2. Red. Systems & Term Rewriting

(a) Reductions in Agda.
(b) Reduction systems.
(c) Termination, confluence, normalisation.
(d) Term rewriting systems.

(a) Reductions in Agda

- Functional programming is essentially based on term reduction:
- Assume we introduce the natural numbers as an algebraic data type built from 0 and S (this is actual Agda code):

$$\begin{aligned} &\mathrm{data}\ \mathbb{N} : \mathrm{Set}\ \mathrm{where} \\ &\quad \mathrm{Z} \ : \ \mathbb{N} \\ &\quad \mathrm{S} \ : \ \mathbb{N} \to \mathbb{N} \end{aligned}$$

(reductionSystems1.agda)

- We write here Z instead of 0, since the symbol 0 will be reserved for the built-in integers.
- $\mathrm{S}\ n$ stands for $n + 1$.

Notations in Agda

```
data ℕ : Set where
  Z  :  ℕ
  S  :  ℕ → ℕ
```

- $\mathrm{data}\ \mathbb{N} : \mathrm{Set}$ means that we have introduced a new set, which is given by the constructors which follow after the symbol where.
  - What is in most programming languages called type is in Agda for historic reasons called “Set”.
  - The above code introduces a new set, namely the set of natural numbers.
  - It has two constructors: the constant Z, and S which takes as argument an $n : \mathbb{N}$ and returns an element of $\mathbb{N}$.

$\mathbb{N}$ as a Reduction System

- So the elements of $\mathbb{N}$ are

$$\mathrm{Z} \qquad \mathrm{S}\ \mathrm{Z} \qquad \mathrm{S}\ (\mathrm{S}\ \mathrm{Z}) \qquad \mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z})) \qquad \cdots$$

- We can now define $+$ and $*$ in $\mathbb{N}$ by induction over the definition of $\mathbb{N}$.
  - For those with mathematical problems: “Induction over the definition of $\mathbb{N}$” means roughly case distinction on $\mathbb{N}$ in a terminating way.

Definition of $+$

$$
\begin{array}{lcl}
\_ + \_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
n + \mathrm{Z} & = & n \\
n + \mathrm{S}\, m & = & \mathrm{S}\, (n + m)
\end{array}
$$

- _ + _ means that
  - the first argument (denoted by the first _) of $+$ is placed before $+$,
  - the second argument (denoted by the second _) of $+$ is placed after $+$,

  which means here that $+$ is used infix:
  - we write $s + t$ instead of _ + _ s t.

Mixed Fixed Symbols

- Agda allows arbitrary mixed fixed symbols:
  - For instance we can define _strange_symbol_ as a symbol which is used as
    - n strange m symbol k
      for
    - _strange_symbol_ n m k.
- There are almost no restrictions.
- If the parsing of an expression is ambiguous, a parse error is given – then one needs to resolve the ambiguity by using parentheses.

Definition of $+$

- The definition of $+$ above means that we have the following reductions:

$$\begin{aligned} s + \mathrm{Z} &\longrightarrow s \ , \\ s + \mathrm{S}\, t &\longrightarrow \mathrm{S}\, (s + t) \ . \end{aligned}$$

- Note that S binds more than $+$. So
  - $\mathrm{S}\, r + s$ reads $(\mathrm{S}\, r) + s$.
  - $r + \mathrm{S}\, s$ reads $r + (\mathrm{S}\, s)$.
- We have $2 + 2 \longrightarrow 4$:

$$\begin{aligned} \mathrm{S}\, (\mathrm{S}\, \mathrm{Z}) + \mathrm{S}\, (\mathrm{S}\, \mathrm{Z}) &\longrightarrow \mathrm{S}\, (\mathrm{S}\, (\mathrm{S}\, \mathrm{Z}) + \mathrm{S}\, \mathrm{Z}) \\ &\longrightarrow \mathrm{S}\, (\mathrm{S}\, (\mathrm{S}\, (\mathrm{S}\, \mathrm{Z}) + \mathrm{Z})) \\ &\longrightarrow \mathrm{S}\, (\mathrm{S}\, (\mathrm{S}\, (\mathrm{S}\, \mathrm{Z}))) \end{aligned}$$

Our first Agda Example

- We are going to show how we can deal with this example in Agda.

Installation

- A substantially improved version of Agda called **Agda2** has been released recently.
  - In this module Agda2 will be used, which has a completely different syntax from Agda1.
- Currently the installation requires some work.
  - Easy to compile versions (which exist for Agda1) are in preparation, but have **not been released yet**.
    - For Agda1 there exists a 1-click-Windows-installer.
    - We hope that this problem will soon be solved.
- Anton Setzer has installed Agda2 under **Linux**.
- He is working on getting it installed under **Windows.**
- Agda can be installed under **Macintosh** as well.

Installation of Agda

- Instructions on how to install Agda under Linux and hopefully as well under Windows will soon be created and can then soon be found under
  http://www.cs.swan.ac.uk/∼csetzer/
  othersoftware/agda2/agda2installation.html
- The installation will provide an Emacs mode for Agda files.
- If a file with extension .agda is loaded into Emacs, then this mode is invoked.
- The source code for the **examples** given in this lecture will be available from the course home page (the names of the files are added in the notes, e.g. in the form **reductionSystems1.agda**.

Agda in the Linux Lab

- Agda will be installed in the Linux lab.
  - When this is ready, follow the item “Getting started with Agda” on the home page of this module.
  - Please check whether the installation works.

Help System of Emacs

- Note that Emacs has an excellent and well-written help system.
  - Includes a hypertext version of most features of the Emacs, search facilities, descriptions of all variables.
- The help system is activated using Control-h plus an additional key stroke.
  - Emacs notation: `C-h`.
- A quick tutorial, which introduces the help system starts when typing, after Emacs has been started, `C-h t`

Working in Agda

- Once, Agda is installed, the above can be defined as follows:
  - One opens in Emacs a file with extension "**.agda**", e.g. "reductionSystems1.agda".
  - Emacs will switch into **Agda mode**.
  - Code written needs to be part of a "**module**" (we will not discuss details of the module system in this lecture course.)
  - We will **create a module** by typing in:

```
module reductionSystems1 where
```

Working in Agda

module reductionSystems1 where

- Now we add the definition of $\mathbb{N}$:

```
module reductionSystems1 where


data ℕ : Set where
  Z  :  ℕ
  S  :  ℕ → ℕ
```

Blanks around “:”

- Please note that there needs to be a blank around all “:”.
  - $\mathbb{Z}$: without a blank in between is considered by Agda as an identifier $\mathbb{Z}$:.
  - :$\mathbb{N}$ without a blank in between is considered by Agda as an identifier :$\mathbb{N}$.
  - Only brackets “(”, “{”, “)”, “}” and blanks (and possibly some other symbols not discovered yet by A. Setzer) break identifiers.

Typing in Special Symbols

- The standard installation of Agda activates a special mode of Emacs which allows to type in special symbols.
- Special symbols are typed in by using command sequences inspired by LaTeX.
- For instance
  - $\mathbb{N}$ is written by typing in `\Bbb{N}`,
  - α by typing in `\alpha`.
- On the homepage for this course under “Other Course Material” a file leimListOfSymbols.tut (which was extracted from the source code of this mode) will be made available, which contains the key bindings for this mode.

Loading the Buffer

- Agda doesn't realise any changes in the buffer, unless we load it.
- For this we can use the **main menu**, which one obtains by right-clicking on the word "Agda2" in the Panel.
- By choosing from the main menu "**Load**", we load the buffer.
- An additional buffer called ∗**All Goals**∗ appears, which will be explained later.
- If there was an error loading the buffer, then instead of ∗**All Goals**∗ a buffer ∗**Error**∗ is displayed showing an error message.

Keyboard Short Cuts

- In the main menu, for each command a keyboard short cut is presented.
  It is advisable to learn the most frequently used ones.
- In order to type check the buffer, we can use the keyboard command
  C-c C-x C-l.

Goals

- When defining code, one can leave some code open for being filled in a later step.
- These holes are called **goal**, which stands for a term not yet defined.
  - Syntax in Agda: $\{!\ \ !\}$, written in "green" in the Emacs mode.
- One can type in as well "?" for a goal, which will then be converted, when loading the buffer, into the symbol $\{!\ \ !\}$.
- So let's type in the beginning of the definition of $+$:

$$\begin{aligned} \_+\_ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + m &= ? \end{aligned}$$

Goals

- When the buffer is loaded, the goal will be shown in a different colour, and one can only edit inside or outside the goal.
- Each goal gets a number.
- When right-clicking on the goal, the **goal-menu** is opened.
  - when using Emacs, is activated and becomes the goal-menu. (Outside a goal this menu doesn't exist).
- If one wants to edit the buffer in a way which is impossible because of the restrictions of editing goals, one can do so by first deactivating agda using Menu **Deactivate Agda (C-c C-x C-d)**
  - When loading the buffer again one gets back to the state in which goals have special status.

Goals

- Goals are numbered by the order in which they were created.
- Goals are displayed together with their type in separate buffer called "∗ Goals ∗".
- This can be activated as well by using menu "**Show Goals (C-c C-?)**".
- In Emacs mode, goals have a special status.
  - When typing in text into a goal, the goal expands.

Goals

- Using the goal-menu, we can find out
  - what type is expected:
    Goal Type (C-c C-t).
    Agda shows: ?0 : $\mathbb{N}$
  - There is as well a variant
    Goal Type (normalised).
    It evaluates the type of the goal using reduction rules (see later).
- Using the main menu we can always show the types of all goals (**Show Goals (C-c C-?)**).
- We can as well jump to the next and previous goal using menu items **Next Goal (C-c C-f)** and **Previous Goal (C-c C-p)**.

Context

- Inside a goal we can as well find out the current context:
  - Using menu **Context (environment) (C-c C-e)**.
  - In our example Agda shows (apart from some library functions):

    $n \quad : \quad \mathbb{N}$

    $m \quad : \quad \mathbb{N}$

  - So when defining $n + m$ we can make use of $n : \mathbb{N}$, $m : \mathbb{N}$ and the function $+$ we are defining at present.

Case Distinction

$$\_ + \_ \ : \ \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$n + m \ = \ \{!\ !\}$$

- In order to define $n + m$, we have to make a case distinction on whether $m = \mathrm{Z}$ or $m = \mathrm{S}\ m'$.
- This can be achieved by replacing the line $n + m = \{!\ !\}$ by two lines for the two cases.
- In order to achieve this we deactive Agda by using main menu command **Deactivate Agda (C-c C-x C-d)**.
- Then we can replace the line $n + m = \{!\ !\}$ by two lines as follows:

$$\begin{aligned} \_ + \_ \ &: \ \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} \ &= \ \{!\ !\} \\ n + \mathrm{S}\ m \ &= \ \{!\ !\} \end{aligned}$$

Coverage Checker

- Agda has built in a **coverage checker**, which makes sure that if one makes a case distinction as above, then all cases are **covered**.
- If we omit one of the cases, e.g. the S-case, and load the buffer:

$$\begin{aligned} \_ + \_ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} &= \{!\ !\} \end{aligned}$$

  then we get an error message (see next slide)

Error Message

```
/home/csetzer/lectures/07/
intertheo/agdalectureexamples/
reductionSystems1.agda:9,1-17 Incomplete
pattern matching for _+_.
Missing cases:
_+_ _ (S_)
when checking the definition of _+_
```

- So in the definition of _ + _, the case where the first argument is arbitrary, and the second argument is of the form S applied to something is missing.

Give and Refine

$$\begin{aligned} \_ + \_ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} &= \{!\ !\} \\ n + \mathrm{S}\ m &= \{!\ !\} \end{aligned}$$

- We can solve the first goal by typing in the value n.
- Then we can right-click on the goal and use from the goal-menu either **Give (C-c C-SPC)** or **Refine (C-d C-r)**.
  - Give works when one has an exact solution as in the situation above.
  - Refine works not always, but allows as well partial solutions, which need refinement. See the case $\mathrm{S}\ n$ on the next slide.

Refine

- We obtain in both cases

$$\begin{aligned} \_ + \_ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} &= n \\ n + \mathrm{S}\ m &= \{!\ \ !\} \end{aligned}$$

- We can use the refine mechanism in case of $n + \mathrm{S}\ m$ as follows:
  - We know that the solution will be of the form $\mathrm{S}\ \{!\ \ !\}$.
  - We can now type into the goal S and then use the command refine.
  - Agda knows that if we apply S to one argument (which is a natural number), then we get something which solves the goal.

Refine

$$
\begin{array}{lcl}
\_ + \_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
n + \mathrm{Z} & = & n \\
n + \mathrm{S}\, m & = & \{!\ \ !\}
\end{array}
$$

- If we type into the goal S and refine it, we obtain

$$
\begin{array}{lcl}
\_ + \_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
n + \mathrm{Z} & = & n \\
n + \mathrm{S}\, m & = & \mathrm{S}\, \{!\ \ !\}
\end{array}
$$

Refine

$$\begin{aligned} \_ + \_ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} &= n \\ n + \mathrm{S}\, m &= \mathrm{S}\, \{!\ !\} \end{aligned}$$

- We want to solve the goal by typing in something of the form $\{!\ !\} + \{!\ !\}$, which is $\_ + \_\ \{!\ !\}\ \{!\ !\}$.
- We can type into the goal $\_ + \_$ and use refine
- Agda realises that $\_ + \_$ applied this time to 2 arguments solves the goal, and rearranges the result in infix form.
  - We obtain

$$\begin{aligned} \_ + \_ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} &= n \\ n + \mathrm{S}\, m &= \mathrm{S}\, (\{!\ !\} + \{!\ !\}) \end{aligned}$$

Termination Checker

$$\begin{aligned} \_ + \_ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} &= n \\ n + \mathrm{S}\ m &= \mathrm{S}\ (\{!\ \ !\} + \{!\ \ !\}) \end{aligned}$$

- The $+$ and the defining symbol _ + _ are now marked in red.
- This is because of the termination checker.
- Agda disallows non-terminating programs, like

$$\begin{aligned} f &: \mathbb{N} \to \mathbb{N} \\ f\ n &= f\ n \end{aligned}$$

Termination Checker

- The termination check is necessary, since otherwise the logic of Agda is inconsistent.
  - This is no problem for Agda used as a dependently typed programming language.
    - But then the validity of any proved parts of it (e.g. that a list returned is sorted) will no longer be guaranteed.

Termination Checker

- In the code

$$\begin{aligned} \_ + \_ &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} &= n \\ n + \mathrm{S}\ m &= \mathrm{S}\ (\{!\ !\} + \{!\ !\}) \end{aligned}$$

we don't know yet whether this will pass the termination checker when the goals are solved or not.

- If we solve it by

$$n + \mathrm{S}\ m = \mathrm{S}\ (n + \mathrm{S}\ m)$$

then we obtain a non-terminating program.

Termination Checker

- If we solve

$$
\begin{array}{lcl}
\_ + \_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
n + \mathrm{Z} & = & n \\
n + \mathrm{S}\ m & = & \mathrm{S}\ (\{!\ \ !\}\ +\ \{!\ \ !\})
\end{array}
$$

in the correct way

$$n + \mathrm{S}\ m = \mathrm{S}\ (n + m)$$

it will pass the termination checker.

Termination Checker

- Note that only a warning (in the form of the symbols coloured) is issued, but no error is issued.
  - This is since this warning only indicates that there is potential problem.
    - It might be solved once all goals are solved.
    - There are as well limitations to any termination checker:
      It is in principal not possible to write a termination checker which accepts all terminating Agda programs.
      - So a program could be terminating and therefore okay, but still not pass the termination checker.
      - This will be discussed later.

Termination Checker

- In order to make sure that there are no termination check problems left, one can use from a shell the command
  "**agda** file"
  e.g.
  agda ~csetzer/r/reductionSystems1.agda
- This command will check the file and report type errors, problems of the coverage checker and problems of the termination checker.
- **Code submitted as coursework should be checked this way,** in order to guarantee that there are no hidden errors.

Finishing Definition of $+$

$$\_ + \_ \;:\; \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$n + \mathrm{Z} \;=\; n$$
$$n + \mathrm{S}\, m \;=\; \mathrm{S}\,(\{!\ \ !\} + \{!\ \ !\})$$

- We can solve now the two goals by using n and m and goal-menu refine or give and obtain

$$\_ + \_ \;:\; \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$n + \mathrm{Z} \;=\; n$$
$$n + \mathrm{S}\, m \;=\; \mathrm{S}\,(n + m)$$

Indentation Sensitivity

- Agda is indentation sensitive.
- So often instead of having parentheses "$\{\langle \mathrm{Code} \rangle\}$", as in other languages, all lines belonging to $\langle \mathrm{Code} \rangle$ have to be intended more then the surrounding code, and usually in the same way.
- Therefore top level definitions have to start in column1. Otherwise they are considered as being an extension of a previous definition.
- All code belonging to such a definition in later columns has to be intended at least once.

Indentation Sensitivity

- Example: The following causes an error:

$$\begin{aligned}&\mathrm{data}\ \mathbb{N} : \mathrm{Set}\ \mathrm{where}\\ &\quad \mathrm{Z} : \mathbb{N}\\ &\quad \mathrm{S}\ : \mathbb{N} \to \mathbb{N}\end{aligned}$$

  - Agda assumes that S is not a constructor of $\mathbb{N}$, but a function, which is not defined yet.

Indentation Sensitivity

- Example: The following causes an error:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{data}\ \mathbb{N} : \mathrm{Set}\ \mathrm{where}} \\
\quad \mathrm{Z} & : & \mathbb{N} \\
\quad \mathrm{S} & : & \mathbb{N} \to \mathbb{N} \\
 & & \\
\quad \_ + \_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\quad n + \mathrm{Z} & = & n \\
\quad n + \mathrm{S}\ m & = & \mathrm{S}\ (n + m)
\end{array}
$$

- Now $\_ + \_$ is considered as a constructor of $\mathbb{N}$, and the equation $n + \mathrm{Z} = n$ doesn't make sense for a constructor and causes a parse error.

Definition of Multiplication

- Definition of $\_ * \_$:

$$\begin{array}{lcl} \_ * \_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n * \mathrm{Z} & = & \mathrm{Z} \\ n * \mathrm{S}\, m & = & (n * m) + n \end{array}$$

- This means that we have the following reductions:

$$\begin{array}{lcl} s * \mathrm{Z} & \longrightarrow & \mathrm{Z} \ , \\ s * \mathrm{S}\, t & \longrightarrow & s * t + s \ . \end{array}$$

Binding of Symbols

- We can add the following two lines:

$$\text{infixl } 60 \ \_ + \_$$
$$\text{infixl } 80 \ \_ * \_$$

- This means that
  - $\_ + \_$ and $\_ * \_$ are infix left-associative:
    - $n + m + k$ is interpreted as $(n + m) + k$, similarly for $*$.
  - If we had used infixr instead, we obtan
    - $n + m + k$ is interpreted as $n + (m + k)$, similarly for $*$.
  - If we use infix, then $n + m + k$ is considered as ambiguous and causes a parse error.

Binding of Symbols

infixl 60 _ + _

infixl 80 _ * _

- Furthermore that _ * _ has a higher number than _ + _ means that $*$ binds more than $+$:
  - $n + m * k$ is interpreted as $n + (m * k)$.
- Without stating the statements above $n + m * k$ is considered as ambiguous and causes a parse error.

Complete Definition of $+$ and $*$

We obtain as definition of $+$ and $*$:

$$\begin{array}{l}\text{infixl } 60 \ \_ + \_ \\ \text{infixl } 80 \ \_ * \_\end{array}$$

$$\begin{array}{lcl}\_ + \_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + \mathrm{Z} & = & n \\ n + \mathrm{S}\, m & = & \mathrm{S}\, (n + m)\end{array}$$

$$\begin{array}{lcl}\_ * \_ & : & \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n * \mathrm{Z} & = & \mathrm{Z} \\ n * \mathrm{S}\, m & = & n * m + n\end{array}$$

Testing the above in Agda

- In order to test the above we can make use of the main-menu (or goal-menu)
  Compute normal form (C-c C-n).
- It will ask in the mini-buffer for an expression.
- If we type in

  $$\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}) + \mathrm{S}\ (\mathrm{S}\ \mathrm{Z})$$

  (for $2 + 2$), Agda shows in another buffer the result

  $$\mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z})))\ ,$$

  i.e. 4.

Using the Builtin Natural Numbers

- We can use as well the builtin natural numbers:
- If we add the following code

```
{-#  BUILTIN  NATURAL    ℕ      #-}
{-#  BUILTIN  ZERO       Z      #-}
{-#  BUILTIN  SUC        S      #-}
{-#  BUILTIN  NATPLUS    _+_    #-}
{-#  BUILTIN  NATTIMES   _*_    #-}
```

then $\mathbb{N}$ is identified with the builtin type of natural numbers, and Z, S, $\_+\_$, $\_*\_$ with the corresponding builtin operations.

Using the Builtin Natural Numbers

- Then we can define for instance

$$\begin{aligned} a &: \mathbb{N} \\ a &= 5 \end{aligned}$$

- and if we compute the normalform of $7 + 9$ we obtain 16.

Agda and Non-Termination

- If one makes a mistake and defines _ + _ so that it doesn't terminate (e.g. defining in case of $\mathrm{S}\ m$ $\mathrm{S}\ (n + \mathrm{S}m)$ instead of $\mathrm{S}\ (n + m)$), then Agda will crash, and not display anything.
  - This can be observed by checking the buffer $*$**ghci**$*$.
    - All Emacs activities will result in Haskell commands been issued to this buffer, and the result is then used in order to modify the emacs buffer.
    - Since the interactive Glasgow Haskell Compiler ghci is used, the buffer for communicating is called $*$ghci$*$.
    - If Agda crashes, one sees that a command was issued there, but no response was returned yet.

Agda and Non-Termination

- If one has non-terminating recursion, Agda might crash during type checking and at other places as well.
  - Can be observed as well by switching to buffer $*$ghci$*$.
- The ghci buffer is a buffer to Haskell with Agda loaded.
  - It can be used for carrying out haskell computations, e.g. for computing $3 + 3$ in Haskell.

Reduction Systems and Agda

- We want $2 + 2$ and 4 to be the same.
- In Agda, referring to our self-defined natural numbers, this means that $\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}) + \mathrm{S}\ (\mathrm{S}\ \mathrm{Z})$ and $\mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z})))$ should be the same.
- We have just seen that $\mathrm{S}\ (\mathrm{S}\ \mathrm{Z}) + \mathrm{S}\ (\mathrm{S}\ \mathrm{Z})$ reduces to $\mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ (\mathrm{S}\ \mathrm{Z})))$.
- The underlying principle behind this is:

> If a term reduces to another term,
> then these two terms are the same.

Towards General Reduction System

- Since we have dependent types, equality plays a rôle in type checking
  - If $A\ r$ is a type depending on r and $a : A\ r$, then $a : A\ s$ provided that r and s are equal.
- In order to understand Agda better, we will study in the following general reduction relations.
- Given by a set of Terms T and a reduction relation $s \longrightarrow t$ between terms s and t.

(b) Reduction Systems

- A **reduction system** is a pair $(\mathrm{T}, \longrightarrow)$ consisting of a set T (of terms) and a binary relation $\longrightarrow$ on T.
- We write $s \longrightarrow t$ for "s, t are in relation $\longrightarrow$" and say usually "s reduces to t".
- **Example 1:**
  - Let T be the set of terms formed from 0, S, $+$ and $*$ in the usual way.
    - So for instance 0, $\mathrm{S}\ 0$ and $\mathrm{S}\ 0 + 0$ are elements of T.
  - Let $\longrightarrow$ be the reduction relation defined as before.
    - So we have for instance
      $\mathrm{S}(\mathrm{S}\ 0) + 0 \longrightarrow \mathrm{S}(\mathrm{S}\ 0)$
      $\mathrm{S}(\mathrm{S}\ 0) + \mathrm{S}\ 0 \longrightarrow \mathrm{S}(\mathrm{S}(\mathrm{S}\ 0))$
  - Then $(\mathrm{T}, \longrightarrow)$ forms a reduction system.

Example 2 (Reduction System)

- A simple reduction system is $\mathrm{T} = \mathbb{N}$ with reductions $n + 1 \longrightarrow n$ for $n \in \mathbb{N}$:

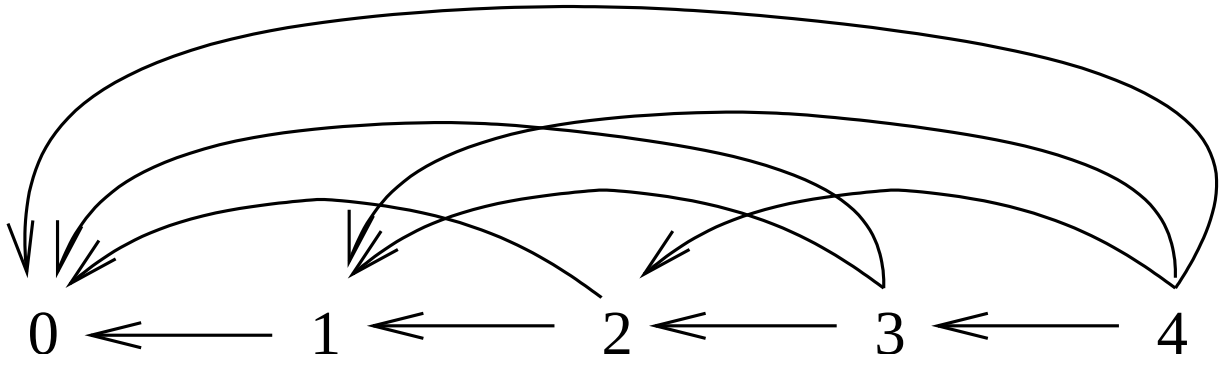
$$0 \longleftarrow 1 \longleftarrow 2 \longleftarrow 3 \longleftarrow 4 \longleftarrow \cdots$$

  - So we have reductions of the form:

$$5 \longrightarrow 4 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1 \longrightarrow 0 \ .$$

Example 3 (Reduction System)

- Another simple example is $\mathrm{T} = \mathbb{N}$ with reductions $n \longrightarrow m$ for $n, m \in \mathbb{N}$ s.t. $n > m$:

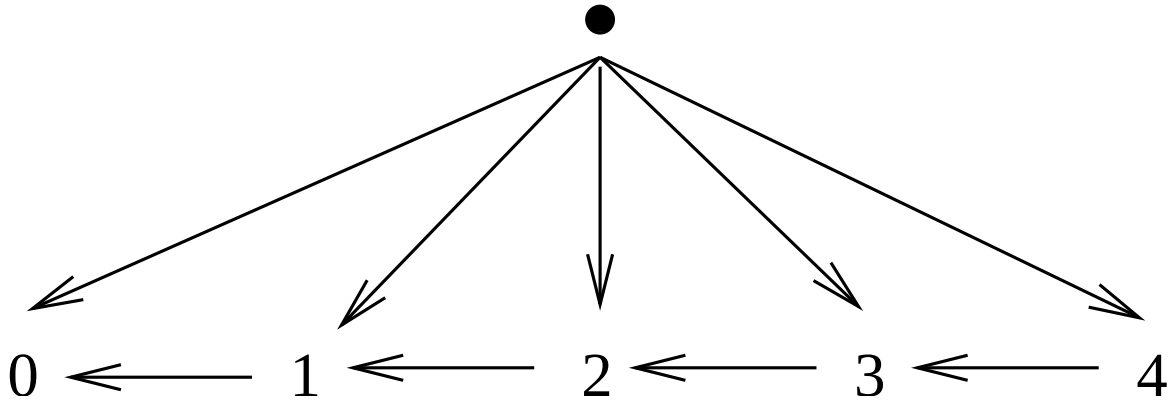


  - So we have reductions of the form:

$$23 \longrightarrow 11 \longrightarrow 3 \longrightarrow 1 \longrightarrow 0 \ .$$

Example 4 (Reduction System)

- A further simple example is $\mathbb{T} = \mathbb{N} \cup \{\bullet\}$, with reductions $n + 1 \longrightarrow n$ for $n \in \mathbb{N}$,
  and $\bullet \longrightarrow n$ for $n \in \mathbb{N}$:



  - So we have reductions of the form:

$$\bullet \longrightarrow 5 \longrightarrow 4 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1 \longrightarrow 0 \ .$$

$\longrightarrow$, $\longleftrightarrow$

- If $(\mathrm{T}, \longrightarrow)$ is a reduction system, we define for $s, t \in \mathrm{T}$

$$s \longleftarrow t \quad :\Leftrightarrow \quad t \longrightarrow s$$
$$s \longleftrightarrow t \quad :\Leftrightarrow \quad s \longrightarrow t \vee s \longleftarrow t$$

- Note that we are using "$\vee$", not "$\wedge$".

$\longrightarrow^*$ and $\longleftrightarrow^*$

- In Agda we said we identify two terms which **reduce** to each other in possible multiple steps.
- Therefore we study two concepts:
  - One is $s \longrightarrow^* t$, which means that s reduces to t in possibly multiple steps.
    - When Agda reduces a term s, it returns a term t s.t. $s \longrightarrow^* t$, and t cannot reduce any further.
  - One is $s \longleftrightarrow^* t$, which is the equality induced by $\longrightarrow$.
    - So Agda identifies terms s and t s.t. $s \longleftrightarrow^* t$.

$\longrightarrow^*$

If $(\mathrm{T}, \longrightarrow)$ is a reduction system, we define

- $s \overset{*}{\longrightarrow} t$ iff there exists a (possibly empty) sequence

$$s \equiv s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots \longrightarrow s_n \equiv t$$

- By empty sequence we mean: $n = 0$ is allowed, in which case we have $s \equiv t$.
- (We write $\equiv$ for syntactic equality between terms in order to avoid confusion with equality modulo reductions introduced later and denoted by $=$).

Example

If we take $\mathbb{N}$ with reductions $n + 1 \longrightarrow n$ for $n \in \mathbb{N}$:

$$0 \longleftarrow 1 \longleftarrow 2 \longleftarrow 3 \longleftarrow 4 \longleftarrow \cdots$$

Then $5 \longrightarrow 4 \longrightarrow 3 \longrightarrow 2$, therefore $5 \longrightarrow^* 2$.
In general $n \longrightarrow^* m \Leftrightarrow n \geq m$.

$\longrightarrow^*$

- In order to express the above shorter, one says that $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$, i.e. the least reflexive and transitive relation containing $\longrightarrow$.
- This means the following:
  - $r \longrightarrow s$ implies $r \longrightarrow^* s$.
  - $\longrightarrow^*$ is reflexive, i.e. for all $r \in \mathbb{T}$ we have $r \longrightarrow^* r$.
  - $\longrightarrow^*$ is transitive, i.e. $r \longrightarrow^* s \longrightarrow^* t$ implies $r \longrightarrow^* t$.
  - If there is any other relation $\longrightarrow'$, which is reflexive, transitive and contains $\longrightarrow$, then $r \longrightarrow^* s$ implies $r \longrightarrow' s$.
- The next slides contain a proof that $\longrightarrow^*$ is indeed the reflexive transitive closure of $\longrightarrow$. We jump over it. Jump over proof.

Proof

- We show that $\longrightarrow^*$, as defined originally, is in fact the reflexive and transitive closure of $\longrightarrow$:
  - If $r \longrightarrow s$, then clearly $r \longrightarrow^* s$ (take $n = 1$).
  - $\longrightarrow^*$ is reflexive: We have $r \longrightarrow r$ for $r \in \mathrm{T}$, by having $n = 0$ in the definition of $\longrightarrow^*$.
  - $\longrightarrow^*$ is transitive: Assume $r \longrightarrow^* s \longrightarrow^* t$. Then there exist n, m, r_i, s_i s.t.

$$r \equiv r_0 \longrightarrow r_1 \longrightarrow \cdots \longrightarrow r_n \equiv s$$
$$s \equiv s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_m \equiv t$$

  But this implies $r \longrightarrow^* t$.

Proof (Cont.)

- Furthermore, if $\longrightarrow'$ is another relation, which contains $\longrightarrow$, and which is reflexive and transitive, then it contains $\longrightarrow^*$:
  - Assume $\longrightarrow'$ having these properties.
  - Assume $r \longrightarrow^* s$.
  - Then there exist $n \in \mathbb{N}, r_i$ s.t.

$$r \equiv r_0 \longrightarrow r_1 \longrightarrow \cdots \longrightarrow r_n \equiv s$$

  - In case $n = 0$ we have $r \equiv s$, therefore $r \longrightarrow' s$.

Proof (Cont.)

$r \equiv r_0 \longrightarrow r_1 \longrightarrow \cdots \longrightarrow r_n \equiv s$

- In case $n > 0$ we have by the fact that $t \longrightarrow t'$ implies $t \longrightarrow' t'$ that

$$r \equiv r_0 \longrightarrow' r_1 \longrightarrow' \cdots \longrightarrow' r_n \equiv s$$

- But since $\longrightarrow'$ is transitive, it follows

$$r \longrightarrow' s$$

and we are done.

- Note how easy it is to overlook that the case $n = 0$ has to be treated separately.
  - If one does this more formally, or using an interactive theorem proving, one would notice this missing case.

If $(\mathrm{T}, \longrightarrow)$ is a reduction system, we define

- $s \overset{*}{\longleftrightarrow} t$ iff there exists a (possibly empty) sequence

$$s \equiv s_0 \longleftrightarrow s_1 \longleftrightarrow s_2 \longleftrightarrow \cdots \longleftrightarrow s_n \equiv t$$

$\longleftrightarrow^*$

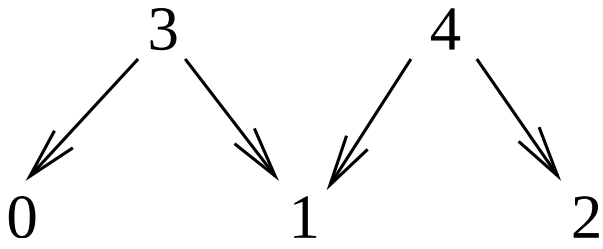
$r \longleftrightarrow^* t$ iff

$s \equiv s_0 \longleftrightarrow s_1 \longleftrightarrow s_2 \longleftrightarrow \cdots \longleftrightarrow s_n \equiv t$

- Note that if we want to identify two elements r, s, if $r \longrightarrow s$, we have to identify r, s if $r \longleftrightarrow^* s$.
  - If we want to identify elements r, s s.t. $r \longrightarrow s$, we have to identify as well elements r, s s.t. $r \longleftarrow s$.
  - Then we have to identify elements r, s s.t. $r \longleftrightarrow s$.
  - Therefore we have, if n, s_i are as in the definition of $s \longleftrightarrow^* t$, to identify s_0 and s_1; s_1 and s_2; etc.; s_{n-1} and s_n.
  - Therefore we have to identify s_0 and s_n.

Example

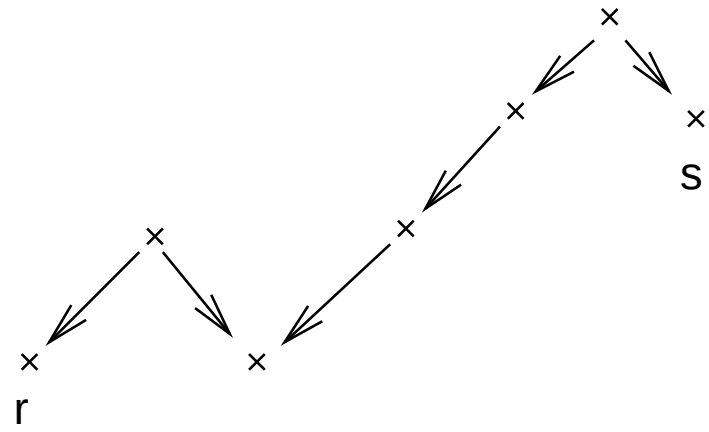
Assume the following reduction system:



Then $0 \longleftarrow 3 \longrightarrow 1 \longleftarrow 4 \longrightarrow 2$, therefore $0 \longleftrightarrow^* 2$.

Jump over next slide.

Illustration of $\longleftrightarrow^*$



In the reduction system above we have $r \longleftrightarrow^* s$.

$\longleftrightarrow^*$

- In order to express the above shorter, one says that $\longleftrightarrow^*$ is the reflexive, symmetric and transitive closure of $\longrightarrow$, i.e. the least reflexive, symmetric and transitive relation containing $\longrightarrow$.
- This means the following:
  - $s \longrightarrow t$ implies $s \longleftrightarrow^* t$.
  - $\longleftrightarrow^*$ is reflexive, and transitive.
  - $\longleftrightarrow^*$ is symmetric, i.e. $s \longleftrightarrow^* t$ implies $t \longleftrightarrow^* s$.
  - If there is any relation $\longleftrightarrow'$ with the above properties, then $s \longleftrightarrow^* t$ implies $s \longleftrightarrow' t$.
- The next slides contain a proof that $\longleftrightarrow^*$ is indeed the reflexive, symmetric and transitive closure of $\longleftrightarrow$. We jump over it. Jump over proof.

Proof

- We show that $\longleftrightarrow^*$, as defined originally, is in fact the reflexive, symmetric, and transitive closure of $\longrightarrow$:
  - That it contains $\longrightarrow$ follows since $r \longrightarrow s$ implies $r \longleftrightarrow s$ by an argument similar to that for $\longrightarrow^*$.
  - That it is reflexive and transitive follows as for $\longrightarrow^*$.
  - $\longleftrightarrow^*$ is symmetric: Assume $r \longleftrightarrow^* s$. Then there exist n, r_i, s.t.

$$r \equiv r_0 \longleftrightarrow r_1 \longleftrightarrow \cdots \longleftrightarrow r_n \equiv s$$

Now $r_i \longleftrightarrow r_{i+1}$ implies $r_{i+1} \longleftrightarrow r_i$, therefore

$$s \equiv r_n \longleftrightarrow r_{n-1} \longleftrightarrow \cdots \longleftrightarrow r_0 \equiv r$$

which implies $s \longleftrightarrow^* r$.

Proof (Cont.)

- Furthermore, if $\longleftrightarrow'$ is another relation which contains $\longleftrightarrow$ and which is reflexive, symmetric and transitive then it contains $\longleftrightarrow^*$:
  - Assume $\longleftrightarrow'$ having these properties.
  - Assume $r \longleftrightarrow^* s$.
  - Then there exist $n \in \mathbb{N}, r_i$ s.t.

$$r \equiv r_0 \longleftrightarrow r_1 \longleftrightarrow \cdots \longleftrightarrow r_n \equiv s$$

  - In case $n = 0$ we have $r \equiv s$, therefore $r \longleftrightarrow' s$.

Proof (Cont.)

$r \equiv r_0 \longleftrightarrow r_1 \longleftrightarrow \cdots \longleftrightarrow r_n \equiv s$

- In case $n > 0$ we first note that, since $\longleftrightarrow'$ contains $\longrightarrow$ and is symmetric, we have that $t \longrightarrow t'$ implies $t \longleftrightarrow' t'$ and $t' \longleftarrow t$ implies $t \longleftrightarrow' t'$.
- Therefore $t \longleftrightarrow t'$ implies $t \longleftrightarrow' t'$, and we obtain by the above

$$r \equiv r_0 \longleftrightarrow' r_1 \longleftrightarrow' \cdots \longleftrightarrow' r_n \equiv s$$

- But since $\longleftrightarrow'$ is transitive, it follows

$$r \longleftrightarrow' s$$

and we are done.

Identification of Elements

- If we have a reduction system $(\mathrm{T}, \longrightarrow)$, one writes

$$s =_{\rightarrow} t$$

or sometimes

$$s = t$$

for $s \longleftrightarrow^* t$.

- In order to avoid confusion, we write

$$s \equiv t$$

for s and t are the same element of T without using any reductions.

Determination of $\longleftrightarrow^*$

- In general it is infeasible to determine whether $s \longleftrightarrow^* t$ holds.
  - One has to check all possible ways of getting from s to t, by both using $\longrightarrow$ and $\longleftarrow$.
- In many cases this can be determined by:
  - Reducing s to some term s' s.t. $s \longrightarrow^* s'$ and s' has no further reductions,
    - i.e. by "evaluating s".
  - Doing the same with t to some term t'.
  - Checking whether s' is identical to t'.
- This way of determining, whether $s \longrightarrow^* t$ holds, is correct, if $\longrightarrow$ is confluent and strongly normalising (see next subsection).

(c) Termination, Confluence, Normalisation

Strong Normalisation

- A reduction system $(\mathrm{T}, \longrightarrow)$ is **terminating** or **strongly normalising**, iff there is no infinite sequence

$$s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow \cdots$$

of elements in T.

Examples

- The following reduction system is terminating:

$$0 \longleftarrow 1 \longleftarrow 2 \longleftarrow 3 \longleftarrow 4 \longleftarrow \cdots$$

  Any reduction sequence will end in 0 and terminate.

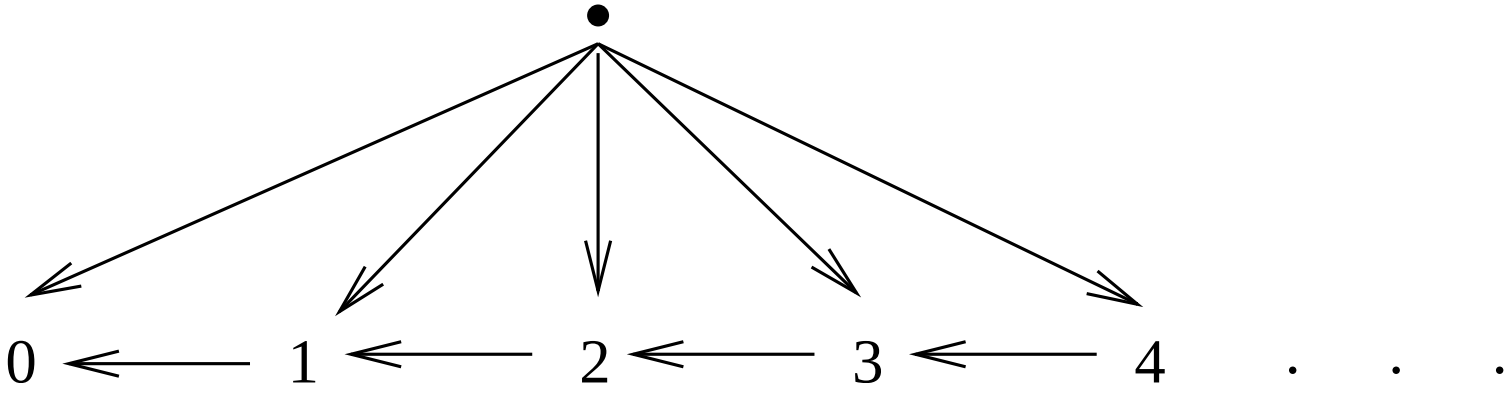
- The following reduction system is non terminating:

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 4 \longrightarrow \cdots$$

  (Take $0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow \cdots$).

Examples

- The following reduction system is terminating, but there are arbitrarily long reduction sequences starting with $\bullet$:



We have $\bullet \longrightarrow n \longrightarrow (n-1) \longrightarrow (n-2) \longrightarrow \cdots \longrightarrow 0$.

Examples

- The untyped λ-calculus (See next Sect.) is non terminating, since we have for $\Omega := (\lambda x.x\ x)\ (\lambda x.x\ x)$

$$\Omega \longrightarrow \Omega \longrightarrow \Omega \longrightarrow \cdots$$

- The typed λ-calculus (see later Subsection) is terminating.
  - In fact, the typed λ-calculus was introduced in order to obtain a terminate subtheory of the λ-calculus.

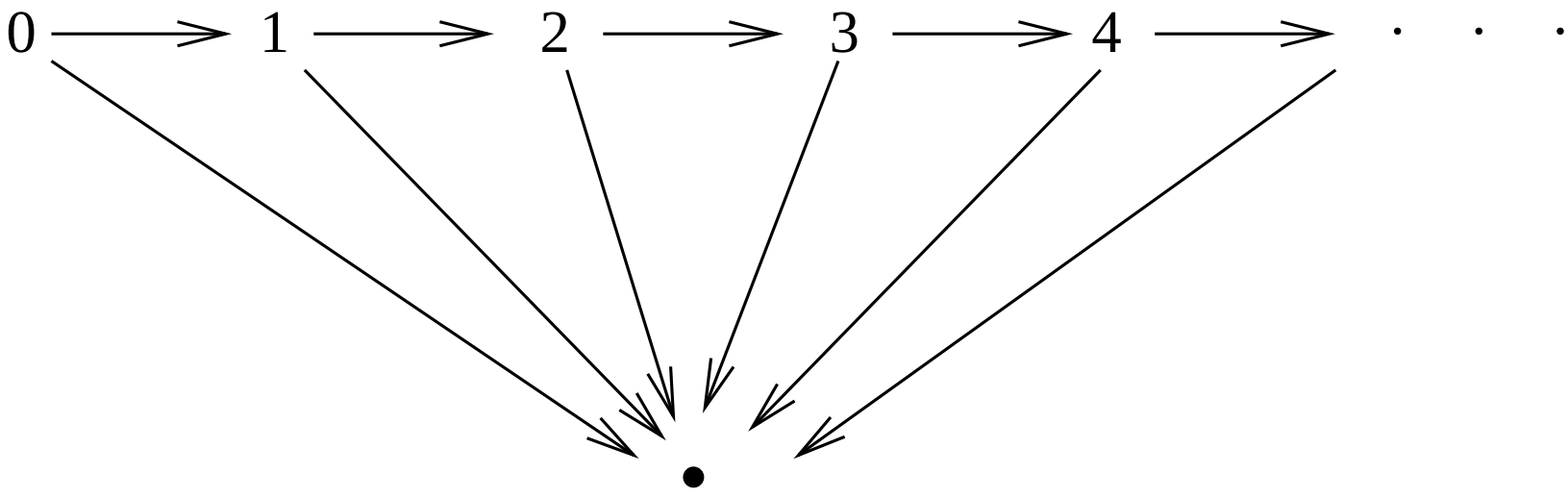
Normal Form and Irreducibility

Let $(\mathrm{T}, \longrightarrow)$ be a reduction system.

- $s \in \mathrm{T}$ is **irreducible**, if there exists no $t \in \mathrm{T}$ s.t. $s \longrightarrow t$.
- t is a **normal form of s** iff $s \longrightarrow^* t$ and t is irreducible.
- $(\mathrm{T}, \longrightarrow)$ is **weakly normalising** or **normalising**, if every $s \in \mathrm{T}$ has a normal form.

Example

The following system is weakly normalising, but not strongly normalising:

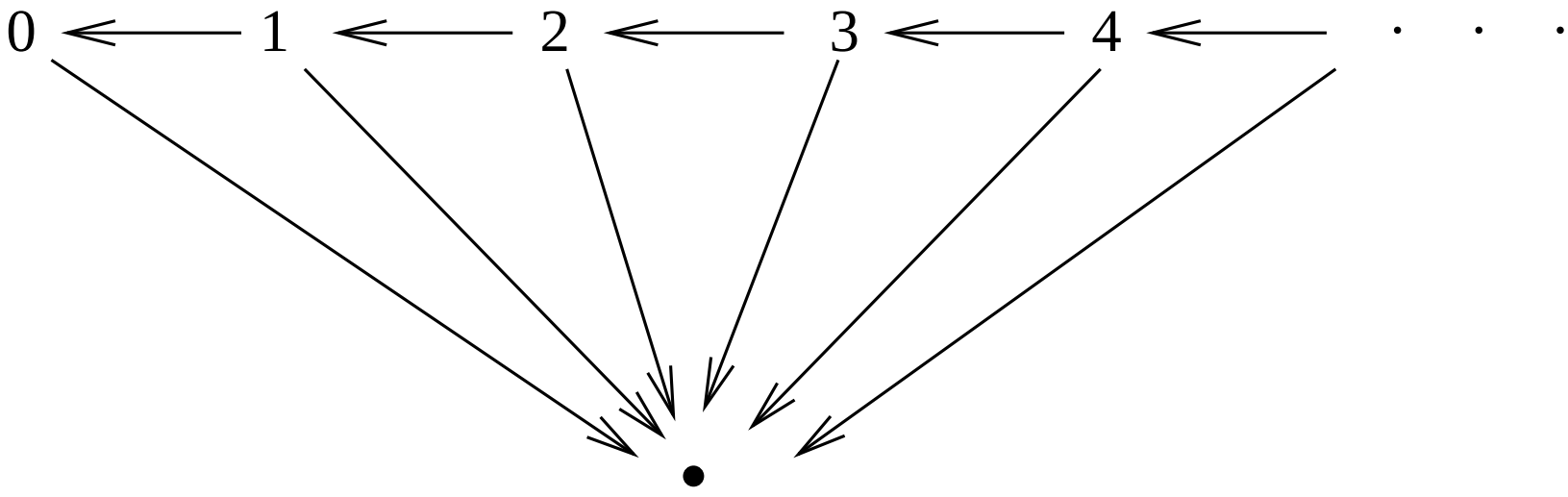


- Every r has a normal form, namely $\bullet$.
- But there exists an infinite sequence

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow \cdots$$

Example 2

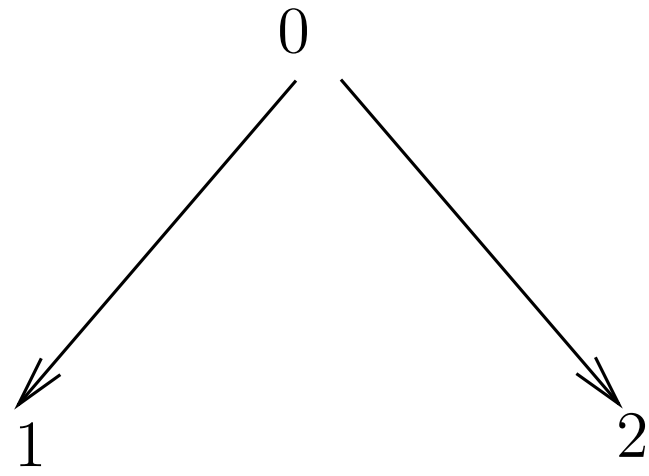
The following system is both weakly normalising and strongly normalising:



- If one reduces any element as long as possible, one finally ends up with •, which doesn't reduce any further.
- So every element has the same normal form namely •.

Example 3

In the following system 0 has two normal forms, namely 1 and 2:



This system is both strongly and weakly normalising, but is not confluent. (“Confluent” will be defined later).

Lemma

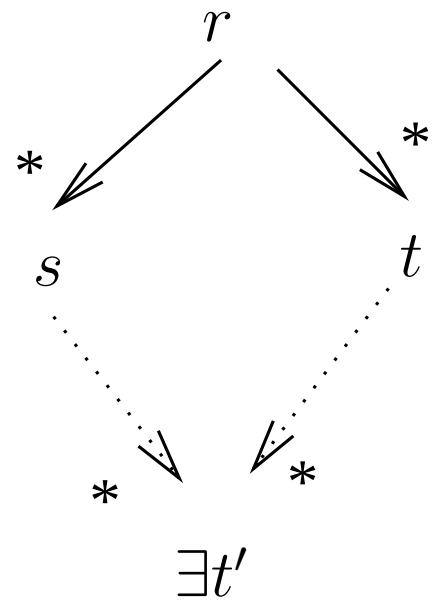
Let $(\mathrm{T}, \longrightarrow)$ be a strongly normalising reduction system.
Then $(\mathrm{T}, \longrightarrow)$ is weakly normalising.

Proof:
A normal form of $s \in \mathrm{T}$ can be obtained by simply reducing s as long as possible:
Since $(\mathrm{T}, \longrightarrow)$ is strongly normalising, the reduction sequence terminates in some $t \in \mathrm{T}$.

t is a normal form of s.

Church-Rosser

- We say a reduction system $(\mathrm{T}, \longrightarrow)$ is **confluent** or has the **Church-Rosser property** iff for all $r, s, t \in \mathrm{T}$ we have
  - if $r \longrightarrow^* s$ and $r \longrightarrow^* t$,
  - then there exists an t' s.t. $s \longrightarrow^* t'$ and $t \longrightarrow^* t'$.

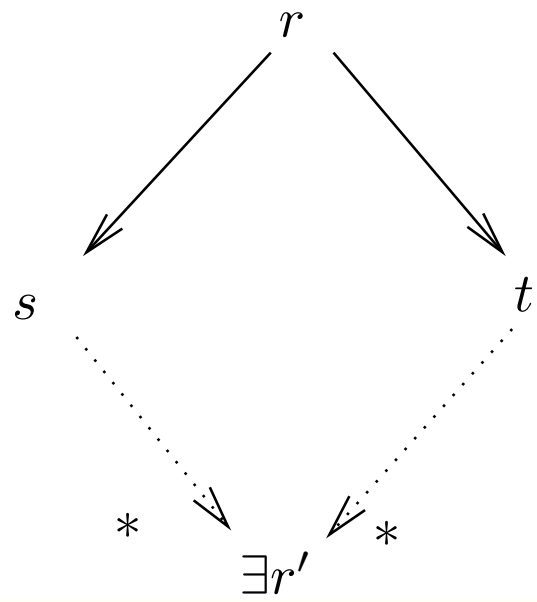


Diamond Property

- Because of the shape of the picture on the previous slide, the Church-Rosser property is sometimes called as well the
  - **Diamond property** or
  - **Triangle property**.
- So Church-Rosser means:
  Every triangle (or better fork) can be closed to a diamond.

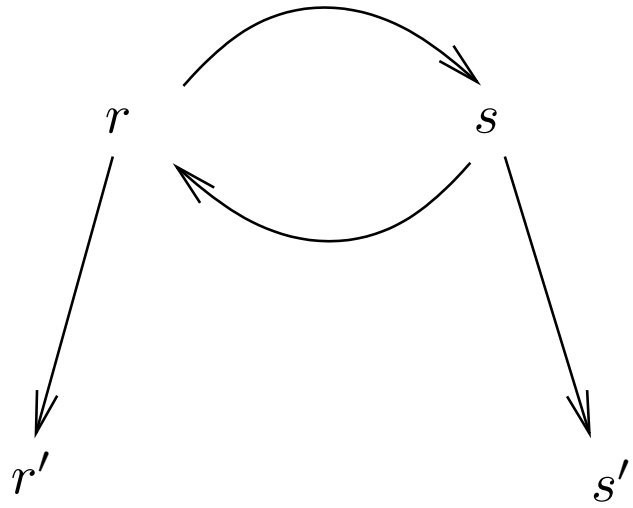
Weakly Church-Rosser

- One might think that a weaker version of Church-Rosser suffices:
  - If $r \longrightarrow s$ and $r \longrightarrow t$ then there exists an r' s.t. $s \longrightarrow^* r'$ and $t \longrightarrow^* r'$.
  - So we demand only that r reduces in **one** step to s and in one step to t.



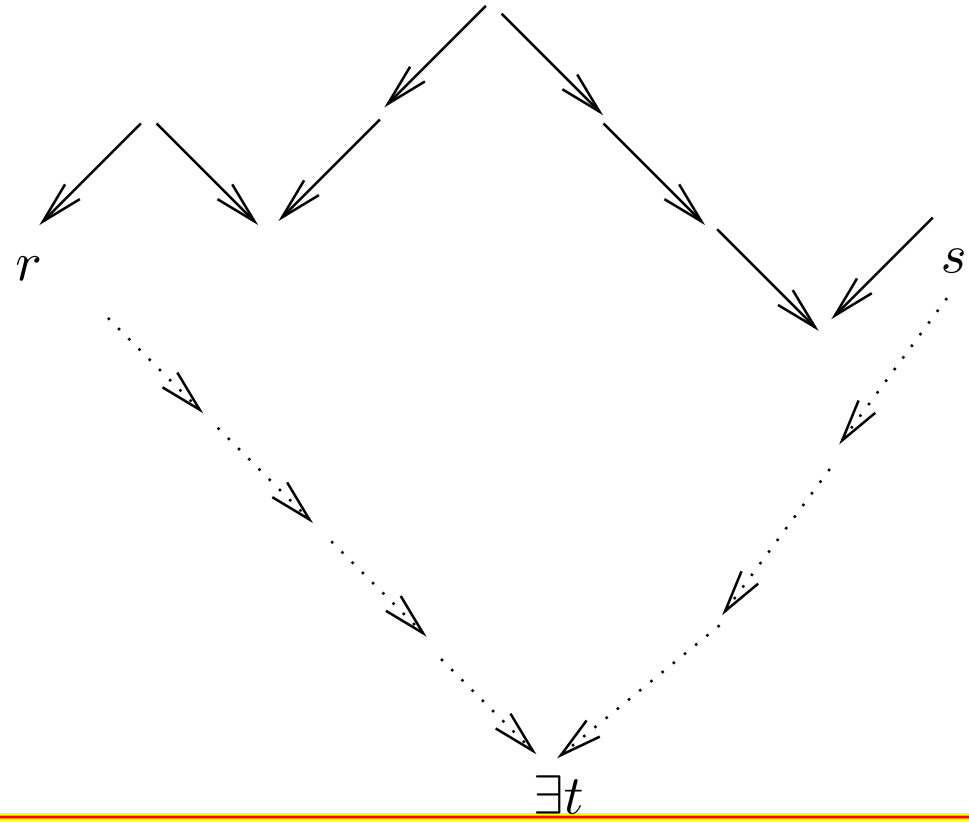
Weak Church-Rosser

- But that condition is weaker than full Church-Rosser.
- The following term rewriting system is weakly Church-Rosser, but doesn't fulfil the full Church-Rosser Property:
  - $r \longrightarrow^* r'$, $r \longrightarrow^* s'$ but there is no t s.t. $r' \longrightarrow^* t$ and $s' \longrightarrow^* t$.

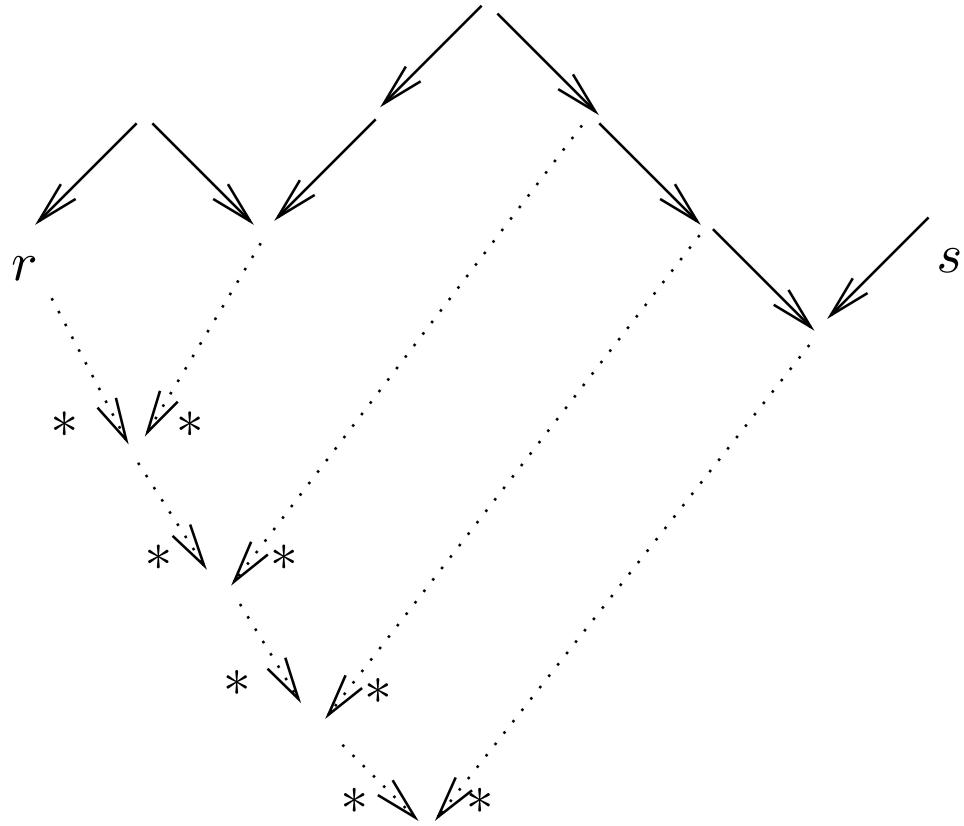


Theorem

- If $(\mathrm{T}, \longrightarrow)$ is confluent, then we have for $r, s \in \mathrm{T}$:
  $r \longleftrightarrow^* s$
  iff there exists a $t \in \mathrm{T}$ s.t.
  $r \longrightarrow^* t \wedge s \longrightarrow^* t$



Idea of Proof



Common Reduct of r and s

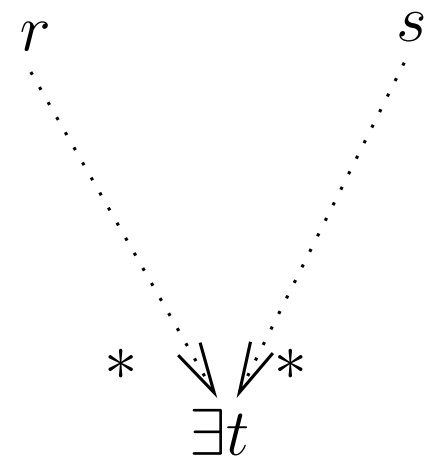
Jump over rest of proof

Proof of the Theorem

- We define for $r, s \in \mathrm{T}$

$$r \downarrow s :\Leftrightarrow \exists t \in \mathrm{T}.(r \longrightarrow^* t \wedge s \longrightarrow^* t) \quad .$$

- So $r \downarrow s$ means that r and s have a common reduct:



Proof of the Theorem

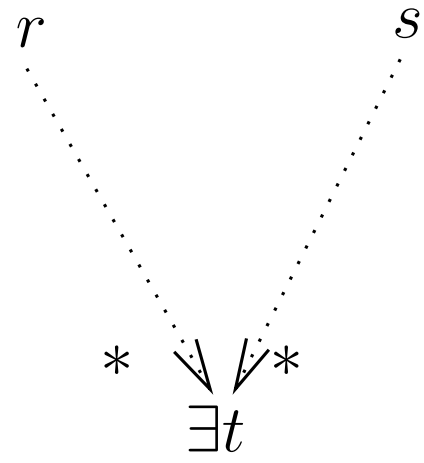
- So we have to show

$$r \longleftrightarrow^* s \Leftrightarrow r \downarrow s$$

- "$\Leftarrow$" is easy. If $r \longrightarrow^* t$, $s \longrightarrow^* t$, then we get

$$r \longrightarrow^* t \ {}^*\!\longleftarrow s$$

(where $t \ {}^*\!\longleftarrow s :\Leftrightarrow s \longrightarrow^* t$) and therefore $r \longleftrightarrow^* s$



Proof of the Theorem

- For the more difficult direction "$\Rightarrow$" we give two proofs:
  - One more concrete and intuitive one.
    - Will be presented during the lecture.
  - One more abstract one.
    - Will not be presented during the lecture.

First Proof of "$\Rightarrow$"

- Assume $r \longleftrightarrow^* s$.
- This means that we have a chain

$$r \equiv r_0 \longleftrightarrow r_1 \longleftrightarrow r_2 \longleftrightarrow \cdots \longleftrightarrow r_n \equiv s$$

- We are going to show successively:
  - $r_0 \downarrow r_0$,
  - $r_0 \downarrow r_1$,
  - $r_0 \downarrow r_2$,
  - $\cdots$
  - $r_0 \downarrow r_n \equiv s$ (the assertion).

First Proof of "$\Rightarrow$"

$r \downarrow s :\Leftrightarrow \exists t.(r \longrightarrow^* t \wedge s \longrightarrow^* t)$.

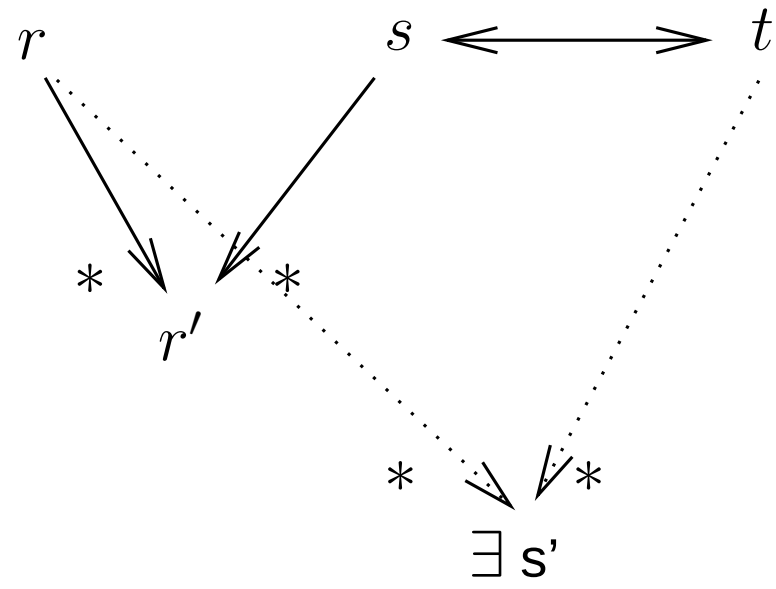
- In order to show this we have to show the following:
  - For the first step, we need to show $r_0 \downarrow r_0$, i.e. in general we need to show

$$(1) \qquad r \downarrow r$$

  - For the step from $r_0 \downarrow r_i$ to $r_0 \downarrow r_{i+1}$ we need to show:
    - If $r_0 \downarrow r_i$ and $r_i \leftrightarrow r_{i+1}$ then $r_0 \downarrow r_{i+1}$.
    - In general we have to show:
      If $r \downarrow s$ and $s \longleftarrow t$, then $r \downarrow t$,
      in short:
      r $\downarrow$ s $\longleftarrow$ t implies r $\downarrow$ t.

Diagram

- That $r \downarrow s \longleftrightarrow t$ implies $r \downarrow t$ can be visualised as follows:



First Proof of "$\Rightarrow$"

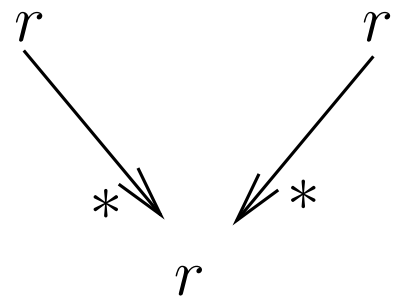
- (1) $r \downarrow r$.
- $r \downarrow s \longleftrightarrow t$ implies $r \downarrow t$.
  - Since $s \longleftrightarrow t$ means $s \longrightarrow t$ or $s \longleftarrow t$, we need to show

$$(2) \qquad r \downarrow s \longrightarrow t \;\Rightarrow\; r \downarrow t \;,$$
$$(3) \qquad r \downarrow s \longleftarrow t \;\Rightarrow\; r \downarrow t \;,$$

- So in total we have to show (1), (2), (3) above.

(1) $r \downarrow r$

- We show $r \downarrow r$.

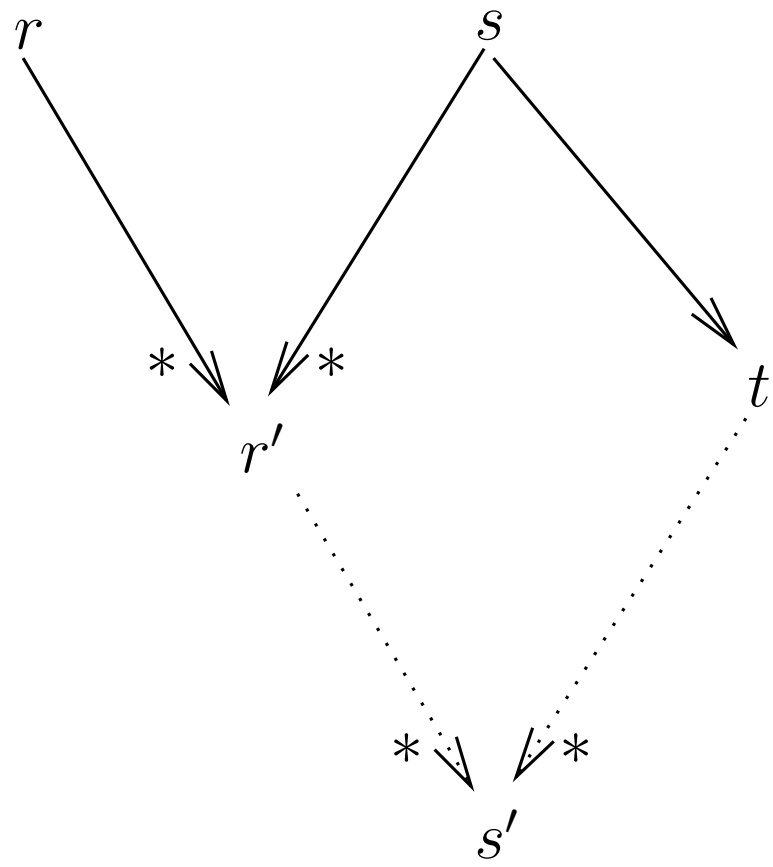


Formally: We have $r \longrightarrow^* r$ and $r \longrightarrow^* r$, therefore $r \downarrow r$.

(2) $r \downarrow s \longrightarrow t$ implies $r \downarrow t$

- Assume $r \downarrow s$, $s \longrightarrow t$. Show $r \downarrow t$.
- $r \longrightarrow^* r'$, $s \longrightarrow^* r'$ for some r'.
- By Church-Rosser, $s \longrightarrow^* r'$ and $s \longrightarrow^* t$ implies that there exists a s' s.t. $r' \longrightarrow^* s'$, $t \longrightarrow^* s'$.
- But then
  - $r \longrightarrow^* r' \longrightarrow^* s'$ therefore $r \longrightarrow^* s'$,
  - $t \longrightarrow^* s'$,
  - therefore $r \downarrow t$.

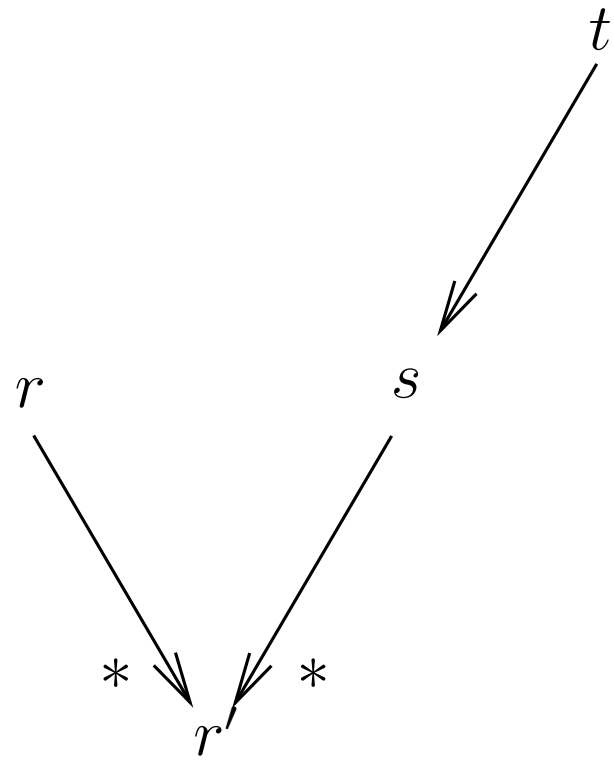
(2) $r \downarrow s \longrightarrow t$ implies $r \downarrow t$



(3) $r \downarrow s \leftarrow t$ implies $r \downarrow t$

- Assume $r \downarrow s$, $s \longleftarrow t$. Show $r \downarrow t$.
- $r \longrightarrow^* r'$, $s \longrightarrow^* r'$ for some r'.
- But then
  - $r \longrightarrow^* r'$
  - $t \longrightarrow^* s \longrightarrow^* r'$,
  - therefore $r \downarrow t$.

$r \downarrow s \leftarrow t$ **implies** $r \downarrow t$



This completes the first proof of the Theorem.

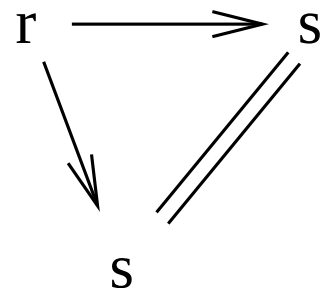
Second Proof of "$\Rightarrow$"

- We show that $\downarrow$ contains $\longrightarrow$ and is reflexive, symmetric and transitive.
- By definition, $\longleftrightarrow$ is the least reflexive, symmetric and transitive relation which contains $\longrightarrow$.
- Therefore $\longleftrightarrow$ is contained in $\downarrow$ and we obtain

$$r \longleftrightarrow^* s \Rightarrow r \downarrow s$$

Second Proof of "$\Rightarrow$"

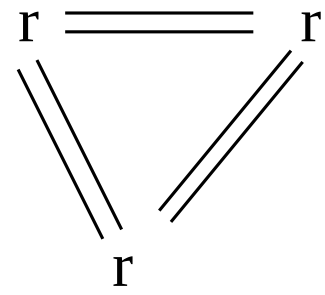
- $\downarrow$ contains $\longrightarrow$:
  $r \longrightarrow s$ implies $r \downarrow s$.



Formally: If $r \longrightarrow s$ then we have with $t := s$ that $r \longrightarrow^* t$ and $s \longrightarrow^* t$.

Second Proof of "$\Rightarrow$"

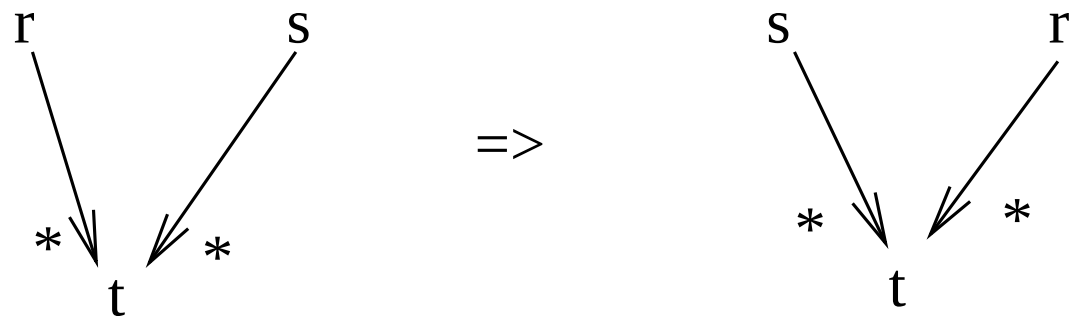
- $\downarrow$ is reflexive. (As in the previous proof).



Formally: We have $r \longrightarrow^* r$ and $r \longrightarrow^* r$, therefore $r \downarrow r$.

Second Proof of "$\Rightarrow$"

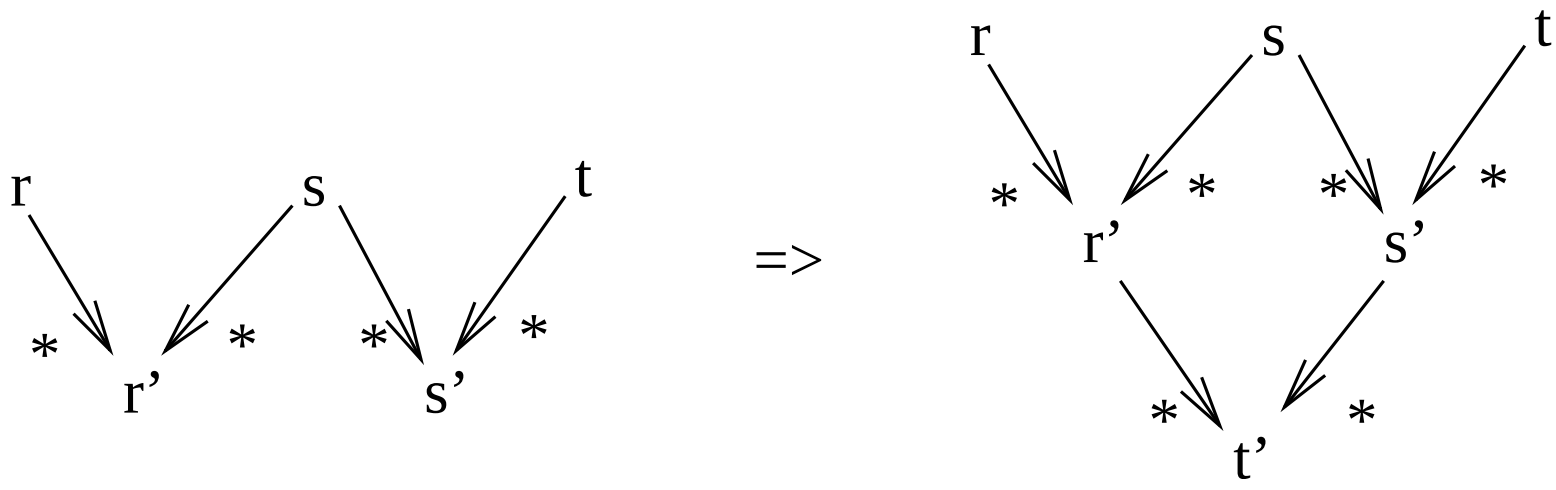
- $\downarrow$ is symmetric:



Formally:
Assume $r \downarrow s$.
Then $r \longrightarrow^* t$ and $s \longrightarrow^* t$ for some t.
Then $s \longrightarrow^* t$ and $r \longrightarrow^* t$.
Therefore $s \downarrow r$.

Second Proof of "$\Rightarrow$"

- $\downarrow$ is transitive:



Second Proof of "$\Rightarrow$"

- ($\downarrow$ is transitive:)
  Formally:
  - Assume $r \downarrow s$ and $s \downarrow t$.
    Then there exists r', s' s.t. $r \longrightarrow^* r'$, $s \longrightarrow^* r'$, $s \longrightarrow^* s'$, $t \longrightarrow^* s'$.
  - Then by confluence there exists an t' s.t. $r' \longrightarrow^* t'$, $s' \longrightarrow^* t'$.
  - Then $r \longrightarrow^* t'$ and $t \longrightarrow^* t'$.
  - Therefore $r \downarrow t$.

Unique Normal Forms

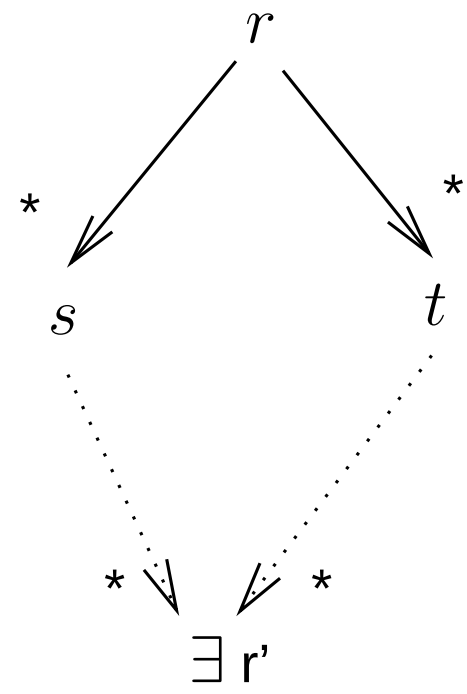
Lemma:
Let $(\mathrm{T}, \longrightarrow)$ be a confluent reduction system.
If $r \in \mathrm{T}$ has a normal form s, then it is unique:

- If t is another normal form, then $s \equiv t$.

Proof:

- We have $r \longrightarrow^* s$ and $r \longrightarrow^* t$.
- By confluence, there exists a r' s.t. $s \longrightarrow^* r'$ and $t \longrightarrow^* r'$.
- But since s and t are normal forms, it follows $s \equiv r'$ and $t \equiv r'$.

Picture



Since s, t are in normal form, $s \equiv r' \equiv t$

Lemma

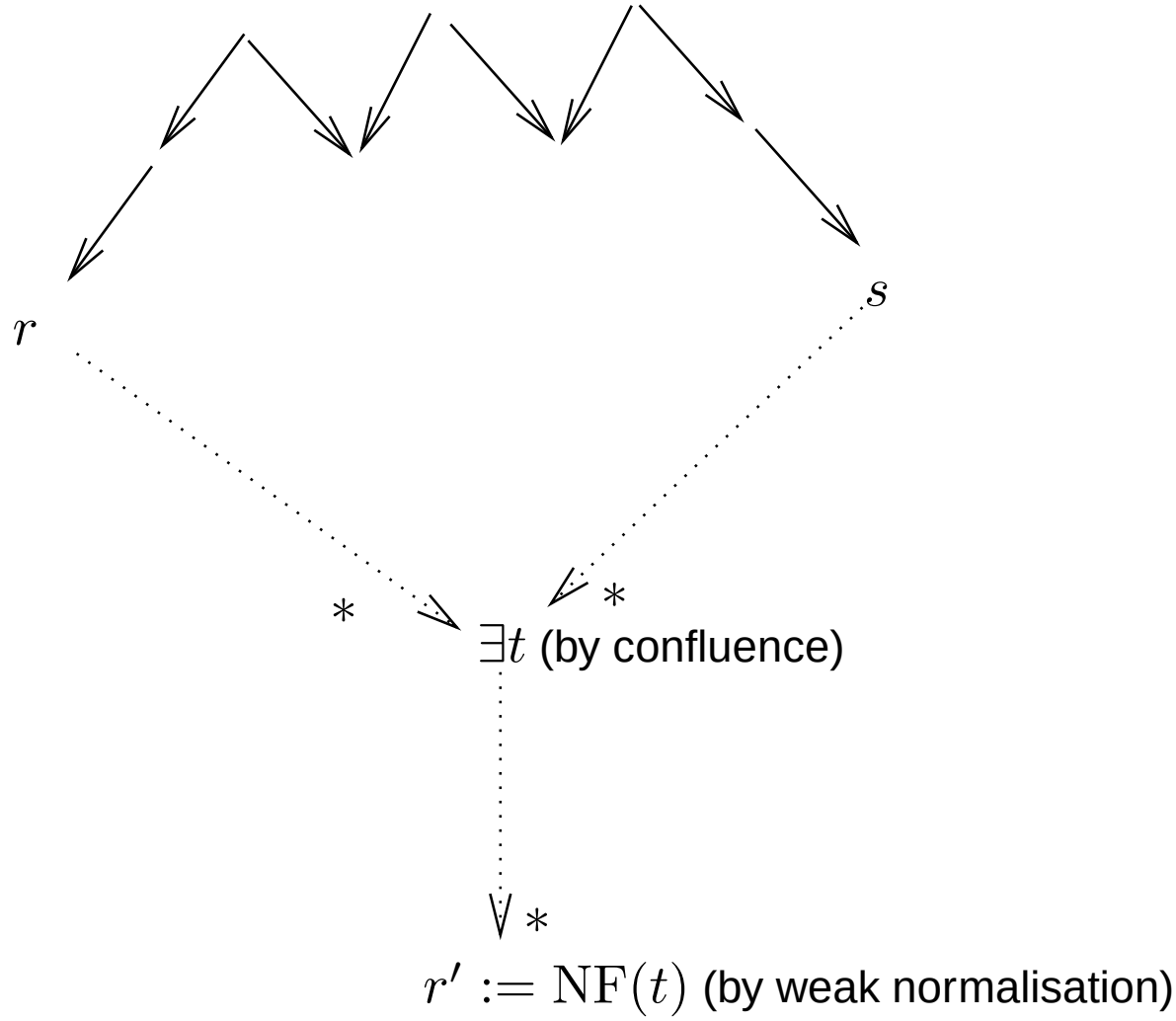
Let $(\mathrm{T}, \longrightarrow)$ be a weakly normalising and confluent reduction system. Then

- $r \longleftrightarrow^* s$ iff the normal forms of r and s coincide.

Proof:

- "$\Rightarrow$": By Church Rosser $r \longleftrightarrow^* s$ implies the existence of a t s.t. $r \longrightarrow^* t$ and $s \longrightarrow^* t$.
  Reduce t further to a normal form r'.
  Then r' is a normal form of both r and s as well.
  Since by the above lemma, normal forms are unique, r' is the normal form of r and s.

Picture (Proof of "$\Rightarrow$")



Lemma

- "$\Leftarrow$": If the normal forms t coincide, then we have $r \longrightarrow^* t \longleftarrow^* s$, therefore $r \longleftrightarrow^* s$.

Remark on Agda

- The underlying reduction system of Agda is strongly normalising and confluent, provided the code has been termination checked.
- The equality derived from this reduction system is used in order to typecheck terms.

(d) Term Rewriting Systems

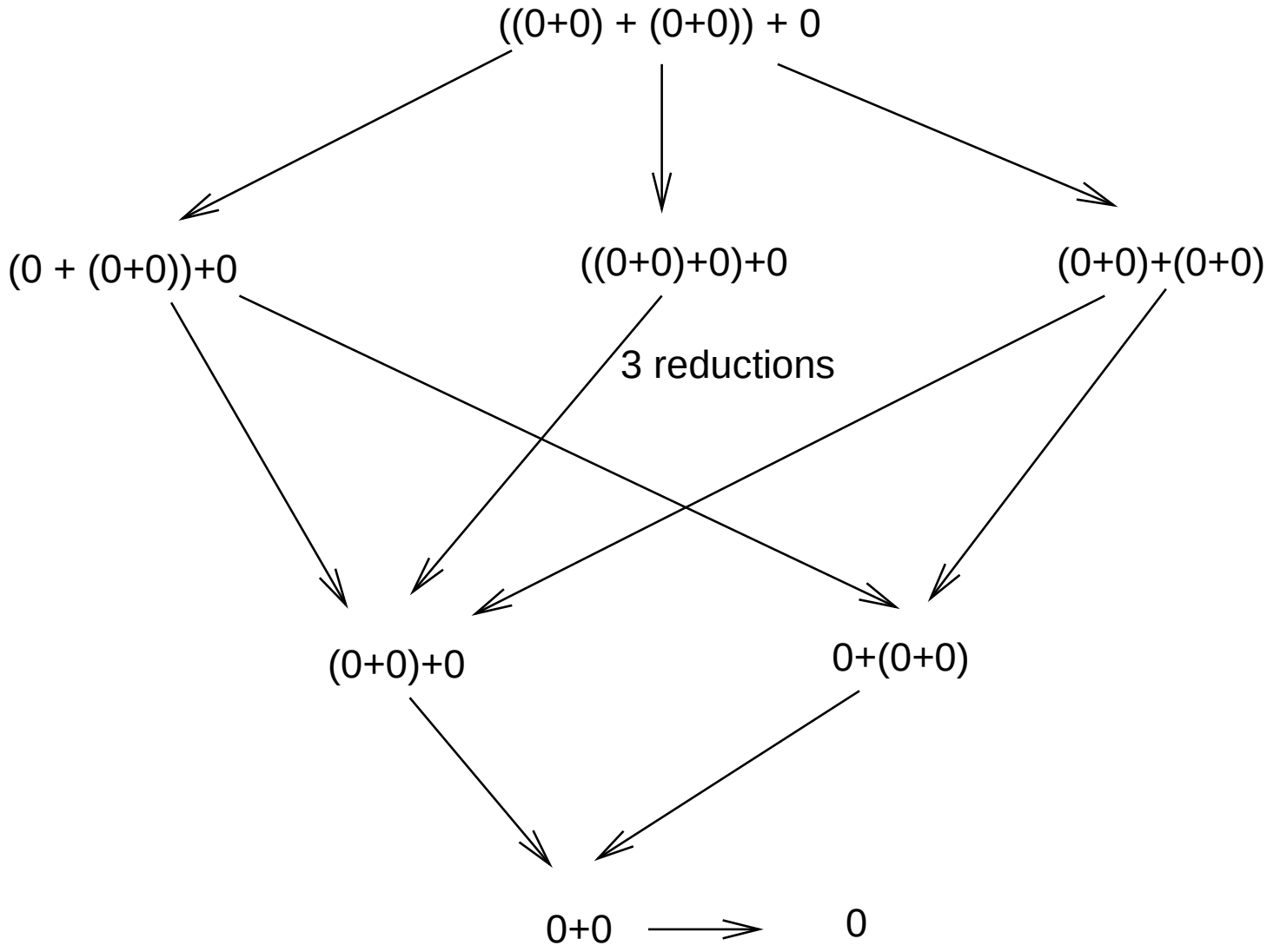
- Term rewriting systems are special cases of reduction systems.
- They are reduction systems, which are generated by a (in many cases finite) set of rules (i.e. basic reductions).

Example of a Term Rewriting Syste

- Take $T =$ set of arithmetic expressions formed from variables, 0 by using the successor operation S (where $\mathrm{S}\ n$ stands $n + 1$), $+$, $*$ and brackets.
  - So the following are elements of T:
    - $x + \mathrm{S}\ 0$,
    - $\mathrm{S}\ 0 + z * (\mathrm{S}\ (\mathrm{S}\ x) + 0)$,
    - $\mathrm{S}\ y * \mathrm{S}\ 0 + \mathrm{S}\ x * 0$.
- Take as rules the following:

$$
\begin{aligned}
x + 0 &\longrightarrow_{\mathrm{Rule}} x \ , \\
x + \mathrm{S}\ y &\longrightarrow_{\mathrm{Rule}} \mathrm{S}\ (x + y) \ , \\
x * 0 &\longrightarrow_{\mathrm{Rule}} 0 \ , \\
x * \mathrm{S}\ y &\longrightarrow_{\mathrm{Rule}} x * y + x \ .
\end{aligned}
$$

Example Reductions



Example of a Term Rewriting Syste

- (The system will be in fact strongly normalising and confluent).

Term Rewriting Systems

$$
\begin{aligned}
x + 0 &\longrightarrow_{\text{Rule}} x \ , \\
x + \mathrm{S}\ y &\longrightarrow_{\text{Rule}} \mathrm{S}\ (x + y) \ , \\
x * 0 &\longrightarrow_{\text{Rule}} 0 \ , \\
x * \mathrm{S}\ y &\longrightarrow_{\text{Rule}} x * y + x \ .
\end{aligned}
$$

- The reduction relation generated by these rules allows to replace in a term
  - any subterm of the form $s + 0$ by s,
  - any subterm of the form $s + \mathrm{S}\ t$ by $\mathrm{S}\ (s + t)$,
  - any subterm of the form $s * 0$ by 0,
  - any subterm of the form $s * \mathrm{S}\ t$ by $s * t + s$.

Term Rewriting Systems

$$
\begin{aligned}
x + 0 &\longrightarrow_{\text{Rule}} x \ , \\
x + \mathrm{S}\ y &\longrightarrow_{\text{Rule}} \mathrm{S}\ (x + y) \ , \\
x * 0 &\longrightarrow_{\text{Rule}} 0 \ , \\
x * \mathrm{S}\ y &\longrightarrow_{\text{Rule}} x * y + x \ .
\end{aligned}
$$

- So we have for instance the following reductions:
  - $0 + \mathrm{S}\ (\mathrm{S}\ 0) \longrightarrow \mathrm{S}\ (0 + \mathrm{S}\ 0)$,
    - Reduce $0 + \mathrm{S}\ s$ to $\mathrm{S}\ (0 + s)$ using $s \equiv \mathrm{S}\ 0$.
  - $\mathrm{S}\ (0 + \mathrm{S}\ 0) \longrightarrow \mathrm{S}\ (\mathrm{S}\ (0 + 0))$,
    - Reduce $s + \mathrm{S}\ t$ to $\mathrm{S}\ (s + t)$, using $s \equiv t \equiv 0$,
  - $\mathrm{S}\ (\mathrm{S}\ (0 + 0)) \longrightarrow \mathrm{S}\ (\mathrm{S}\ 0)$.

Definition of Term Rewriting Systen

- A term rewriting system consists of
  - a set of terms T built from variables, constants and some function symbols,
  - a relation $\longrightarrow_{\mathrm{Rule}}$ between terms
    - (if $r \longrightarrow_{\mathrm{Rule}} s$ we say hat $r \longrightarrow_{\mathrm{Rule}} s$ is a **rule**),
  - s.t., if $s \longrightarrow_{\mathrm{Rule}} t$, then
    - s is not a variable, and
    - all variables in t occur in s.
- The variable conditions is needed so that the theory of term rewriting systems goes through smoothly.
  - This is not important for this lecture, and therefore the explanation will be omitted.
    Jump over explanation.

Condition on Variables

- In the previous definition we demanded two variable conditions for $s \longrightarrow_{\mathrm{Rule}} t$:
  - s is not a variable.
    - If we allowed s to be a variable say x, then the rule would have the form $x \longrightarrow t$.
      That would mean that any term r has a reduction, namely to $t[x := r]$.
  - All variables in t occur in s.
    - Assume y were a variable in t but not in s.
      If we substitute in s and t all variables from s by closed terms, and obtain s' and t' then we would have that s' would have potentially infinitely many reductions, namely for any substitution of the other variables of t' by closed terms.

Condition on Variables

- The second variable condition has something to do with determinism:
  - Assume we have chosen a rule $r \longrightarrow s$ and chosen a substitution of variables in r, which matches a term t.
  - Then the reduct with respect to this rule is uniquely determined.
    - There are no other free variables in s which allow additional choices for substitutions.

Condition on Variables

- Both these case would cause problems in the theory of term-rewriting systems (we won't touch those problems).

Reduction generated by $\longrightarrow_{\mathrm{Rule}}$

- If we have a term rewriting system $(T, \longrightarrow_{\mathrm{Rule}})$ we obtain a reduction relation $\longrightarrow$ on T as follows:
  - First we construct a relation $\longrightarrow'$ obtained from reductions rules $r \longrightarrow_{\mathrm{Rule}} r'$ by substituting the variables in both r and r' by some terms.
    - So the same substitutions are carried out in both r and r'.
    - If $s \longrightarrow' s'$ is obtained by carrying out such a substitution in $r \longrightarrow_{\mathrm{Rule}} r'$, then $s \longrightarrow' s'$ is called **an instance of rule** $r \longrightarrow_{\mathrm{Rule}} r'$.

Example (Instance of a Rule)

$$
\begin{aligned}
x + 0 &\longrightarrow_{\mathrm{Rule}} x \ , \\
x + \mathrm{S}\ y &\longrightarrow_{\mathrm{Rule}} \mathrm{S}\ (x + y) \ , \\
x * 0 &\longrightarrow_{\mathrm{Rule}} 0 \ , \\
x * \mathrm{S}\ y &\longrightarrow_{\mathrm{Rule}} x * y + x \ .
\end{aligned}
$$

- $0 + 0 \longrightarrow' 0$ is an instance, obtained by substituting in $x + 0 \longrightarrow_{\mathrm{Rule}} x$ the variable x by 0.
- $\mathrm{S}\ 0 * \mathrm{S}\ 0 \longrightarrow' \mathrm{S}\ 0 * 0 + \mathrm{S}\ 0$
  is an instance, obtained by substituting in $x * \mathrm{S}\ y \longrightarrow_{\mathrm{Rule}} x * y + x$ the variable x by $\mathrm{S}\ 0$ and the variable y by 0.

Reduction generated by $\longrightarrow_{\mathrm{Rule}}$

- Then $s \longrightarrow s'$, if there exists an instance $t \longrightarrow' t'$ of a rule s.t. s contains subterm t, and s' is the result of substituting in s the term t by t'.
  - The subterm s is called a **redex** w.r.t. the term rewriting system used.
    - "Redex" is short for **reducible expression**.
    - Plural of redex is **redexes**.
- The reductions $s \longrightarrow s'$ obtained this way are the reductions **generated** by the term rewriting system.

Example 1

$$
\begin{aligned}
x + 0 &\longrightarrow_{\mathrm{Rule}} x \ , \\
x + \mathrm{S}\ y &\longrightarrow_{\mathrm{Rule}} \mathrm{S}\ (x + y) \ , \\
x * 0 &\longrightarrow_{\mathrm{Rule}} 0 \ , \\
x * \mathrm{S}\ y &\longrightarrow_{\mathrm{Rule}} x * y + x \ .
\end{aligned}
$$

- $0 + \mathrm{S}\ (\mathrm{S}\ 0) \longrightarrow \mathrm{S}\ (0 + \mathrm{S}\ 0)$ is obtained as follows:
  - The rule used is

$$x + \mathrm{S}\ y \longrightarrow_{\mathrm{Rule}} \mathrm{S}\ (x + y) \ .$$

  - By substituting x by 0 and y by $\mathrm{S}\ 0$ we obtain the instance

$$0 + \mathrm{S}\ (\mathrm{S}\ 0) \longrightarrow' \mathrm{S}\ (0 + \mathrm{S}\ 0) \ .$$

  - In this example, the redex is the full term $0 + \mathrm{S}\ (\mathrm{S}\ 0)$ which is then reduced.

Example 2

$$
\begin{aligned}
x + 0 &\longrightarrow_{\mathrm{Rule}} x \ , \\
x + \mathrm{S}\ y &\longrightarrow_{\mathrm{Rule}} \mathrm{S}\ (x + y) \ , \\
x * 0 &\longrightarrow_{\mathrm{Rule}} 0 \ , \\
x * \mathrm{S}\ y &\longrightarrow_{\mathrm{Rule}} x * y + x \ .
\end{aligned}
$$

- $\mathrm{S}\ (0 + \mathrm{S}\ 0) \longrightarrow \mathrm{S}\ (\mathrm{S}\ (0 + 0))$ is obtained as follows:
  - The rule used is $x + \mathrm{S}\ y \longrightarrow_{\mathrm{Rule}} \mathrm{S}\ (x + y)$.
  - By substituting x and y by 0 we obtain the instance

$$0 + \mathrm{S}\ 0 \longrightarrow' \mathrm{S}\ (0 + 0) \ .$$

  - The left hand side of our reduction $\mathrm{S}\ (0 + \mathrm{S}\ 0)$ contains now the redex $0 + \mathrm{S}\ 0$.
  - By substituting it by $\mathrm{S}\ (0 + 0)$ we obtain the right hand side of the reduction, $\mathrm{S}\ (\mathrm{S}\ (0 + 0))$.

Example 3

$$
\begin{aligned}
x + 0 &\longrightarrow_{\mathrm{Rule}} x \ , \\
x + \mathrm{S}\, y &\longrightarrow_{\mathrm{Rule}} \mathrm{S}\, (x + y) \ , \\
x * 0 &\longrightarrow_{\mathrm{Rule}} 0 \ , \\
x * \mathrm{S}\, y &\longrightarrow_{\mathrm{Rule}} x * y + x \ .
\end{aligned}
$$

- $\mathrm{S}\, (\mathrm{S}\, (0 + 0)) \longrightarrow \mathrm{S}\, (\mathrm{S}\, 0)$.
  - The rule used is $x + 0 \longrightarrow_{\mathrm{Rule}} x$.
  - By substituting 0 for x, we obtain the instance

$$0 + 0 \longrightarrow' 0 \ .$$

  - The left hand side of the reduction $\mathrm{S}\, (\mathrm{S}\, (0 + 0))$ contains the redex $0 + 0$.
  - By substituting it by 0 we obtain the right hand side of the reduction $\mathrm{S}\, (\mathrm{S}\, 0)$.