitive recursive**, and grows much harder than any primitive recursive function.

We can express this function as a TRS:

$$\begin{array}{rcl} \mathtt{A}(\mathtt{0}, x) & \Rightarrow & \mathtt{s}(x) \\ \mathtt{A}(\mathtt{s}(x), \mathtt{0}) & \Rightarrow & \mathtt{A}(x, \mathtt{s}(\mathtt{0})) \\ \mathtt{A}(\mathtt{s}(x), \mathtt{s}(y)) & \Rightarrow & \mathtt{A}(x, \mathtt{A}(\mathtt{s}(x), y)) \end{array}$$

We prove termination of the Ackermann function by a lexicographic path ordering with $\mathtt{A} \rhd \mathtt{s}$.

For example, for the last rule:

A.	$\mathtt{A}(\mathtt{s}(x), \mathtt{s}(y))$	$\succ_{\text{LPO}}$	$\mathtt{A}(x, \mathtt{A}(\mathtt{s}(x), y))$	by **(lex)**, B, C, D
B.	$\mathtt{A}(\mathtt{s}(x), \mathtt{s}(y))$	$\succ_{\text{LPO}}$	x	by **(sub)**, D
C.	$\mathtt{A}(\mathtt{s}(x), \mathtt{s}(y))$	$\succ_{\text{LPO}}$	$\mathtt{A}(\mathtt{s}(x), y)$	by **(lex)**, E, F, G
D.	$\mathtt{s}(x)$	$\succ_{\text{LPO}}$	x	by **(sub)**, $x \succeq_{\text{LPO}} x$
E.	$\mathtt{A}(\mathtt{s}(x), \mathtt{s}(y))$	$\succ_{\text{LPO}}$	$\mathtt{s}(x)$	by **(sub)**, $\mathtt{s}(x) \succeq_{\text{LPO}} \mathtt{s}(x)$
F.	$\mathtt{A}(\mathtt{s}(x), \mathtt{s}(y))$	$\succ_{\text{LPO}}$	y	by **(sub)**, G
G.	$\mathtt{s}(y)$	$\succ_{\text{LPO}}$	y	by **(sub)**, $y \succeq_{\text{LPO}} y$

Hence, this TRS is terminating. As it is also confluent (for which we will see methods in a later lecture), this TRS **defines** the Ackermann function.

While termination theoretically holds, the normal form of $\mathtt{A}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{0})))), \mathtt{s}(\mathtt{s}(\mathtt{0})))$ is a term containing N symbols $\mathtt{s}$, for N being a number of 19,729 decimal digits (as observed before). Hence, this normal form does not fit in all computer memory of the world.

4.2 Automating LPO

30

An automatic solution?

For large term rewriting systems it is not always easy to test if the rules can be oriented by LPO. Certainly not if $\rhd$ is not known.

In fact, we have a problem of the form:

> Do there exist a symbol ordering $\rhd$,
> and a sequence of proof steps
> such that the given inequalities $s \succ_{\text{LPO}} t$ all hold?

This is a SAT problem!

Idea:

- encode the ordering using variables $\langle \mathtt{f} \rhd \mathtt{g} \rangle$ for all $\mathtt{f}, \mathtt{g}$
- require: $\neg\langle \mathtt{f} \rhd \mathtt{f} \rangle$ for all $\mathtt{f}$
- require: $\langle \mathtt{f} \rhd \mathtt{g} \rangle \leftrightarrow \neg\langle \mathtt{g} \rhd \mathtt{f} \rangle$ for all $\mathtt{f}, \mathtt{g}$ with $\mathtt{f} \neq \mathtt{g}$
- require: $\langle \mathtt{f} \rhd \mathtt{g} \rangle \wedge \langle \mathtt{g} \rhd \mathtt{h} \rangle \rightarrow \langle \mathtt{f} \rhd \mathtt{h} \rangle$ for all $\mathtt{f}, \mathtt{g}, \mathtt{h}$
- encode the requirements for each inequality $\ell \succ_{\text{LPO}} r$!

But how to do that last part?

31

Automation as one big formula

$$\mathtt{f}(\mathtt{g}(x)) \succ_{\mathrm{LPO}} \mathtt{h}(\mathtt{f}(x))$$

is equivalent to:

$$\begin{array}{ll} \mathtt{g}(x) \succeq_{\mathrm{LPO}} \mathtt{h}(\mathtt{f}(x)) & \vee \\ (\ \langle \mathtt{f} \rhd \mathtt{h}\rangle \wedge \mathtt{f}(\mathtt{g}(x)) \succ_{\mathrm{LPO}} \mathtt{f}(x)\ ) & \end{array}$$

is equivalent to:

$$\begin{array}{ll} x \succeq_{\mathrm{LPO}} \mathtt{h}(\mathtt{f}(x)) & \vee \\ (\ \langle \mathtt{g} \rhd \mathtt{h}\rangle \wedge \mathtt{g}(x) \succ_{\mathrm{LPO}} \mathtt{f}(x)\ ) & \vee \\ (\ \langle \mathtt{f} \rhd \mathtt{h}\rangle \wedge (\ \mathtt{g}(x) \succeq_{\mathrm{LPO}} \mathtt{f}(x) \vee & \\ \qquad (\ \mathtt{f}(\mathtt{g}(x)) \succ_{\mathrm{LPO}} x \wedge \mathtt{g}(x) \succ_{\mathrm{LPO}} x\ )\ )\ ) & \end{array}$$

is equivalent to:

$$\begin{array}{ll} \bot & \vee \\ (\ \langle \mathtt{g} \rhd \mathtt{h}\rangle \wedge \qquad (x \succeq_{\mathrm{LPO}} \mathtt{f}(x) \vee (\langle \mathtt{g} \rhd \mathtt{f}\rangle \wedge \mathtt{g}(x) \succ_{\mathrm{LPO}} x)\ ) & \vee \\ (\ \langle \mathtt{f} \rhd \mathtt{h}\rangle \wedge \ (\quad (x \succeq_{\mathrm{LPO}} \mathtt{f}(x) \vee (\langle \mathtt{g} \rhd \mathtt{f}\rangle \wedge \mathtt{g}(x) \succ_{\mathrm{LPO}} x)\ ) \vee (\top \wedge \top)\ )\ ) & \end{array}$$

is equivalent to:

$$\begin{array}{ll} \bot & \vee \\ (\ \langle \mathtt{g} \rhd \mathtt{h}\rangle \wedge \qquad (\bot \vee (\langle \mathtt{g} \rhd \mathtt{f}\rangle \wedge \top)\ ) & \vee \\ (\ \langle \mathtt{f} \rhd \mathtt{h}\rangle \wedge \ (\quad (\bot \vee (\langle \mathtt{g} \rhd \mathtt{f}\rangle \wedge \top)\ ) \vee (\top \wedge \top)\ )\ ) & \end{array}$$

This approach has some downsides:

- A solution gives you the **symbol ordering**, but you cannot read off a proof that each $\ell \succ_{\mathrm{LPO}} r$.
- The formula may become exponentially large due to duplication. (Note how $\mathtt{g}(x) \succ_{\mathrm{LPO}} \mathtt{f}(x)$ was expanded twice.)

32

Automation through "defining formulas"

For a more constructive approach, consider the following observation:

> Whenever $a \succ_{\mathrm{LPO}} b$ is used in the proof of $s \succ_{\mathrm{LPO}} t$:
> a is a subterm of s, and b is a subterm of t.

Hence, we can list – and create variables for – all the possible proof obligations in a derivation of $s \succ_{\mathrm{LPO}} t$.

Aside from the variables $\langle \mathtt{f} \rhd \mathtt{g}\rangle$, we introduce the following variables:

For every subterm a of s;

for every subterm b of t;

for every relation $\sharp$ in $\{\succ_{\text{LPO}}, \succ_{\text{LPO}}^{\text{sub}}, \succ_{\text{LPO}}^{\text{copy}}, \succ_{\text{LPO}}^{\text{lex}}\}$:

a variable $\langle a \sharp b\rangle$.

We do not need a separate variable for $\succeq_{\text{LPO}}$! Instead, we use the notation $\langle a \succeq_{\text{LPO}} b\rangle$ to denote either the formula $\top$ (if $a = b$) or the variable $\langle a \succ_{\text{LPO}} b\rangle$ (otherwise).

33

Defining formulas per variable

For each variable $\langle a \sharp b\rangle$ we now require the **defining formula**.

For example, if $a = \mathtt{f}(x, s, y)$ and $b = \mathtt{f}(x, t, u)$ with $s \neq t$:

- $\langle a \succ_{\text{LPO}} b\rangle \to \langle a \succ_{\text{LPO}}^{\text{sub}} b\rangle \vee \langle a \succ_{\text{LPO}}^{\text{copy}} b\rangle \vee \langle a \succ_{\text{LPO}}^{\text{lex}} b\rangle$
- $\langle a \succ_{\text{LPO}}^{\text{sub}} b\rangle \to \langle x \succeq_{\text{LPO}} b\rangle \vee \langle s \succeq_{\text{LPO}} b\rangle \vee \langle y \succeq_{\text{LPO}} b\rangle$
- $\neg\langle a \succ_{\text{LPO}}^{\text{copy}} b\rangle$ (since $\mathtt{f} \rhd \mathtt{f}$ cannot hold)
- $\langle a \succ_{\text{LPO}}^{\text{lex}} b\rangle \to \langle a \succ_{\text{LPO}} x\rangle \wedge \langle a \succ_{\text{LPO}} t\rangle \wedge \langle a \succ_{\text{LPO}} u\rangle \wedge \langle s \succ_{\text{LPO}} t\rangle$

34

Defining formulas: formally

For all subterms a of s and b of t, add the following formulas:

- if $a = b$, then $\neg\langle a \sharp b\rangle$ for all $\sharp \in \{\succ_{\text{LPO}}, \succ_{\text{LPO}}^{\text{sub}}, \succ_{\text{LPO}}^{\text{copy}}, \succ_{\text{LPO}}^{\text{lex}}\}$
- otherwise, if a is a variable: $\neg\langle a \sharp b\rangle$ for all $\sharp \in \{\succ_{\text{LPO}}, \succ_{\text{LPO}}^{\text{sub}}, \succ_{\text{LPO}}^{\text{copy}}, \succ_{\text{LPO}}^{\text{lex}}\}$
- otherwise, if $a = \mathtt{f}(a_1, \dots, a_n)$:
  - $\langle a \succ_{\text{LPO}} b\rangle \to \langle a \succ_{\text{LPO}}^{\text{sub}} b\rangle \vee \langle a \succ_{\text{LPO}}^{\text{copy}} b\rangle \vee \langle a \succ_{\text{LPO}}^{\text{lex}} b\rangle$
  - $\langle a \succ_{\text{LPO}}^{\text{sub}} b\rangle \to \langle a_1 \succeq_{\text{LPO}} b\rangle \vee \dots \vee \langle a_n \succeq_{\text{LPO}} b\rangle$
  - if $b = \mathtt{f}(b_1, \dots, b_n)$, and i is the lowest index such that $a_i \neq b_i$, then:
    * $\neg\langle a \succ_{\text{LPO}}^{\text{copy}} b\rangle$
    * $\langle a \succ_{\text{LPO}}^{\text{lex}} b\rangle \to \langle a \succ_{\text{LPO}} b_1\rangle \wedge \dots \wedge \langle a \succ_{\text{LPO}} b_n\rangle \wedge \langle a_i \succ_{\text{LPO}} b_i\rangle$
  - otherwise, if $b = \mathtt{g}(b_1, \dots, b_m)$ with $\mathtt{f} \neq \mathtt{g}$ then:
    * $\langle a \succ_{\text{LPO}}^{\text{copy}} b\rangle \to \langle \mathtt{f} \rhd \mathtt{g}\rangle \wedge \langle a \succ_{\text{LPO}} b_1\rangle \wedge \dots \wedge \langle a \succ_{\text{LPO}} b_m\rangle$
    * $\neg\langle a \succ_{\text{LPO}}^{\text{lex}} b\rangle$

– otherwise, $\neg\langle a \sharp b\rangle$ for $\sharp \in \{\succ_{\text{LPO}}^{\text{copy}}, \succ_{\text{LPO}}^{\text{lex}}\}$

35

Example

Let us encode: $\mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))$.

$$\begin{array}{rcl}
\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))\rangle & \to & \langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{sub}} \mathtt{h}(\mathtt{f}(x))\rangle \vee \\
 & & \langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{copy}} \mathtt{h}(\mathtt{f}(x))\rangle \vee \\
 & & \langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{lex}} \mathtt{h}(\mathtt{f}(x))\rangle \vee \\
\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{sub}} \mathtt{h}(\mathtt{f}(x))\rangle & \to & \langle \mathtt{g}(x) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))\rangle \\
\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{copy}} \mathtt{h}(\mathtt{f}(x))\rangle & \to & \langle \mathtt{f} \rhd \mathtt{h}\rangle \wedge \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}} \mathtt{f}(x)\rangle \\
\neg\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{lex}} \mathtt{h}(\mathtt{f}(x))\rangle & & \\
\langle \mathtt{g}(x) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))\rangle & \to & \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{sub}} \mathtt{h}(\mathtt{f}(x))\rangle \vee \\
 & & \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} \mathtt{h}(\mathtt{f}(x))\rangle \vee \\
 & & \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{lex}} \mathtt{h}(\mathtt{f}(x))\rangle \\
\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} \mathtt{h}(\mathtt{f}(x))\rangle & \to & \langle \mathtt{g} \rhd \mathtt{h}\rangle \wedge \langle \mathtt{g}(x) \succ_{\text{LPO}} \mathtt{f}(x)\rangle \\
\langle \mathtt{g}(x) \succ_{\text{LPO}} \mathtt{f}(x)\rangle & \to & \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{sub}} \mathtt{f}(x)\rangle \vee \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} \mathtt{f}(x)\rangle \vee \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{lex}} \mathtt{f}(x)\rangle \\
\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} \mathtt{f}(x)\rangle & \to & \langle \mathtt{g} \rhd \mathtt{f}\rangle \wedge \mathtt{g}(x) \succ_{\text{LPO}} x\rangle \\
\langle \mathtt{g}(x) \succ_{\text{LPO}} x\rangle & \to & \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{sub}} x\rangle \vee \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} x\rangle \vee \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{lex}} x\rangle \\
\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{sub}} x\rangle & \to & \top \\
\dots & &
\end{array}$$

36

Finishing up the SAT encoding

We also require:

$$\langle \ell \succ_{\text{LPO}} r\rangle$$

for each rule, since the topmost inequality should hold.

The full formula has:

- $|\ell| * |r| * 4 + 1$ formulas per rule
- $\mathcal{O}(|\Sigma|^3)$ formulas to define $\rhd$

These formulas are small, and easily converted to CNF by the Tseitin transformation.

A satisfying assignment immediately allows us to read off the proof!

37

Reading off the proof

$$\begin{array}{rcl}
\underline{\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))\rangle} & \to & \underline{\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{sub}} \mathtt{h}(\mathtt{f}(x))\rangle} \vee \\
 & & \langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{copy}} \mathtt{h}(\mathtt{f}(x))\rangle \vee \\
 & & \langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{lex}} \mathtt{h}(\mathtt{f}(x))\rangle \vee \\
\underline{\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{sub}} \mathtt{h}(\mathtt{f}(x))\rangle} & \to & \underline{\langle \mathtt{g}(x) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))\rangle} \\
\underline{\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{copy}} \mathtt{h}(\mathtt{f}(x))\rangle} & \to & \langle \mathtt{f} \rhd \mathtt{h}\rangle \wedge \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}} \mathtt{f}(x)\rangle \\
\neg\langle \mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{lex}} \mathtt{h}(\mathtt{f}(x))\rangle & & \\
\underline{\langle \mathtt{g}(x) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))\rangle} & \to & \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{sub}} \mathtt{h}(\mathtt{f}(x))\rangle \vee \\
 & & \underline{\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} \mathtt{h}(\mathtt{f}(x))\rangle} \vee \\
 & & \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{lex}} \mathtt{h}(\mathtt{f}(x))\rangle \\
\underline{\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} \mathtt{h}(\mathtt{f}(x))\rangle} & \to & \underline{\langle \mathtt{g} \rhd \mathtt{h}\rangle} \wedge \underline{\langle \mathtt{g}(x) \succ_{\text{LPO}} \mathtt{f}(x)\rangle} \\
\underline{\langle \mathtt{g}(x) \succ_{\text{LPO}} \mathtt{f}(x)\rangle} & \to & \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{sub}} \mathtt{f}(x)\rangle \vee \underline{\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} \mathtt{f}(x)\rangle} \vee \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{lex}} \mathtt{f}(x)\rangle \\
\underline{\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} \mathtt{f}(x)\rangle} & \to & \underline{\langle \mathtt{g} \rhd \mathtt{f}\rangle} \wedge \mathtt{g}(x) \succ_{\text{LPO}} x\rangle \\
\underline{\langle \mathtt{g}(x) \succ_{\text{LPO}} x\rangle} & \to & \underline{\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{sub}} x\rangle} \vee \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{copy}} x\rangle \vee \langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{lex}} x\rangle \\
\underline{\langle \mathtt{g}(x) \succ_{\text{LPO}}^{\text{sub}} x\rangle} & \to & \top \\
\dots & &
\end{array}$$

If the underlined variables are set to *true*, we can read off:

- $\mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))$ because $\mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{sub}} \mathtt{h}(\mathtt{f}(x))$ – so by rule **(sub)**;
- $\mathtt{f}(\mathtt{g}(x)) \succ_{\text{LPO}}^{\text{sub}} \mathtt{h}(\mathtt{f}(x))$ holds because $\mathtt{g}(x) \succ_{\text{LPO}} \mathtt{h}(\mathtt{f}(x))$.
- This holds by rule **(copy)**, because $\mathtt{g} \rhd \mathtt{h}$ and $\mathtt{g}(x) \succ_{\text{LPO}} \mathtt{f}(x)$.
- This holds by rule **(copy)**, because $\mathtt{g} \rhd \mathtt{f}$ and $\mathtt{g}(x) \succ_{\text{LPO}} x$.
- This holds by rule **(sub)**.

4.3 Recursive path orderings

Variants of LPO

The lexicographic path ordering is a **recursive path ordering**. Many variations exist, for instance:

- replacing the **(lex)** rule: to have $\mathtt{f}(s_1, \dots, s_n) \succ_{\text{LPO}} \mathtt{f}(t_1, \dots, t_n)$, instead of comparing $[s_1, \dots, s_n]$ and $[t_1, \dots, t_n]$ lexicographically, we could use a different method:
  - simply placewise comparison (i.e., each $s_i \succeq_{\text{LPO}} t_i$ and at least one strict)
  - the multiset comparison (see the assignment)
  - either lexicographic or multiset, depending on the function symbol
- using a **quasi-ordering** for the symbol comparison $\rhd$, hence allowing two distinct function symbols to be equal in the ordering; this is very useful for mutually recursive function symbols (e.g., $\mathtt{f}(\mathtt{s}(x)) \Rightarrow \mathtt{g}(x)$, $\mathtt{g}(\mathtt{s}(x)) \Rightarrow \mathtt{f}(x)$)

- if `c` is the smallest symbol in $\rhd$ and has arity 0, letting $x \succeq_{\mathrm{LPO}} \mathtt{c}$ also for variables

---

39

Subterm property

The lexicographic path ordering (and all variations discussed before) has the following property:

Theorem

If s is a proper subterm of t, then $t \succ_{\mathrm{LPO}} s$.

Proof. This easily follows from rule **(sub)**. □

This will be useful in **superposition**, which we will discuss in a later lecture.

5. Exercises

40

Quiz

1. What is the difference between weak and strong normalisation?
2. What is the difference between local confluence and general confluence?
3. Use the lexicographic path ordering (by hand) to prove termination of:

$$\begin{array}{rcl} \mathtt{f}(\mathtt{g}(x), \mathtt{g}(\mathtt{b})) & \Rightarrow & \mathtt{f}(x, x) \\ \mathtt{g}(\mathtt{a}) & \Rightarrow & \mathtt{b} \\ \mathtt{b} & \Rightarrow & \mathtt{a} \end{array}$$

4. What properties should a relation $\succ$ satisfy to be a reduction order?

41

Some Advice for the Practical assignment

Suppose: you have to find a solution for the Traveling Salesman Problem.

Problem: find **the shortest** route visiting a number of cities

Solution:

- encode, for given N, the problem "find a route $\leq N$" into SMT
- use binary search to find the smallest N for which this is satisfiable

You will need to do something like this in the practical assignment. In some cases, you can do this by hand – for example in the traveling salesman problem, make an educated guess which bound will be close, and then run the solver a few times around that bound. This is probably the best way to start! Once you have written the program to test your bound and everything works well, then you can adapt it to automate the search.