0. Overview

(a) What is Interact. Theorem Proving

(b) Need for Theorem Proving

(c) Concentration on the latter in this module.

We need to prove theorems in order to establish mathematical theorems.
E.g. that certain problems are decidable, undecidable, polynomial computable etc.

We need them as well in order to establish the correctness of software and hardware.
Is floating point division for the Intel processor correct?
Is a railway control system safe?
When verifying the Swedish railway system, lots of bugs were found.
4 Ways of Proving Theorems

1. Theorem proving by hand.
   - What mathematicians do all the time.
   - Will remain in the near future the main way for proving theorems.
   - Problem: Errors.
     - As in programs after a certain amount of lines there is a bug, after a certain amount of lines a proof has a bug.
     - The problem can only be reduced by careful proof checking, but not eliminated completely.
   - Unsuitable for verifying large software and hardware systems.
   - Data usually too large.
   - Likely that one makes the same mistakes as in the software.

2. Theorem proving with some machine support.
   - Machine checks the syntax of the statements, creates a good layout, translates it into different languages.
   - Theorem proving still to be done by hand.
   - **Example:** most systems for specification of software (e.g. CSP-CASL, as used by Dr. Roggenbach).
   - **Advantages:**
     - Less errors.
     - User is forced to obey a certain syntax.
     - Specifications can be exchanged more easily.
   - **Disadvantages:** Similar to 1.

3. Interactive Theorem Proving.
   - Proofs are fully checked by the system.
   - Proof steps have to be carried out by the user.
   - **Advantages:**
     - Correctness guaranteed (provided the theorem prover is correct).
     - Everything which can be proved by hand, should be possible to be proved in such systems.
   - **Disadvantages:**
     - It takes much longer than proving by hand.
     - Similar to programming:
       - To say in words what a program should do, doesn't take long.
       - To write the actual program, can take a long time, since much more details are involved than expected.
   - Requires experts in theorem proving.
4 Ways of Proving Theorems

   - The theorem is shown by the machine.
   - It is the task of the user to
     state the theorem,
     bring it into a form so that it can be solved,
     usually adapt certain parameters so that the
     theorem proving solves the problem within
     reasonable amount of time.
   - Espec. Dr. Kullmann is an expert in this area.

4 Ways of Proving Theorems

(Automated theorem proving)

- **Advantages**
  - Less complicated to “feed the theorem into the
    machine” rather than actually proving it.
  - Might be done by non-specialists.
  - Sometimes faster than interactive theorem
    proving.

Disadvantages

- Many problems cannot be proved automatically.
- Can often deal only with finite problems.
  - We can show the correctness of one particular
    processor.
  - But we cannot show a theorem, stating the
    correctness of a parametric unit (like a generic
    \(n\)-bit adder for arbitrary \(n\)).
  - In some cases this can be overcome.
- Limits on what can be done (some hardware
  problems can be verified as 32 bit versions, but not
  as 64 bit versions).

(b) Administrative Issues

Address:
Dr. A. Setzer
Dept. of Computer Science
University of Wales Swansea
Singleton Park
SA2 8PP
UK

Room  Room 211, Faraday Building
Tel.   (01792) 513368
Fax.   (01792) 295651
Email
Assessment

- 80% Exam.
- 20% coursework:
  - 4 small assignments. Each counts 5% (Plan, might be changed).
  - Handed out approx. every 2nd week.
  - Due two weeks later.

(c) Plan

0.
1.
2.
3.
4. Interactive programs in dependent type theory.
5. Cayenne – A programming language with dependent types.

The plan might be changed slightly.

Timetable, Course Material

- Two lectures per week.
  - Tuesday, 11:00, Faraday J.
  - Thursday, 10:00, Vivian 112.
- Web page contains overhead slides from the lectures. Course material will be continually updated.

(d) Literature

- In general, the module is self-contained.
Main Course Literature


  Similar as the previous book, but shorter.

More advanced Books

  Contains some material of interest (e.g. BHK interpretation of logical connective). Postgraduate level.

  Book on postgraduate level. Deviates from “official Martin Lof type theory”.

Other Introductory Books

  Relatively easy short book, from the father of the type theory we are using. Intended for philosophers.

  Use of type theory in linguistics and for translation between languages. Has a good and simple introduction into type theory.

1. Introduction

(a)  
(b)  
(c)  
(d)  
(e)  
(f)  
(g)  
(a) Approaches to Verified Software

We consider 4 principal approaches towards writing verified software.

(i) **First a program is written. Then its correctness is verified.**

- Most common approach, when formal methods are applied.
- Main advantage: Ordinary programming languages can be used.
- Disadvantage: all or most considerations of the programmers are lost.
- Requires advanced automated theorem proving technologies.
- **Dr. Kullmann** is an expert on the theorem proving techniques used there.

(ii) **Prove that a solution for the problem exists. Extract a program from it.**

- E.g. from a proof of the statement

  \[
  \text{For every list there exists a sorted list having the same elements}
  \]

  one can extract a program, which computes from a list a sorted list having the same elements. The correctness is guaranteed.
- Technology not yet far developed.
- **Dr. Berger** is an expert in this area of research.

(iii) **Programs written in a language which allows to state properties of the program.**

Example: “This program sorts a list”. Properties should be verified when compiling the program

- **Advantages:**
  - Programmer is forced to think very clearly.
  - Programs will be very well documented.
  - The information about properties needed might guide the programmer.
  - In some cases parts of the program can even be found automatically.

**Disadvantages of (iii):**

- Requires new programming languages.
- Still essentially area of research. However advanced tools exist already.
- Might be too difficult for ordinary programmers.

**Effect:**

- Proving and programming will be the same.
Approaches to Verified Software

(iv) **Mixtures** between (i), (iii).
   - E.g. SPARK Ada.

In this lecture, we will follow the approach of (iii), based on dependent type theory.

---

(b) **The Theorem Prover Agda**

- There are several implementations of dependent type theory:
  - NuPrl (Cornell, USA), the technically most advanced system.
  - Uses so called “extensional type theory”.
  - Coq (INRIA, France), as well technically very advanced.
  - LEGO (Edinburgh), about to be replaced.
  - Both use so called “impredicative type theory”.

---

### The Theorem Prover Agda (Cont.)

- Implementations of dependent type theory (Cont.)
  - The “Alf-family” (Gothenburg, Sweden) – has probably the clearest concepts.
  - Alf (developed by Lena Magnusson)
  - Half (= Haskell Alf), developed by Thierry Coquand, Dan Synek.
  - Agda developed by Catarina Coquand.
  - Alfa, a graphical user interface for Agda, developed by Thomas Hallgren.
  - “Cayenne” (Gothenburg, Sweden) is a dependently typed programming language (not intended as a theorem prover).

- In this module we will use Agda and briefly consider Cayenne, but Alfa can be used to create Agda code.

---

Proofs in Agda

- Half, Agda, Alf are written in Haskell.
- Half and Agda have an Emacs mode, which makes it quite convenient to develop proofs in it.
- In most theorem provers, one has to follow one or several goals, and derive proofs for them. This is close to the way, proofs are carried out by hand.
Proofs in Agda (Cont.)

- The Alf-family has a different approach of **successive refinement**.
- One starts writing the proof code similarly to writing functional programs.
- What cannot be done without machine assistance can be left open in the form of holes (**goals**).
- Now one can successively, assisted by the system, fill in those goals.
- Therefore proof/program development in the Alf family is very close to ordinary programming.

Installation of Agda

- Agda is installed in the Linux lab.
  - Follow the item “Getting started with Agda” on the home page of this module.
  - Please check whether the installation works.
- Agda is most easily installed under **Linux** or other versions of Unix.
- It can as well be installed under CYGWIN, a UNIX emulation under Windows.
- See information from the course home page.
- The source code for the examples given in this lecture will be available from the course home page.

Thierry Coquand

The main theoretician behind Agda (which was implemented by his wife, of whom I have no picture).

(c) Concept of a Type

**Typed vs. untyped languages**

- **Examples of typed languages**: Pascal, C,C++,Java, C#, Haskell, ML, . . .
- **Examples of untyped languages**: Perl, Python, Visual Basic, Lisp, . . .
Advantages/Disadvantages

- **Advantage of untyped languages:**
  Greater freedom in programming.

- **Advantages of typed languages:**
  - Many errors are avoided, especially when using operations defined somewhere else. To find such errors in untyped languages can be very difficult.
  - Types are very natural comments to programs, which express the basic functionality of a program. Typed languages enforce this kind of comments, and therefore better documentation.

Concept of a Type (Cont.)

- In order to guarantee **correctness of software**, we make use of a much more refined type system.
  - It will allow to specify any property of a program, which can be defined as a formula, as a type.

Need for Rich Type Structures

- **In general programming, one wants a very rich type structure.**
  - The richer the type structure, the more data types one can define, the more flexibility one has when writing programs, without loosing the advantages of types (preventing errors).
  - In this lecture we will mainly focus on theorem proving.
  - Greater flexibility touched at the end.

Types used in other Languages

- **Scalar types:**
  Booleans, integers, floating point numbers, characters, enumeration types.

- **Simple compound types:**
  Arrays, strings, record types, lists, sets.

- In **functional programming** additionally:
  Function types, algebraic types (= what can be defined using “data”).

- In **object-oriented programming** (not relevant here): interfaces (and classes).
**Types used in Dep. Type Theory**

- **Function types.**
  Let NatList be the type of lists of natural numbers. Then NatList → NatList is the type of functions mapping lists of natural numbers to lists of natural numbers.
  E.g. sorting functions (without any correctness conditions) are elements of this type.

- **Products** (essentially records).
  Int × Char is the type of pairs ⟨r, s⟩, where r is an integer and s is a character.
  E.g. ⟨2, ‘c’⟩ : Int × Char.
  In Haskell notation, products are written as follows:
  Haskell notation for A × B is (A, B),
  Haskell notation for ⟨a, b⟩ is (a, b).
  e.g. (2, 3) :: (Int, Int).
  Agda notation will be that of a record type in other languages.

**Types used in Dep. Type Theory**

- **Algebraic types.** More about this later.
- **Dependent versions** of the above.

**Per Martin-Löf**

Professor at Stockholm University.
Philosopher, mathematician and computer scientist.
The Father of Martin-Löf Type Theory, the variant of dependent type theory used in this module.
(d) The $\lambda$-Calculus

Basic idea of the $\lambda$-calculus:
We want to define functions “on the fly” (so called “anonymous functions”).

Example:
We want to apply a function to all elements of a list.
For instance, we want to upgrade a list of student numbers to one with one extra digit.

---

Example for need of $\lambda$

Can be done by multiplying each student number by 10.
Let $f : \mathbb{N} \rightarrow \mathbb{N}, f(x) := x \times 10$.
In many languages (e.g. C++, Perl, Python, Haskell) there is a pre-defined operation `map`, which takes a function $f$, and a list $l$, and applies $f$ to each element of the list.
So for the above $f$ we have

\[
\text{map}(f, [210345, 345698, 296458]) = [2103450, 3456980, 2964580].
\]

---

Introduction to $\lambda$-Terms

Often the $f$ is only needed once, and introducing first a new name $f$ for it is tedious.
So one needs a short notation for “the function $f$, s.t. $f(x) = x \times 10$”.
Notation is $\lambda x. x \times 10$.
So we have

\[
\text{map}(\lambda x. x \times 10, [210345, 345698, 296458]) = [2103450, 3456980, 2964580].
\]

In general $\lambda x. t$ stands for the function $f$ s.t. $f(x) = t$, where $t$ might depend on $x$.
above $t = x \times 10$.

---

Notation

One writes in functional programming usually $s t$ for the application of $s$ to $t$ instead of $s(t)$ as usual.
This is used since we have often to apply a function several times, writing something like $f(a)(b)(c)$.
Instead we write $f a b c$.

As indicated by the example, $r s t$ stands for $(r s) t$, in general $r_0 r_1 r_2 \cdots r_n$ stands for $(\cdots ((r_0 r_1) r_2) \cdots r_n)$. 
Infix Operators

- We use + and * infix.
  The corresponding operators are written as (+), (*).
  - So \( x + y \) stands for \((+) \ x \ y\).
  - \( x \cdot y \) stands for \((\cdot) \ x \ y\).
- + and * will bind less than any non-infix constants.
  Therefore \( S \ x + S \ y \) stands for \((S \ x) + (S \ y)\).
- * binds more than +.
  Therefore \( x + y \cdot z \) stands for \( x + (y \cdot z)\),
  and \( S \ x + S \ y \cdot z \) stands for \((S \ x) + ((S \ y) \cdot z)\).
- These are the same conventions as in Agda.

Scope of \( \lambda x \).

- In \((\lambda x \cdot x) \ 5\), the scope \( \lambda x \cdot \) cannot be extended beyond the closing bracket.
  - So it is “\( x \)”,
  - not “\( x \cdot 5 \)”, which doesn’t make sense.
- In \( f(\lambda x \cdot x + 5, 3)\), the scope of \( \lambda x \)
  is “\( x + 5 \)”,
  - not “\( x + 5, 3 \)”, which doesn’t make sense.
- In \((\lambda x \cdot x + 5) \ 3\), the scope of \( \lambda x \)
  is \( x + 5 \)
  - not \( x + 5) \ 3\), which doesn’t make sense.

Scope of \( \lambda x \).

- How do we read \( \lambda x \cdot x + 5? \)
  - As \((\lambda x \cdot x) \ + 5? \)
  - Or as \( \lambda x \cdot (x + 5)? \)

  **Convention:** The scope of \( \lambda x \cdot \) is as long as possible.
  - So \( \lambda x \cdot x + 5 \) reads as \( \lambda x \cdot (x + 5)\).
  - \( \lambda x \cdot (\lambda y \cdot y) \ 5 \) reads as \( \lambda x \cdot ((\lambda y \cdot y) \ 5)\).

\( \lambda \) without a Dot

- Sometimes, \( \lambda x \ t \) (without a dot) is used, if one wants to have the scope of \( \lambda x \) as short as possible.
  - E.g. \( \lambda x \ x \ y \) would denote \( (\lambda x \cdot x) \ y\).
- In this lecture we don’t use this notation.
Bound and Free Variables

- We have now bound and free variables:
  - **Free variables** are variables $x$, which don’t occur in the scope of a $\lambda$-abstraction “$\lambda x.$”.
  - **Bound variables** are the other variables: they are variables $x$ that occur in the scope of a $\lambda$-abstraction “$\lambda x.$”.

In $\lambda x. x + y$,
  - $x$ is bound (since in the scope of $\lambda x$),
  - $y$ is free (since it is not in the scope of $\lambda y$).

Bound and Free Variables

In $(\lambda y. y + z) y$,
  - the first occurrence of $y$, $y$ is bound,
  - the second occurrence of $y$, $y$ is free,
  - $z$ is free.

In $(\lambda y. (\lambda z. z) y) x$, we have
  - $z$ is bound,
  - $y$ is bound (in the scope of $\lambda y$),
  - $x$ is free.

Evaluation of $\lambda$-Terms

- How do we evaluate $(\lambda x. x \ast 10) 5$?
  - We first replace in $x \ast 10$, the variable $x$ by $5$.
  - We obtain $5 \ast 10$.
  - Then we reduce this further, using other reduction rules (not introduced yet).
    Using suitable rules, we would reduce $5 \ast 10$ to $50$.
  - We will first look only at the pure $\lambda$-calculus without any additional reduction rules.
    There $(\lambda x. x \ast 10) 5$ reduces to $5 \ast 10$, which cannot be reduced any further.
Basics of the \(\lambda\)-Calculus

In general, the result of applying \(\lambda x.t\) to \(r\), is obtained by substituting in \(t\) the variable \(x\) by \(r\).

E.g.

- \((\lambda x.x + 10)\ 5\) evaluates to \(5 + 10\),
  - If we substitute in \(x + 10\) the variable \(x\) by \(5\), we obtain \(5 + 10\).
- \((\lambda x.x) \ "Student\”\) evaluates to \"Student\”.
  - If we substitute in \(x\) the variable \(x\) by \"Student\”, we obtain \"Student\”.
- \((\lambda x.x) \ (\lambda y.y)\) evaluates to \(\lambda y.y\).
  - If we substitute in \(x\) the variable \(x\) by \(\lambda y.y\), we obtain \(\lambda y.y\).

Substitution

- \(t[x := s]\) denotes the result of substituting in \(t\) the variable \(x\) by \(s\), e.g.
  - \((x + 10)[x := 5]\) = \(5 + 10\),
  - \(x[x := \ "Student\”]\) = \"Student\”,
  - \(x[x := \lambda y.y]\) = \(\lambda y.y\).

Reductions

- We write \(\rightsquigarrow\) (read as \"reduces to\”) for a one step-reduction as above. So we have
  - \((\lambda x.x + 10)\ 5 \rightarrow \ 5 + 10\),
  - \((\lambda x.x) \ "Student\” \rightarrow \ "Student\”\),
  - \((\lambda x.x) \ (\lambda y.y) \rightarrow \lambda y.y\).
- In general
  - \((\lambda x.t)\ s \rightarrow t[x := s]\).

\(\beta\)-Reduction

- The reduction
  \[(\lambda x.t)\ s \rightarrow t[x := s]\]
  is called \(\beta\)-reduction.
\section*{\textbf{\textbeta-Reduction}}

- We might apply \(\beta\)-reduction to a subterm.

\subsection*{Examples:}
- \((\lambda x. x + 5) \, 3 \mapsto (3 + 5) + 7,\)
since
\((\lambda x. x + 5) \, 3 \mapsto 3 + 5.\)
- Assume we have a pairing operation \((s, t)\) for the pair \(s, t\), then
\(\langle (\lambda x. x + 5) \, 3, 7 \rangle \mapsto (3 + 5, 7).\)
- We can apply \(\beta\)-reduction under a \(\lambda\) term as well:
\(\lambda x. ((\lambda y. y + 5) \, 3) \mapsto \lambda x. 3 + 5.\)

\section*{\textbf{\textbeta-Redex}}

A subterm of the form \((\lambda x. r)\) \(s\) of a \(\lambda\)-term \(t\) is called a \textbf{\(\beta\)-redex}.
- "Redex" is short for \textbf{reducible expression}.
- So \(\beta\)-reduction means reducing one \(\beta\)-redex.
- A \(\lambda\)-term might have several \(\beta\)-redexe:
  - E.g. In \((\lambda x. x) \, ((\lambda y. y) \, z)\) we have
    - one redex \((\lambda x. x) \, ((\lambda y. y) \, z)\)
    - and one redex \((\lambda y. y) \, z\).

\section*{\textbf{\textalpha-Conversion}}

- We identify \(\lambda\)-terms, which only differ in the choice of the bound variables (variables abstracted by \(\lambda\)):
  - So \(\lambda x. x + 5\) and \(\lambda y. y + 5\) are identified.
  - Makes sense, since they both denote the same function \(f\) s.t. \(f(x) = x + 5\).
  - \((\lambda x. x + 5) \, 3 + 7\) and \(\lambda y. y + 5 \, 3 + 7\) are identified.
  - \(\lambda x. \lambda y. y\) and \(\lambda y. \lambda x. x\) are identified.
- This equality is called \textbf{\(\alpha\)-equality}, and the step from one term to another \(\alpha\)-equal term is called \textbf{\(\alpha\)-conversion}.
- So \(\lambda x. \lambda y. y\) and \(\lambda y. \lambda x. x\) are \(\alpha\)-\textbf{\textequal}, written as \(\lambda x. \lambda y. y \equiv \alpha \lambda y. \lambda x. x\).
\(\alpha\)-Conversion

- Note that \(\lambda x.\lambda x.x =_\alpha \lambda y.\lambda x.x\).
  - The \(x\) refers to the second lambda abstraction \(\lambda x\), not the first one (\(\lambda x\)).
  - Therefore, when changing the variable of the first \(\lambda\)-abstraction, \(x\) remains unchanged.

Substitution and Bound Variables

- If we carry out a substitution in a \(\lambda\)-term, we have to be careful.
  - \((\lambda x.x + 7) [x := 3] = \lambda x.x + 7\).
  - It doesn’t make sense to substitute the \(x\) in \(\lambda x.x + 7\), since \(x\) is bound by \(\lambda x\).
  - \(x\) is a bound variable, which is not changed by the substitution.
  - In general, in \(s[x := t]\) we only substitute free occurrences of \(x\) in \(s\) by \(t\).
  - All bound occurrences remain unchanged.

Substitution and Bound Variables

- When substituting in \(\lambda\)-terms, we sometimes have to carry out an \(\alpha\)-conversion first:
  - If we substitute in \(\lambda y.y + x\), the variable \(x\) by 3, we obtain correctly \(\lambda y.y + 3\), the function \(f\) s.t. \(f(y) = y + 3\).
  - If we substitute in \(\lambda y.y + x\), the variable \(x\) by \(y\), we should obtain a function \(f\) s.t. \(f(z) = z + y\).
  - If we did this naively, we would obtain \(\lambda y.y + y\).
  - So the free variable \(y\), which we substituted for \(x\), has become, when substituting it in \(\lambda y.y + x\), to a bound variable.
  - This is not the correct way of doing it.
**Substitution and $\alpha$-Conversion**

- The **correct way** is as follows:
  - First we $\alpha$-convert $\lambda y.y + x$, so that the binding variable $y$ is different from the free variable we are substituting $x$ by:
  - Replace for instance $\lambda y.y + x$ by $\lambda z.z + x$.
  - Now we can carry out the substitution:
    
    $$(\lambda y.y + x)[x := y] = (\lambda z.z + x)[x := y] = \lambda z.z + y .$$

- Similarly, we compute $(\lambda y.y + x)[x := y + y]$ as follows:
    
    $$(\lambda y.y + x)[x := y + y] = (\lambda z.z + x)[x := y + y] = \lambda z.z + (y + y) .$$

**Examples**

- $(\lambda x.(\lambda y.z)[z := x]) = (\lambda u.(\lambda y.z)[z := x]) = (\lambda u.(\lambda y.x))$.
- $(\lambda x.(\lambda y.z)[z := y]) = (\lambda x.(\lambda u.z)[z := y]) = (\lambda x.(\lambda u.y))$.
- $(\lambda x.(\lambda y.y)[z := y]) = (\lambda x.(\lambda y.y)[z := y]) = (\lambda x.(\lambda y.y)y)$.

There is no problem in substituting the $z$ by $y$, since it is not in the scope of $\lambda y$.

**In general, the substitution** $t[x := s]$ is carried out as follows:

- $\alpha$-convert $t$ s.t.
  - if $x$ occurs in $t$ free and is in the scope of some $\lambda u$,
  - then $u$ doesn’t occur free in $s$.
- In other words, $\alpha$-convert $t$ s.t. one never would substitute for $x$ the $s$ in such a way that one of the free variables of $s$ becomes bound.
- Then carry out the substitution.

**Examples**

- $(\lambda x.z)[z := \lambda x.x] = \lambda x.\lambda x.x$.
  - There is no problem with this substitution, since $x$ does not occur free in $\lambda x.x$.
  - Note that the $x$ in $\lambda x.\lambda x.x$ refers to the second $\lambda$-binding $\lambda x$.
- $(\lambda x.z)[z := (\lambda x.x)x] = (\lambda u.z)[z := (\lambda x.x)x] = \lambda u.(\lambda x.x)x$.
  - Now $x$ occurs free in $(\lambda x.x)x$ (the second occurrence is free), so we need to $\alpha$-convert it.
Substitution and $\alpha$-Conversion

- If you have problems understanding this, you can proceed as follows, and are on the safe side:
  - $\alpha$-convert $t$ so that all bound variable in $t$ are different from all free variables in $s$.
  - Then carry out the substitution.
- An unnecessary $\alpha$-conversion doesn’t hurt.

$\lambda$-terms

- Now we can define the untyped $\lambda$-calculus as follows:
  - $\lambda$ terms are:
    - Variables $x$,
    - If $r$ and $s$ are $\lambda$-terms, so is $(r \ s)$.
    - If $x$ is a variable and $r$ is a $\lambda$-term, so is $\lambda x. r$.
  - As usual brackets can be omitted, using the above mentioned conventions about the scope of $\lambda x$,
  - and that $r \ s \ t$ is read as $(r \ s) \ t$.

$\lambda$-terms

- Examples:
  - $\lambda x. x$,
  - $\lambda x. ((\lambda y. y) \ x)$,
  - $\lambda x. x$,
  - $(\lambda x. x \ x) \ (\lambda x. x \ x)$,
  - $(\lambda f. \lambda x. f \ (f \ x))$.

$\lambda$-terms

- One might need additional constants to the language, then we have additionally:
  - Any constant is a $\lambda$-term.
  - So for instance, if $c$ is a constant, then $\lambda x. c$, $(\lambda x. x) \ c$ are $\lambda$-terms,
    - if $+$ is a constant, then $\lambda x. + \ x \ x$ is a $\lambda$-term.
  - For standard operators like $+$, $*$ we usually write
    - $x + y$ instead of $+ x \ y$,
    - $x * y$ instead of $* x \ y$,
    - etc.
\(\alpha\)-Conv, \(\beta\)-Reduction

We identify as before \(\alpha\)-equivalent \(\lambda\)-terms, and define \(\beta\)-reduction as before.

Abbreviations

We write \(\lambda x, y, \ldots\) for \(\lambda x.\lambda y. \ldots\).
Similarly for \(\lambda x, y, z\) etc.
E.g. \(\lambda x, y, z. x\ (y\ z)\) stands for \(\lambda x.\lambda y.\lambda z. x\ (y\ z)\).

Examples of \(\beta\)-Reduction

\[
\begin{align*}
(\lambda x.\lambda y. x)\ y & \rightarrow (\lambda y. x)[x := y] = \lambda u. y \quad = \alpha (\lambda u. x)[x := y] = \lambda u. y \\
(\lambda z.\lambda x.\lambda y. z)\ x & \rightarrow (\lambda x.\lambda y. z)[z := x] = \alpha (\lambda u.\lambda y. z)[z := x] \\
& = \lambda u.\lambda y. x \\
(\lambda x. (\lambda y. y)\ z)\ y & \rightarrow (\lambda x. (\lambda y. y)\ z)[z := y] = \lambda x. (\lambda y. y)\ y \\
\lambda x. (\lambda y. y)\ y & \rightarrow \lambda x. y
\end{align*}
\]

Examples (Longer Reduction)

\[
\begin{align*}
(\lambda x, y. x\ (x\ y))\ (\lambda u, v. u\ (u\ v)) & \rightarrow \lambda y. (\lambda u, v. u\ (u\ v))\ ((\lambda u, v. u\ (u\ v))\ y) \\
& \rightarrow \lambda y. (\lambda u, v. u\ (u\ v))\ (\lambda v. y\ (y\ v)) \\
& \rightarrow \lambda y.\lambda v. (\lambda v. y\ (y\ v))\ ((\lambda v. y\ (y\ v))\ v) \\
& \rightarrow \lambda y.\lambda v. (\lambda v. y\ (y\ v))\ (y\ v)) \\
& \rightarrow \lambda y.\lambda v. y\ (y\ (y\ v))) \\
& = \lambda y, v. y\ (y\ (y\ v)))
\end{align*}
\]
Examples of Non-Termination

- **Reproduction** (Term reduces to itself).
  Let \( \omega := \lambda x. x\ x, \Omega := \omega \omega \). Then
  \[
  \Omega = \omega \omega = (\lambda x. x\ x) \omega \longrightarrow \omega \omega = \Omega.
  \]

- **Expansion** (Term reduct becomes bigger).
  Let \( \Omega := \lambda x. x\ (x\ x) \).
  Then
  \[
  \Omega \overset{\beta}{\longrightarrow} (\lambda x. x\ (x\ x)) \Omega \\
  \overset{\beta}{\longrightarrow} (\Omega (\Omega \Omega)) \\
  \overset{\beta}{\longrightarrow} \ldots
  \]

Reduction Systems

A **reduction system** is a pair \((T, \rightarrow)\) consisting of a set \(T\) (of terms) and a binary relation \(\rightarrow\) on \(T\).

We write \(s \rightarrow t\) for “\(s, t\) are in relation \(\rightarrow\)” and say usually “\(s\) reduces to \(t\)”.

**Example:** The set of \(\lambda\)-terms with \(\beta\)-reduction forms a reduction system:

- \(T\) is the set of \(\lambda\)-terms,
- More precisely, we take \(\lambda\)-terms identified module \(\alpha\)-conversion.
- \(\rightarrow\) is \(\beta\)-reduction.

**Example 1 (Reduction System)**

A simple reduction system is \(\mathbb{N}\) with reductions \(n + 1 \rightarrow n\) for \(n \in \mathbb{N}\):

\[
0 \leftarrow 1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 5
\]

So we have reductions of the form:

\[
5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0.
\]
Example 2 (Reduction System)

Another one is \( \mathbb{N} \) with reductions 
\( n \rightarrow m \) for \( n, m \in \mathbb{N} \) s.t. \( n > m \):

\[
\begin{array}{c}
0 \rightarrow 1 \\
1 \rightarrow 2 \\
2 \rightarrow 3 \\
3 \rightarrow 4
\end{array}
\]

So we have reductions of the form:

\[23 \rightarrow 11 \rightarrow 3 \rightarrow 1 \rightarrow 0.\]

Example 3 (Reduction System)

A third one is \( T = \mathbb{N} \cup \{\ast\} \), with reductions 
\( n + 1 \rightarrow n \) for \( n \in \mathbb{N} \),
and \( \ast \rightarrow n \) for \( n \in \mathbb{N} \):

\[
\begin{array}{c}
0 \rightarrow 1 \\
1 \rightarrow 2 \\
2 \rightarrow 3 \\
3 \rightarrow 4
\end{array}
\]

So we have reductions of the form:

\[\ast \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0.\]

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 System

Take \( T = \) set of arithmetic expressions formed from variables, 0 by using the successor operation \( S \) (where \( S \) \( n \) stands \( n + 1 \)), +, \( \ast \) and brackets.

So the following are elements of \( T \):

\[x + S 0,\]
\[(S 0 + z) \ast (S (S x) + 0),\]
\[(S y) \ast (S 0 + (S x \ast 0)).\]

Take as rules the following:

\[
\begin{align*}
x + 0 & \rightarrow \text{Rule} \quad x, \\
x + S y & \rightarrow \text{Rule} \quad S (x + y), \\
x \ast 0 & \rightarrow \text{Rule} \quad 0, \\
x \ast S y & \rightarrow \text{Rule} \quad (x \ast y) + x.
\end{align*}
\]
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 $s + S t$ by $S(s + t)$,
- any subterm of the form $s * 0$ by $0$,
- any subterm of the form $s * (S t)$ by $(s * t) + s$.

So we have for instance the following reductions:
- $0 + S(S 0) \longrightarrow S(0 + S 0)$,
- Reduce $0 + S s$ to $S(0 + s)$ using $s = S 0$.
- $S(\theta + S \theta) \longrightarrow S(S(\theta + \theta))$,
- Reduce $s + S t$ to $S(s + t)$, using $s = t = 0$,
- $S(S(\theta + \theta)) \longrightarrow S(S(\theta))$.
Reduction generated by \( \longrightarrow \text{Rule} \)

If we have a term rewriting system \((T, \longrightarrow_{\text{Rule}})\) we obtain a reduction relation \(\longrightarrow\) on \(T\) as follows:

First we construct a relation \(\longrightarrow'\) obtained from reductions rules \(r \longrightarrow_{\text{Rule}} r'\) by substituting the free 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 \(r \longrightarrow_{\text{Rule}} r'\), then \(s \longrightarrow' s'\) is called an instance of rule \( r \longrightarrow_{\text{Rule}} r'\).

---

Example 1

\[
\begin{align*}
x + 0 & \longrightarrow_{\text{Rule}} x , \\
x + S y & \longrightarrow_{\text{Rule}} S (x + y) , \\
x \ast 0 & \longrightarrow_{\text{Rule}} 0 , \\
x \ast S y & \longrightarrow_{\text{Rule}} (x \ast y) + x .
\end{align*}
\]

\(0 + S (S 0) \longrightarrow S (0 + S 0)\) is obtained as follows:

- The rule used is \(x + S y \longrightarrow_{\text{Rule}} S (x + y)\).
- By substituting \(x\) by \(0\) and \(y\) by \(S 0\) we obtain the instance \(0 + S (S 0) \longrightarrow' S (0 + S 0)\).
- In this example, the full term is reduced.

---

Example 2

\[
\begin{align*}
x + 0 & \longrightarrow_{\text{Rule}} x , \\
x + S y & \longrightarrow_{\text{Rule}} S (x + y) , \\
x \ast 0 & \longrightarrow_{\text{Rule}} 0 , \\
x \ast S y & \longrightarrow_{\text{Rule}} (x \ast y) + x .
\end{align*}
\]

\(S (0 + S 0) \longrightarrow S (S (0 + 0))\) is obtained as follows:

- The rule used is \(x + S y \longrightarrow_{\text{Rule}} S (x + y)\).
- By substituting \(x\) and \(y\) by \(0\) we obtain the instance \(0 + S 0 \longrightarrow' S (0 + 0)\).
- The left hand side of our reduction \(S (0 + S 0)\) contains now the subterm \(0 + S 0\).
- By substituting it by \(S (0 + 0)\) we obtain the right hand side of the reduction, \(S (S (0 + 0))\).
Example 3

\[
\begin{align*}
  x + 0 & \quad \rightarrow \quad x, \\
  x + S \ y & \quad \rightarrow \quad S \ (x + y), \\
  x \ast 0 & \quad \rightarrow \quad 0, \\
  x \ast S \ y & \quad \rightarrow \quad (x \ast y) + x. \\
\end{align*}
\]

The rule used is \( \rightarrow \) Rule \( x \).

By substituting \( 0 \) for \( x \), we obtain the instance

\[ 0 + 0 \rightarrow' 0. \]

The left hand side of the reduction \( S \ (S \ (0 + 0)) \) contains subterm \( 0 + 0 \).

By substituting it by \( 0 \) we obtain the right hand side of the reduction \( S \ (S \ 0) \).

---

In order to express the above shorter, one says that \( \rightarrow^* \) is the transitive and reflexive closure of \( \rightarrow \), i.e. the least transitive and reflexive relation containing \( \rightarrow \):

- \( a \rightarrow b \) implies \( a \rightarrow^* b \).
- \( \rightarrow^* \) is reflexive, i.e. for all \( a \) we have \( a \rightarrow^* a \).
- \( \rightarrow^* \) is transitive, i.e. \( a \rightarrow^* b \rightarrow^* c \) implies \( a \rightarrow^* c \).
- If there is any relation \( \rightarrow' \) with the above 3 properties, then \( x \rightarrow' y \) implies \( x \rightarrow^* y \).

---

Example

If \( (T, \rightarrow) \) is a reduction system, we define

- \( a \rightarrow^* b \) if there exists a (possibly empty) sequence of reductions
  \[ a = a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n = b \]

By empty reduction we mean: if \( a = b \), then we have \( a \rightarrow^* b \).

---

If we take \( \mathbb{N} \) with reductions \( n + 1 \rightarrow n \) for \( n \in \mathbb{N} \):

\[ 0 \leftarrow 1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 5 \]

Then \( 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \), therefore \( 5 \rightarrow^* 2 \).

In general \( n \rightarrow^* m \Leftrightarrow n \geq m \).
If \( (T, \rightarrow) \) is a reduction system, we define
\[
\frac{a \rightarrow^{*} b}{\text{iff there exists a (possibly empty) sequence of reductions}}
\]
\[
a = a_0 \leftrightarrow a_1 \leftrightarrow a_2 \leftrightarrow \cdots \leftrightarrow a_n = b
\]
where
\[
a \leftrightarrow b \iff (a \rightarrow b \lor b \rightarrow a)
\]
Determination of $\leftrightarrow^*$

In general it is infeasible to determine whether $a \leftrightarrow^* b$ holds.

- One has to check all possible ways of getting from $a$ to $b$, by both using $\rightarrow$ and $\leftarrow$.

In many cases this can be determined by:

- Simply reducing $a$ as long as possible to some term $a'$ s.t. $a \rightarrow^* a'$ and s.t. $a'$ has no further reductions, i.e. by “evaluating $a'$”.
- Doing the same with $b$ to some term $b'$.
- Checking whether $a'$ is identical to $b'$.

This is possible, if $\rightarrow$ is confluent and strongly normalising (see next slides).

Agda

- The underlying reduction system of Agda is “essentially” strongly normalising.
- The equality derived from this reduction system is used in order to typecheck terms.

Interactive Theorem Proving, CS\_336, Lentterm 2004, Sec. 1

Strong Normalisation

- A reduction system $(A, \rightarrow)$ is **terminating** or **strongly normalising**, iff there is no infinite sequence $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \cdots$

Examples

- The following reduction system is terminating:

$$
\begin{array}{c}
0 \leftarrow 1 \leftarrow 2 \leftarrow 3 \leftarrow 4 \leftarrow 5 \\
\end{array}
$$

Any reduction sequence will end in 0 and terminate.

- The following reduction system is not terminating:

$$
\begin{array}{c}
0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \\
(Take 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots)
\end{array}
$$

Interactive Theorem Proving, CS\_336, Lentterm 2004, Sec. 1
Examples

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

$$\ast \rightarrow n \rightarrow (n-1) \rightarrow (n-2) \rightarrow \cdots \rightarrow 0.$$

We have $\ast \rightarrow n \rightarrow (n-1) \rightarrow (n-2) \rightarrow \cdots \rightarrow 0$.

- The untyped $\lambda$-calculus is not terminating, since we have

$$\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \cdots$$

Normal Form and Irreducibility

Let $(A, \rightarrow)$ be a reduction system.

- $a \in A$ is **irreducible**, if there exists no $b \in A$ s.t. $a \rightarrow b$.

- $b$ is a **normal form of $a$** if $a \rightarrow^* b$ and $b$ is irreducible.

- $(A, \rightarrow)$ is **weakly normalising** or **normalising**, if every $a \in A$ has a normal form.

Example

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

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow \cdots$$

- Every $a$ has a normal form, namely $\ast$.

- But there exists an infinite sequence

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \cdots$$

Lemma

Let $(A, \rightarrow)$ be a strongly normalising reduction system. Then $(A, \rightarrow)$ is weakly normalising.

**Proof:**

A normal form of $a$ can be obtained by simply reducing $a$ as long as possible:

Since $(A, \rightarrow)$ is strongly normalising, the reduction sequence terminates in some $b \in A$.

$b$ is a normal form of $A$. 
Confluence

A reduction system \((A, \rightarrow)\) is confluent, iff for all \(x, y, z \in A\) we have
- if \(x \rightarrow^* y\) and \(x \rightarrow^* z\),
- then there exists an \(u\) s.t. \(y \rightarrow^* u\) and \(z \rightarrow^* u\).

\[
\begin{array}{c}
\ast \\
\downarrow \\
y \\
\ast \\
\ast \\
\downarrow \\
z \\
\ast \\
x \\
\ast \\
\end{array}
\]

exists \(u\)

Church-Rosser

If \((A, \rightarrow)\) is confluent, then it has the Church-Rosser property:
If \(a \leftrightarrow^* b\) iff there exists a \(c\) s.t.
\[
a \rightarrow^* c \land b \rightarrow^* c
\]

\[
\begin{array}{c}
a \\
\leftarrow \\
\ast \\
\ast \\
\end{array}
\]

\[
\begin{array}{c}
b \\
\rightarrow \\
\ast \\
\ast \\
\end{array}
\]

exists \(c\)

Proof of Church-Rosser

Define \(a \downarrow b :\iff \exists c. (a \rightarrow^* c \land b \rightarrow^* c)\).

We show that \(\downarrow\) contains \(\rightarrow\) and is reflexive, symmetric and transitive. Therefore it contains \(\leftrightarrow^*\), and we get \(a \leftrightarrow^* b \Rightarrow a \downarrow b\).

In the other direction, \(a \downarrow b\) clearly implies \(a \leftrightarrow^* b\).

\[
\begin{array}{c}
a \\
\rightarrow \\
\ast \\
\ast \\
\end{array}
\]

\[
\begin{array}{c}
b \\
\rightarrow \\
\ast \\
\ast \\
\end{array}
\]

Formally: If \(a \rightarrow b\) then we have with \(c := b\) that \(a \rightarrow^* c\) and \(b \rightarrow^* c\).
Proof of Church-Rosser

—is reflexive,

Formally: We have $a \rightarrow^* a$ and $a \rightarrow^* a$, therefore $a \downarrow a$.

—is transitive:

Formally:
- Assume $a \downarrow b$ and $b \downarrow c$ by $a \rightarrow^* d$, $b \rightarrow^* d$, $b \rightarrow^* e$, $c \rightarrow^* e$.
- Then by confluence there exists an $f$ s.t. $d \rightarrow^* f$, $e \rightarrow^* f$.
- But then $a \rightarrow^* f$ and $c \rightarrow^* f$, $a \downarrow c$.

Proof of Church-Rosser

—is symmetric:

Formally: If $a \downarrow b$ by $a \rightarrow^* c$ and $b \rightarrow^* c$, then $b \downarrow a$ by

$b \rightarrow^* c$ and $a \rightarrow^* c$.

Proof of Church-Rosser

Fact: The $\lambda$-calculus is confluent (if we identify $\alpha$-equivalent terms), and fulfills therefore the Church-Rosser property.

Note that this doesn’t give yet an easy way of determining whether $a =_\beta b$ holds:

- One needs to find a $c$ s.t. $a \rightarrow^* c$ and $b \rightarrow^* c$.
- But simply reducing $a$ might never terminate.
Unique Normal Forms

Lemma:
Let \((A, \rightarrow)\) be a confluent reduction system. If \(a \in A\) has a normal form \(a'\), then it is unique:
- If \(a''\) is another normal form, then \(a' = a''\).

Proof:
- We have \(a \rightarrow^* a'\) and \(a \rightarrow^* a''\).
- By confluence, there exists a \(b\) s.t. \(a' \rightarrow^* b\) and \(a'' \rightarrow^* b\).
- But since \(a'\) and \(a''\) are normal forms, it follows \(a' = b\) and \(a'' = b\).

Towards the Typed \(\lambda\)-Calculus

Problem of the untyped \(\lambda\)-calculus:
- Non-Termination, therefore \(=\beta\) difficult to check.
- In fact \(=\beta\) is semi-decidable (r.e.), but not decidable (recursive).
- Caused by the possibility of self-application, which allows to write essentially fully recursive programs.
- Avoided by the simply typed \(\lambda\)-calculus, which is strongly normalising.

Interactive Theorem Proving, CS\_336, Lentterm 2004, Sec. 1
Basics of the Typed $\lambda$-Calculus

- $\lambda x : \text{int}.x + 5$ is only applicable to some $s : \text{int}$, therefore not applicable to elements of other types, e.g. to “Student” (: String).

So
- $(\lambda x : \text{int}.x + 5) 3$ is allowed,
- $(\lambda x : \text{int}.x + 5)$ “Student” is not allowed.

Simple Types

- The **simple types** used in the simply typed $\lambda$-calculus are defined inductively as follows:
  - The **ground type** $o$ is a type.
  - If $\sigma$, $\tau$ are types, so is $(\sigma \to \tau)$.

  “Inductively” means that the set of simple types is the least set containing the ground type, and which closed under $\to$.

  One sometimes modifies the set of ground types, especially when adding constants to the $\lambda$-terms.
  - E.g. when using arithmetic expressions, one can say for instance that the ground types are $\text{int}$ and $\text{float}$.
  - Then we talk about the **simple types based on ground types** $\text{int}$ and $\text{float}$.

Simple Types

- Usually we denote types by Greek letters, e.g. $\alpha$ (“alpha”), $\beta$ (“beta”), $\gamma$ (“gamma”), $\sigma$ (“sigma”), $\tau$ (“tau”).

- We omit brackets as usual using the convention that $\alpha \to \beta \to \gamma$ stands for $\alpha \to (\beta \to \gamma)$.

Examples types:
- $o$,
- $o \to o$,
- $(o \to o) \to o$,
- $((o \to o) \to o) \to o \to o$,
- which stands for $(((o \to o) \to (o \to o)) \to ((o \to o) \to (o \to o)))$.

Abbreviation

- In order to make writing down such types easier, one can use sometimes the following abbreviations (these are non-standard abbreviations, and should be defined explicitly when using outside this lecture).
  - $o2 := o \to o$,
  - $o3 := o2 \to o2$,
  - etc.

So
- an element of type $o2$ can be applied to an element of type $o$ and one obtains an element of type $o$.
- an element of type $o3$ can be applied to an element of type $o2$ and one obtains an element of type $o2$.
- etc.
Contexts

To determine the type of a term makes only sense, if we know the types of its variables.

For instance, in case of the \( \lambda \)-term \( x \ y \), we could have

- \( x : o \rightarrow o, y : o \) and therefore \( x \ y : o \),
- or \( x : o \rightarrow o, y : (o \rightarrow o) \rightarrow o \rightarrow o \), and therefore \( x \ y : o \rightarrow o \).

Therefore we will give a type to \( \lambda \) terms in a context, which determines the types of the variables.

---

Contexts

A **context** is an expression of the form

\[ x_1 : \sigma_1, \ldots, x_n : \sigma_n \]

where

- \( x_i \) are variables,
- \( \sigma_i \) are simple types,
- \( \sigma_i \) will be types of that theory.
- \( x_i \) are different.
- \( n = 0 \) is allowed, and we write \( \emptyset \) for the empty context.

Examples

- \( x : o, y : o \rightarrow o \) is a context.
- \( x : o \rightarrow o, x : o \) is **not** a context.

Note that contexts are **lists** of elements of the form \( x : \sigma \), so the order matters.

In case of the simply typed \( \lambda \)-calculus, it wouldn’t make a difference to have as context unordered sets of expressions of the form \( x : \sigma \).

However, when moving later to dependent type theory, the order of the expressions \( x : \sigma \) will be relevant.

---

Contexts

In the following, the capital Greek letters \( \Gamma \) (“Gamma”), \( \Delta \) (“Delta”) denote contexts.

We write \( \Gamma \vdash s : \sigma \) for “in context \( \Gamma \), \( s \) has type \( \sigma \)”.

Expressions of this form are called **judgements**.

Examples:

- \( x : o \rightarrow o, y : o \Rightarrow x \ y : o \),
- \( x : \text{float} \rightarrow \text{int}, y : \text{float} \Rightarrow x \ y : \text{int} \)
  (assuming ground types float and int),
- \( x : (o \rightarrow o) \rightarrow o \rightarrow o, y : o \rightarrow o \Rightarrow x \ y : o \rightarrow o \).

In case \( \Gamma \) is empty, we write \( s : \sigma \) instead of \( \emptyset \Rightarrow s : \sigma \).
Contexts

If \( \Gamma, \Delta \) are contexts, \( \Gamma, \Delta \) denotes the concatenation of both contexts, e.g. if
\[ \Gamma = x : o, y : o \rightarrow o, \]
\[ \Delta = z : o \]
then
\[ \Gamma, \Delta \] denotes \( x : o, y : o \rightarrow o, z : o, \)
\[ \Delta, \Gamma \] denotes \( z : o, x : o, y : o \rightarrow o, \)
\[ \Gamma, u : o \] denotes \( x : o, y : o \rightarrow o, u : o. \)

Simply Typed \( \lambda \)-Calculus

**Definition** of the simply typed \( \lambda \)-terms, depending on a context, together with their type.

1. **Assumption.**
   Variables, occurring in the context, are terms having the type they have in the context:
   \[ \Gamma, x : \sigma, \Delta \Rightarrow x : \sigma \]
   - Note that \( \Gamma, x : \sigma, \Delta \) stands for any context, in which \( x : \sigma \) occurs.
   - **Explanation:** From the assumption \( x : \sigma \) we can derive \( x : \sigma \).

2. **Application.**
   If \( s \) is of type \( \sigma \rightarrow \tau \) and \( t \) of type \( \sigma \), depending on context \( \Gamma \), then \( s \ t \) is of type \( \tau \) under context \( \Gamma \):
   \[ \Gamma \Rightarrow s : \sigma \rightarrow \tau, \quad \Gamma \Rightarrow t : \sigma \quad \text{(Ap)} \]
   - **Explanation:**
     - Assume we have \( s \) of type \( \sigma \rightarrow \tau \).
     - So \( s \) is a function, taking an \( x : \sigma \) and returning an element of type \( \tau \).
     - Assume we have \( t \) is an element of type \( \sigma \).
     - Then we can apply the function \( s \) to this \( t \), written as \( s \ t \), and obtain an element of type \( \tau \).

3. **Abstraction.**
   If \( t \) is a term of type \( \tau \), depending on context \( \Gamma, x : \sigma \), then \( \lambda x : \sigma.t \) is a term of type \( \sigma \rightarrow \tau \) depending on context \( \Gamma \):
   \[ \Gamma, x : \sigma \Rightarrow t : \tau \]
   \[ \Gamma \Rightarrow (\lambda x : \sigma.t) : \sigma \rightarrow \tau \quad \text{(Abs)} \]
   - **Explanation:**
     - If we have under assumption \( x : \sigma \) shown that \( t : \tau \), then we can form a new \( \lambda \)-term by binding that \( x \), and form \( \lambda x : \sigma.t \).
     - The result is a function taking as input \( x : \sigma \) and returning \( t : \tau \), so we obtain an element of \( \sigma \rightarrow \tau \).
**Rules**

We had three rules:

1. \( \Gamma, x : \sigma, \Delta \Rightarrow x : \sigma \)

2. \( \frac{\Gamma \Rightarrow s : \sigma \rightarrow \tau \quad \Gamma \Rightarrow t : \sigma}{\Gamma \Rightarrow st : \tau} \) (Ap)

3. \( \frac{\Gamma, x : \sigma \Rightarrow t : \tau}{\Gamma \Rightarrow (\lambda x : \sigma.t) : \sigma \rightarrow \tau} \) (Abs)

*It expresses:*
- Whenever we have derived \( \Gamma, x : \sigma \Rightarrow t : \tau \)
- (for arbitrary context \( \Gamma \), types \( \sigma, \tau \), variable \( x \) and term \( t \)),
then we can derive from this \( \Gamma \Rightarrow (\lambda x : \sigma.t) : \sigma \rightarrow \tau \).

**Derivations**

Using rules we can derive more complex judgements:

- We start with axioms, and use rules with premises in order to derive further judgements.

*Example 1:*

\( \frac{x : o \Rightarrow x : o}{(\lambda x : o.x) : o \rightarrow o} \) (Abs)
Example 2

\[
\frac{x : o \rightarrow o, y : o \Rightarrow x : o \rightarrow o}{x \rightarrow o, y : o \Rightarrow y \rightarrow o} \quad \text{(Ap)}
\]

\[
\frac{x : o \rightarrow o, y : o \Rightarrow x \rightarrow y}{x : o \rightarrow o \Rightarrow (\lambda y : o. y) : o \rightarrow o} \quad \text{(Abs)}
\]

Note that we have the following dependencies in the derived λ-term:

\[
(\lambda x : o \rightarrow o. \overbrace{\lambda y : o. x}^{o \rightarrow o} \quad \overbrace{y^{o \rightarrow o}}^{o} : o \rightarrow o
\]

Self-Application

In the simply typed λ-calculus we cannot assign a type to \( \lambda x.x x \), i.e. there are no types \( \sigma, \tau \) s.t. \( (\lambda x : \sigma. x x) : \tau \).

Assume we could derive this.

The only way to derive \( (\lambda x : \sigma. x x) : \tau \) is by the rule of λ-abstraction.

Then \( \tau \) must be equal to \( \sigma \rightarrow \tau_1 \) for some \( \tau_1 \), and the derivation reads then

\[
\frac{x : \sigma \Rightarrow x \rightarrow \tau_1}{(\lambda x : \sigma. x x) : \sigma \rightarrow \tau_1} \quad \text{(Abs)}
\]

The only way to derive \( x : \sigma \Rightarrow x \rightarrow \tau_2 \rightarrow \tau_1 \) and \( x : \sigma \Rightarrow x \rightarrow \tau_2 \) is by using the assumption rule.

In order for \( x : \sigma \Rightarrow x \rightarrow \tau_2 \rightarrow \tau_1 \) to be derivable by the assumption rule, we need \( \sigma = \tau_2 \rightarrow \tau_1 \).

Similarly, in order to derive \( x : \sigma \Rightarrow x : \tau_2 \), we need \( \tau_2 = \sigma \).

So we have \( \tau_2 \rightarrow \tau_1 = \sigma = \tau_2 \).

But \( \tau_2 = \tau_2 \rightarrow \tau_1 \) cannot be fulfilled, since \( \tau_2 \rightarrow \tau_1 \) is longer than \( \tau_2 \).

So we cannot find types \( \sigma, \tau \) s.t. \( (\lambda x : \sigma. x x) : \tau \).
Remark (Weakening)

- If we have derived $t : \sigma$ under some context, then the same holds for any other context, which expands the original one.
- Formally, this means: Assume $\Gamma, \Delta \Rightarrow t : \sigma$.

Then we have as well $\Gamma, x : \tau, \Delta \Rightarrow t : \sigma$, provided $\Gamma, x : \tau, \Delta$ is a context (i.e. provided $x$ does not occur in $\Gamma, \Delta$).
- The process of extending the context is called weakening.

Proof of the Remark

- Assume a derivation of $\Gamma, \Delta \Rightarrow t : \sigma$.
- Insert at all corresponding positions in the contexts in the derivation $x : \tau$.
- One needs to rename variables, in order to avoid conflicts with $x$.
- The result is a derivation of $\Gamma, x : \tau, \Delta \Rightarrow t : \sigma$.

Example (Weakening)

From the derivation

$$y : o, x : o \Rightarrow x : o \quad \text{(Abs)}$$

$$y : o \Rightarrow (\lambda x : o.x) : o \Rightarrow o$$

$$y : o \Rightarrow y : o \quad \text{(Ap)}$$

we obtain a derivation of

$$y : o, x : o \Rightarrow (\lambda x : o.x) y : o$$

as follows:
**Example (Weakening)**

Because of the possibility of weakening, we will usually omit unused parts of contexts.

So a derivation of \( x : o \rightarrow o, y : o \Rightarrow x \ (x \ y) : o \), which in full reads as follows:

\[
\frac{x : o \rightarrow o, y : o \Rightarrow x \ (x \ y) : o}{x : o \rightarrow o, y : o \Rightarrow x \ (x \ y) : o}
\]

will usually be presented as follows:

\[
\frac{x : o \rightarrow o, y : o \Rightarrow x \ (x \ y) : o}{x : o \rightarrow o, y : o \Rightarrow x \ (x \ y) : o}
\]

**Weakening**

- First rename in this derivation \( x \) by \( u \) in order to avoid conflicts (note that \( \lambda x : o. x \) is \( \alpha \)-equivalent to \( \lambda u : o. u \)):

\[
\frac{y : o, u : o \Rightarrow u : o}{y : o \Rightarrow (\lambda u : o. u) : o \rightarrow o \quad y : o \Rightarrow y : o}
\]

Also note that \( \lambda x : o.x \) and \( \lambda u : o.u \) are identified.

**Example (Weakening)**

- \( y : o, x : o \Rightarrow x : o \quad (\text{Abs}) \)

\[
\frac{y : o \Rightarrow (\lambda x : o. x) : o \rightarrow o}{y : o \Rightarrow y : o}
\]

\( y : o \Rightarrow (\lambda x : o. x) \ y : o \)

Now we obtain the following derivation of

\[
\frac{y : o, x : o \Rightarrow (\lambda x : o. x) \ y : o}{y : o, x : o, u : o \Rightarrow u : o}
\]

\[
\frac{y : o, x : o \Rightarrow (\lambda u : o. u) : o \rightarrow o}{y : o, x : o \Rightarrow y : o}
\]

\( y : o, x : o \Rightarrow (\lambda u : o. u) \ y : o \)

Note that \( \lambda x : o.x \) and \( \lambda u : o.u \) are identified.

**\( \beta \)-Reduction**

- \( \beta \)-reduction for typed \( \lambda \)-terms is defined as for untyped \( \lambda \)-terms.

- One has only to carry around the types as well.

- Formally we have

\[
(\lambda x : \sigma.t) \ s \longrightarrow t[x := s]
\]

- And as before \( \beta \)-reduction can be applied to any subterm.

- A subterm \( (\lambda x : \sigma.t) \ s \) of a term \( s \) is called a \( \beta \)-redex of \( s \).
\[(\lambda x : (o \to o) \to o . \lambda y : o \to o . x \ (x \ y)) \ (\lambda x : o \to o . \lambda y : o . x \ (x \ y))\]

\[\to (\lambda y : o \to o . (\lambda x : o \to o . \lambda y : o . x \ (x \ y)) \ (\lambda z : o . y \ (y \ z)))\]

\[\to_\alpha (\lambda y : o \to o . (\lambda x : o \to o . \lambda u : o . x \ (x \ u)) \ (\lambda z : o . y \ (y \ z)))\]

\[\to (\lambda y : o \to o . \lambda u : o . (\lambda z : o . y \ (y \ z)) \ (\lambda z : o . y \ (y \ z)) \ u)\]

\[\to (\lambda y : o \to o . \lambda u : o . (\lambda z : o . y \ (y \ z)) \ (y \ (y \ u)))\]

\[\to (\lambda y : o \to o . \lambda u : o . y \ (y \ (y \ u)))\]

\[\eta\text{-Rule}\]

- If we have a function \(f : \sigma \to \tau\), then this function applied to \(x : \sigma\) gives result \(f \ x\).

Therefore \(f\) is as a function the same as \(\lambda x . f \ x\) (where \(x\) is fresh).

- However, if for instance \(f\) is a variable, we don't have \(f =_\beta \lambda x . f \ x\).

Especially, when working later in dependent type theory we want to identify as many terms as possible, which are equal.

This will make it easier to prove certain goals.

Therefore we introduce a rule, which expresses that \(f\) is always equal to \(\lambda x . f \ x\) w.r.t. \(\beta, \eta\)-reduction (where \(x\) is fresh).

\[\eta\text{-Rule}\]

- The \(\eta\)-rule expresses that subterms \(t : \sigma \to \tau\) can be \(\eta\)-expanded to \(\lambda x . t \ x\) (where \(x\) does not occur free in \(t\)).

However, we need to impose some restrictions, in order to avoid circularities:
  - If \(t\) is of the form \(\lambda y . s\), then
    
    \[\lambda x . t \ x = \lambda x . (\lambda y . s) \ x \to_\beta \lambda x . s[y := x] =_\alpha t\ ,\]
    
    so if we allowed to expand \(t\), we would obtain a circularity.
  - If \(t\) occurs in the form \(t \ r\), then we get
    
    \[(\lambda x . t \ x) \ r \to_\beta t \ r\ .\]
    
    Again expanding \(t\) would result in a circularity.
  - All other terms can be expanded without obtaining a new redex.
\(\eta\)-Expansion

\(\eta\)-expansion (or \(\eta\)-rule) is the rule which expands one subterm of a \(\lambda\)-term

- of the form \(r : \sigma \rightarrow \tau\)
- s.t. \(r\) is not of the form \(\lambda y : \sigma . t\)
- and such that \(r\) is not applied to some other term to \(\lambda x : \sigma . x\), where \(x\) does not occur free in \(r\).

We write \(r \eta\) for \(s\) is obtained from \(r\) by the \(\eta\)-rule,

\(r \rightarrow^* \beta, \eta\) for \(s\) is obtained from \(r\) by using \(\beta\)-reduction or \(\eta\)-expansion.

Notions like \(\beta, \eta\)-normal form, etc. are to be understood correspondingly.

Example

\[
(\lambda f : (o \rightarrow o) \rightarrow o \rightarrow o . \lambda x : o \rightarrow o . f \ x) \ f
\]

\[
\rightarrow_{\beta} \lambda x : o \rightarrow o . f \ x
\]

\[
\rightarrow_{\eta} \lambda x : o \rightarrow o . \lambda y : o . f \ x \ y
\]

\[
\rightarrow_{\eta} \lambda x : o \rightarrow o . \lambda y : o . f (\lambda z : o . x \ z) \ y
\]

\(\lambda x : o \rightarrow o . \lambda y : o . f (\lambda z : o . x \ z) \ y\) is therefore the \(\beta, \eta\)-normal form of

\((\lambda f : (o \rightarrow o) \rightarrow o \rightarrow o . \lambda x : o \rightarrow o . f \ x) \ f\).

Since \(f \rightarrow_{\eta} \lambda x : o \rightarrow o . f \ x\), this is as well the \(\beta, \eta\)-normal form of \(f : (o \rightarrow o) \rightarrow o \rightarrow o\).

Theorem

- The typed \(\lambda\)-calculus with \(\beta\)-reduction and \(\eta\)-expansion is confluent and strongly normalising.
### η-Rule

With the η-rule we obtain now that if \( r : \sigma \rightarrow \tau \), then
\[
\begin{align*}
\Gamma \Rightarrow s : \sigma & \quad \Gamma \Rightarrow t : \tau \\
\Gamma \Rightarrow \langle s, t \rangle : \sigma \times \tau
\end{align*}
\]

- If \( r : \sigma \rightarrow \tau \) is of the form \( \lambda y : \sigma.t \) then we have

\[
\begin{align*}
\lambda x : \sigma.x \ x & = \lambda x : \sigma.(\lambda y : \sigma.t \ x) \ x \\
\rightarrow_{\beta} & \quad \lambda x : \sigma.t[y := x] \\
=_{\alpha} & \quad \lambda y : \sigma.t \\
= & \quad r
\end{align*}
\]

- Otherwise \( r \rightarrow_{\eta} \lambda x.\sigma.x \ x \).

Therefore, every function is of the form \( \lambda x.\text{something} \).

### η-Reduction

In the literature one often uses instead of η-expansion

\[\eta\text{-reduction}\], which allows to reduce \( \lambda x : \sigma.x \ x \) to \( r \), if \( x \) doesn’t occur free in \( r \).

The computation of η-reduction is more difficult than η-expansion, since one has to check, whether \( x \) doesn’t occur free in \( r \).

Therefore in the context of interactive theorem proving, we prefer η-expansion.

### Products

One can expand the set of \( \lambda \)-types and \( \lambda \)-terms as follows:

- Types are defined as before, but we have additionally:
  - If \( \sigma, \tau \) are types, so is \( \sigma \times \tau \).

### Products

- The set of typed-\( \lambda \)-terms are defined as before but we have:
  - If \( s : \sigma, t : \tau \) then \( \langle s, t \rangle : \sigma \times \tau \):

\[
\begin{align*}
\frac{\Gamma \Rightarrow s : \sigma \quad \Gamma \Rightarrow t : \tau}{\Gamma \Rightarrow \langle s, t \rangle : \sigma \times \tau}
\end{align*}
\]

- If \( s : \sigma \times \tau \), then \( \pi_0(s) : \sigma \) and \( \pi_1(s) : \tau \):

\[
\begin{align*}
\frac{\Gamma \Rightarrow s : \sigma \times \tau}{\Gamma \Rightarrow \pi_0(s) : \sigma} \\
\frac{\Gamma \Rightarrow s : \sigma \times \tau}{\Gamma \Rightarrow \pi_1(s) : \tau}
\end{align*}
\]
**β-Reduction for Pairs**

- β-reduction for the pairs is the rule which allows to replace
  - any subterm of the form \( \pi_0((r_0, r_1)) \) by \( r_0 \),
  - any subterm of the form \( \pi_1((r_0, r_1)) \) by \( r_1 \).

The subterms \( \pi_i((r_0, r_1)) \) are as before called \( \beta \)-redex of the term in question.

- β-reduction for the typed λ-calculus with products includes both β-reduction for functions and β-reduction for pairs.

---

**η-Rule for Products**

- The η-rule expresses that subterms \( t : \sigma \times \tau \) can be η-expanded to \( \langle \pi_0(t), \pi_1(t) \rangle \).

- However, as for functions, we need to impose some restrictions, in order to avoid circularities:
  - If \( t \) is of the form \( \langle r_0, r_1 \rangle \),

\[
\langle \pi_0(t), \pi_1(t) \rangle \beta \longrightarrow \langle r_0, r_1 \rangle = t
\]

so if we allowed to expand \( t \), we would obtain a circularity.

---

**η-Expansion for Products**

- If we have a product \( r : \sigma \times \tau \), then its projections are β-equal to the projections of \( \langle \pi_0(r), \pi_1(r) \rangle \):
  - \( \pi_0(\langle \pi_0(r), \pi_1(r) \rangle) =_\beta \pi_0(r) \),
  - \( \pi_1(\langle \pi_0(r), \pi_1(r) \rangle) =_\beta \pi_1(r) \).

- Therefore, similarly to functions, we would like to have that every term \( r : \sigma \times \tau \) is equal to \( \langle \pi_0(r), \pi_1(r) \rangle \).

---

**η-Rule for Products**

- If \( t \) occurs in the form \( \pi_i(t) \), then we get

\[
\pi_i(\langle \pi_0(t), \pi_1(t) \rangle) \beta \longrightarrow \pi_i(t)
\]

expanding \( t \) would result in a circularity.

- All other terms can be expanded without obtaining a new redex.
\[ \eta \text{-Expansion for Products} \]

- \( \eta \)-expansion for products is the rule which allows to replace in a typed \( \lambda \)-term \( t \)
  - one subterm \( s : \sigma \times \tau \)
  - which is not of the form \( <r_0, r_1> \)
  - and does not occur in the form \( \pi_0(s) \) or \( \pi_1(s) \) by \( <\pi_0(s), \pi_1(s)> \).

- \( \eta \)-expansion for the typed \( \lambda \)-calculus with products includes both \( \eta \)-expansion for functions and for pairs.

\[ \eta \text{-Rule} \]

With the \( \eta \)-rule we obtain now that if \( r : \sigma \times \tau \), then
\[ r \Rightarrow_{\beta, \eta} <\pi_0(r), \pi_1(r)>. \]

- If \( r : \sigma \times \tau \) is of the form \( <r_0, r_1> \) then we have
  \[ r \Rightarrow_{\beta} <\pi_0(r), \pi_1(r)> : \]
  \[ \pi_0(<r_0, r_1>), \pi_1(<r_0, r_1>) \]
  \[ <r_0, r_1> \]

- Otherwise \( r \Rightarrow_{\eta} <\pi_0(r), \pi_1(r)>. \)

Therefore, every element of a product type is of the form \( <\text{something}_0, \text{something}_1> \).
One can combine the $\lambda$-calculus with term writing.

Consider the $\lambda$-calculus with terms using additional constants.

Assume some term rewriting rules as before (which might involve some $\lambda$-terms).

As in case of ordinary term rewriting, we form instantiations of the rules by replacing variables by arbitrary $\lambda$-terms (in the extended language).

Assume for instance the rule

$$\text{double } \longrightarrow \lambda x. x + x$$

Then we have

$$(\lambda f. \lambda x. f (f x)) \text{ double}$$

$$\longrightarrow \lambda x. \text{double} (\text{double } x)$$

$$\longrightarrow \lambda x. \text{double} ( (\lambda x. x + x) x)$$

$$\longrightarrow \lambda x. \text{double} (x + x)$$

$$\longrightarrow \lambda x. (\lambda x. x + x) (x + x)$$

$$\longrightarrow \lambda x. (x + x) + (x + x)$$

Then $s \longrightarrow t$, if

$s \ \beta$-reduces (or $\eta$-expands, if one allows the $\eta$-rule) to $t$ or there exists an instantiation $s' \longrightarrow t'$ s.t. $s'$ is a subterm of $s$ and $t$ is the result of replacing this subterm in $s$ by $t'$.

$s'$ is called as usual a redex of $s$.

When referring to ordinary term rewriting rules, then for a term $t$ to have subterm $s$ meant essentially that there is a term $t'$ in which a new variable $x$ occurs exactly once, and $t = t'[x := s]$.

Replacing this subterm by $s'$ means that we replace $t$ by $t'[x := s']$. 
What does Subterm Mean?

When referring to $\lambda$-terms, this is no longer the case:

- Assume for instance the rewrite rule $x + 0 \longrightarrow_{\text{Rule}} x$.
- $\lambda x. x + 0$ has subterm $x + 0$, but there is no term $t$ s.t. $\lambda x. x + 0 = t[y := x + 0]$:
  - If we substitute for instance in $\lambda x. y$ by $x + 0$ we obtain $\lambda z. x + 0$.

The reason is that when matching a rewrite rule, free variables in the instantiation of the rule used might become bound.

So we can apply $x + 0 \longrightarrow_{\text{Rule}} x$ to $\lambda x. x + 0$ and have therefore $\lambda x. x + 0 \longrightarrow \lambda x. x$.

Replacing a subterm by another subterm is to be understood verbally.

Higher Order Rewrite Systems

The full definition of so called higher order term rewriting systems imposes more restrictions on the reduction rules.

For our purposes the naive interpretation just presented suffices.

Reduction to Closed Terms

One can always replace term rewriting rules for the $\lambda$-calculus by one in which for all rules $s \longrightarrow_{\text{Rule}} t$ we have that $s, t$ are closed.

This can be done in such a way that equality (modulo the rewriting rules, $\beta$ and possibly $\eta$) in both systems coincide:

Assume a rule $s \longrightarrow_{\text{Rule}} t$

and let $x_1, \ldots, x_n$ be the free variables in $s$.

Then replace this rule by

$$
\lambda x_1, \ldots, x_n.s \longrightarrow_{\text{Rule'} \lambda x_1, \ldots, x_n.t}.
$$

Proof

We write in the following $\overline{x}$ for $x_1, \ldots, x_n$.

Assume a term $r$ reduces using this rule in the original system to a term $u$:

Then $r$ contains a subterm of the form $s'$ where $s'$ is the result of substituting in $s x_i$ by some terms $t_i$.

Let $t'$ be the result of substituting in $t x_i$ by $t_i$. Then $u$ is the result of replacing $s'$ in $r$ by $t'$.

Let then $r'$ be the result of replacing $s'$ by $(\lambda \overline{x}.s) t_1, \ldots, t_n$, and $u'$ be the result of replacing in $s s'$ by $(\lambda \overline{x}.t) t_1, \ldots, t_n$.

Then we have $r =_\beta r' \longrightarrow_{\text{Rule'}} u' =_\beta u$, so the reduction can be simulated in the second system.
Proof

On the other hand, if \( r \rightarrow u \) by using in the second system the rule \( \lambda \overline{x}. s \rightarrow \text{Rule } \lambda \overline{x}. t \), then \( r \rightarrow u \) in the previous system by using the rule \( s \rightarrow \text{Rule } t \).

- \( r \) contains a subterm equal to \( \lambda \overline{x}. s \) and \( u \) is the result of substituting this subterm in \( r \) by \( \lambda \overline{x}. t \).
- But then \( r \) contains the subterm \( s \) and \( t \) is the result of substituting this subterm in \( r \) by \( t \).

Example

We can replace the rewriting rules

\[
x + 0 \rightarrow x
\]
\[
x + S \ y \rightarrow S \ (x + y)
\]

by

\[
\lambda x. x + 0 \rightarrow \lambda x. x
\]
\[
\lambda x, y. x + S \ y \rightarrow \lambda x, y. S \ (x + y)
\]

That

\[
S \ (0 + S \ 0) \rightarrow S \ (S \ (0 + 0)) \rightarrow S \ (S \ 0)
\]

becomes in the new system

\[
S \ (0 + S \ 0) = S \ ((\lambda x, y. x + S \ y) \ 0 \ 0)
\]
\[
= S \ ((\lambda x, y. S \ (x + y)) \ 0 \ 0) = S \ (S \ (0 + 0))
\]
\[
= S \ (S \ ((\lambda x. x + 0) \ 0)) \rightarrow S \ (S \ ((\lambda x. x) \ 0)) = S \ (S \ 0)
\]

Extended Typed \( \lambda \)-Calculus

Finally, we can combine the typed \( \lambda \)-calculus (with or without products, with or without \( \eta \)-expansion) with term rewriting rules.

- We have to make the following adaptations:
  - We assign a type to each additional constant.
  - The set of typed \( \lambda \)-terms is then introduced by the same rules as before, but we have as additional rule:
    - If \( c \) is a constant of type \( \sigma \), then we have
      \[
      \Gamma \Rightarrow c : \sigma
      \]

Example

Assuming \((+): \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}, \) and writing as usual \( r + s \) for \((+) \ r s \) we have the following derivation of

\[
l \lambda x : \text{nat}. x + x : \text{nat} \rightarrow \text{nat}:
\]

\[
x : \text{nat} \Rightarrow (+) : \text{nat} \rightarrow \text{nat} \quad x : \text{nat} \Rightarrow x : \text{nat}
\]

\[
\frac{x : \text{nat} \Rightarrow (+) \ x \ x : \text{nat}}{(\lambda x : \text{nat}. x + x) : \text{nat} \rightarrow \text{nat}}
\]

The left most leaf in this derivation follows by the rule for the constant \((+)\).
Example

Then we have

\[(\lambda f : \text{nat} \to \text{nat} \cdot \lambda x : \text{nat}. f (f \, x)) \text{ double} \]
\[\to \lambda x : \text{nat}. \text{ double} (\text{ double } x) \]
\[\to \lambda x : \text{nat}. \text{ double} ((\lambda x : \text{nat}. x + x) \, x) \]
\[\to \lambda x : \text{nat}. \text{ double} (x + x) \]
\[\to \lambda x : \text{nat}. (\lambda x : \text{nat}. x + x) \, (x + x) \]
\[\to \lambda x : \text{nat}. (x + x) + (x + x) \]

Extended Typed $\lambda$-Calculus

- Instantiations of a rule $\Gamma \Rightarrow s \to \text{Rule } t : \sigma$ are now obtained by replacing variables $x$ of type $\tau$ by terms $r : \tau$ (possibly depending on some context $\Delta$).

- Reductions w.r.t. the rules are obtained by replacing subterms $r : \sigma$, which coincide with the left hand side of an instantiation of a rule $r \to r' : \sigma$ by the right hand side $r'$.

Example

- Assume
  - ground type $\text{ nat}$,
  - constants $(+) : \text{ nat} \to \text{ nat} \to \text{ nat}$ (written infix, i.e. $r + s \text{ for } (+) \, r \, s$),
  - and $\text{ double } : \text{ nat} \to \text{ nat}$.

- and the reduction rule
  - $\text{ double } \to (\lambda x : \text{nat}. x + x) : \text{ nat} \to \text{ nat}$.
Example

Then we have

\[(\lambda f : \text{nat} \to \text{nat}. \lambda x : \text{nat}. f(f(x))) \text{ double}\]
\[\to \lambda x : \text{nat}. \text{double} (\text{double } x)\]
\[\to \lambda x : \text{nat}. \text{double} ((\lambda x.x + x) x)\]
\[\to \lambda x : \text{nat}. \text{double} (x + x)\]
\[\to \lambda x : \text{nat}. (\lambda x.x + x) (x + x)\]
\[\to \lambda x : \text{nat}. (x + x) + (x + x)\]

Curried/Uncurried Functions

- In the \(\lambda\)-calculus with products, there are two versions of a function \(f\) taking two integers and returning an integer:
  - \(f_1 : (\text{int} \times \text{int}) \to \text{int}\)
  - \(f_2 : \text{int} \to \text{int} \to \text{int}\).
- We say
  - that \(f_1\) is in **Uncurried** form,
  - and \(f_2\) is in **Curried** form.
- The name “Curry” honours Haskell Curry.
- The application of these two functions to arguments \(x\) and \(y\) is written
  - \(f_1(x, y)\),
  - \(f_2 x y\).

Products with many Components

- We write \(\sigma_1 \times \cdots \times \sigma_n\) for \((\cdots((\sigma_1 \times \sigma_2) \times \sigma_3) \cdots \times \sigma_n)\).
- Define \(a_1 : \sigma_1, \ldots, a_n : \sigma_n\)
  \(\langle a_1, \ldots, a_n \rangle := \langle \cdots \langle a_1, a_2 \rangle, a_3 \rangle, \cdots a_n \rangle : \sigma_1 \times \cdots \times \sigma_n\)
- One can easily define corresponding projections \(\pi^n_i : (\sigma_1 \times \cdots \times \sigma_n) \to \sigma_i, \text{s.t.}\)
  \[\pi^n_i(\langle a_1, \ldots, a_n \rangle) =_\beta a_i\]

Haskell Brooks Curry

Haskell Brooks Curry
(1900 - 1982)
Curried/Uncurried Functions

The above generalizes to functions with arbitrarily (but finitely) many arguments of different type.

The **Curried version** of a function $f$ with arguments of types $\sigma_1, \ldots, \sigma_n$ and result type $\rho$ is of type $\sigma_1 \to \cdots \to \sigma_n \to \rho$.

The **Uncurried version** of it has type $(\sigma_1 \times \cdots \times \sigma_n) \to \rho$.

**Currying**

We can obtain from the Uncurried form $f_{\text{Uncurry}}$ of a function its Curried form $f_{\text{Curry}}$ by

$$f_{\text{Curry}} = \lambda x_1, \ldots, x_n. f_{\text{Uncurry}} \langle x_1, \ldots, x_n \rangle$$

Again we can define

$$\text{Curry} : (\sigma_1 \times \cdots \times \sigma_n) \to \sigma_1 \to \cdots \to \sigma_n \to \rho$$

Curry := $\lambda f, x_1, \ldots, x_n. f \langle x_1, \ldots, x_n \rangle$

s.t. Curry $f_{\text{Uncurry}} \to_{\beta} f_{\text{Curry}}$.

This transformation is called **Currying**.

It is an easy exercise to show Curry (Uncurry $f$) = $\beta, \eta$ $f$

and Uncurry (Curry $f$) = $\beta, \eta$ $f$.

**Uncurrying**

We can obtain from the Curried form $f_{\text{Curry}}$ of a function its Uncurried form $f_{\text{Uncurry}}$ by

$$f_{\text{Uncurry}} = \lambda x. f_{\text{Curry}} \pi_0(x) \cdots \pi_n(x)$$

where $\pi^n_i : (\sigma_1 \times \cdots \times \sigma_n) \to \sigma_i$ are the projections.

One can as well define a $\lambda$-term

$$\text{Uncurry} : (\sigma_1 \to \cdots \sigma_n \to \rho) \to (\sigma_1 \times \cdots \times \sigma_n) \to \rho$$

Uncurry := $\lambda f, x. f \pi_0(x) \cdots \pi_n(x)$

s.t. Uncurry $f_{\text{Curry}} \to_{\beta} f_{\text{Uncurry}}$.

This transformation is called **Uncurrying**.

(Un)Currying in Programming

The Uncurried form of a function corresponds to the form functions are presented usually outside functional programming.

In functional programming one often prefers the Curried form.

This allows to apply a functional partially to its arguments.

E.g. if we take $(+)$ as usual in Curried form, then

$(+)$ 3 : int $\to$ int is the function taking $x$ and returning $x + 3$.

For instance $\text{map} ((+) 3) [1, 2, 3] = [4, 5, 6]$.

If we apply the function increasing every $x$ by 3 to the list $[1, 2, 3]$, we obtain the result of incrementing each list element by 3, i.e. $[4, 5, 6]$.
(Un)Currying in Programming

- One usually avoids in functional programming (and as well in Agda) the formation of products (or record types).
- The packing and unpacking of products makes programming often harder.
- E.g. instead of defining a function \( f : \sigma \rightarrow (\rho \times \tau) \) it is usually better to form two functions \( f_1 : \sigma \rightarrow \rho \) and \( f_2 : \sigma \rightarrow \tau \), (which are often defined simultaneously).

(e) Basic Derivations in Agda

- Agda is based on dependent type theory.
- This extends the simply typed \( \lambda \)-calculus.

Notations in Agda

- In type theory, one uses single colons `:` for "has type", as we did above.
- In Haskell and as well in Cayenne, `:` is used in lists, and `::` is used for "has type".
- In order to be close to Haskell and Cayenne, it was decided to use in Agda as well `::` (although lists haven't been introduced yet).
- In this lecture we will usually use `::`, except when referring to explicit Agda code (then `::` is used).

Notations in Agda

- In Agda one writes \( \lambda (x :: A) \rightarrow r \) for \( \lambda x : A. r \).
  - \( A \) can often be inferred by Agda automatically, using menu `Solve (C-c =)`
    (When presenting keyboard-macros, "C-c" stands for "Control-c").
  - When presenting Agda code we will write \( \lambda (x :: A) \rightarrow r \) for the above, so \( \lambda \) means \( \backslash \) and \( \rightarrow \) means \( \rightarrow \) in real Agda code.
  - When reasoning in type theory itself (outside Agda), we use standard type theoretic notation \( \lambda x : A. r \).
    - We sometimes omit \( A \), and write simply \( \lambda x. r \).
Postulate

In Agda one has no predefined types, all types have to be defined explicitly (e.g. the type of natural numbers, the type of Booleans, etc.).

In order to obtain ground types with no specific meaning (like \( \text{\textit{o}} \) above), we have to postulate such types, (or use packages as introduced later).

In Agda the lowest type level, which corresponds to types in the simply typed \( \lambda \)-calculus, is called for historic reasons \( \text{\textit{Set}} \).

So in order to introduce a ground type \( A \) we write:

\[
\text{postulate } A :: \text{Set}
\]

We can now introduce other constants. For instance, in order to introduce a function from \( A \) to \( B \) where \( A \) and \( B \) are ground types, and an element of type \( A \), we write the following:

\[
\text{postulate } A :: \text{Set} \\
\text{postulate } B :: \text{Set}. \\
\text{postulate } f :: A \rightarrow B. \\
\text{postulate } a :: A.
\]

See examplePostulate1.agda

Basic Usage of Agda

Instructions on how to install Agda can be found under http://www-comp.sci.swan.ac.uk/~csetzer/othersoftware/agda/agdainstallation.html

The installation will provide an Emacs/XEmacs mode for agda files.

If a file with extension .agda is loaded into Emacs/XEmacs, then this mode is invoked.

There are two menus:

- One from the top menu line of Emacs/XEmacs, invoked with a right mouse click.
- One which is activated at any position in the buffer, again invoked by a right mouse click.

(Under Emacs, this menu is only active when one is inside a goal – see below).

The latter one depends on where one is in the buffer.
Basic Usage of Agda

- On the menu, with each menu a keyboard short cut is presented. It is advisable to learn the most frequently used ones.
- In order to type check the buffer, use Load-Buffer (C-c C-x C-b).
- Agda allows to load several buffers in sequence, and to keep the old definitions.
  - E.g. one has one file, containing the definition of the natural numbers, and one main file using it.
  - Then one first loads the file containing the natural number definitions, and then the main file.

Basic \(\lambda\)-Terms

- Definitions introduced must be consistent – each definition can occur only once.
- When moving to a different file, which associates with the same identifier the same or a different definition, one gets an error, since the other definition from the previous file is still present.
  - In order to get rid of the previous definition, one has to restart Agda first, using (Re)start Agda (C-c C-x C-c).
  - Can be used as well in case Agda crashes (which happens sometimes, since evaluation of terms can be very time and space intensive).
  - Then one has to load the buffer.

- Assuming the above postulates, we can now introduce new terms.
- We have to give a name and a type to each new definition.
- **Example:** Using the above postulates, we can define \(b := f \ a : B\) as follows:
  \[
  b :: B = f \ a
  \]

---

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 1

177

---

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 1

178

---

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 1

179

---

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 1

180
**λ-Terms**

postulate $A :: \text{Set}$
postulate $B :: \text{Set}$.
postulate $f :: A \rightarrow B$.
postulate $a :: A$.

Instead of defining $\lambda$-terms by using $\lambda$ directly, it is usually more convenient to use a notation of the following kind:

$$g \ (a :: A) :: A = a$$

Note that in the above example, the local $a$ overrides the global $a$.

See examplePostulate3.agda

---

**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 inside a goal, the menu active from inside the buffer,

- when using XEmacs, changes to the `goal-menu`;
- 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 using menu `Text State`.

- When reloading the buffer one gets back to the state in which goals have special status.

---

In Agda one can leave terms or types, which one doesn’t know yet, open.

The part left open is written as `{! !}`, and we say “goal” for it.

Example:

$$g \ (x :: A) :: A = \{! !\}$$

exampleGoal1.agda
Goals

- Inside a goal we can as well find out the current context:
  - Using menu **Context**.
  - In our example Agda shows:
    
    \[
    a :: A \\
g :: (x :: A) \to A \\
a :: A \\
f :: A \to B \\
B :: Set \\
A :: Set
    \]

Indention Sensitivity

- Agda is indentation sensitive.

- So often instead of having parentheses “\{(Code)\}”, as in other languages, all lines belonging to \(\{\text{Code}\}\) 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:

  ```
  postulate A :: Set 
  postulate B :: Set 
  ```

- We have to type instead:

  ```
  postulate A :: Set 
  postulate B :: Set 
  ```

- The following causes an error:

  ```
  a' :: A 
  = a 
  ```

- We have to type instead:

  ```
  a' :: A 
  = A. 
  ```

Derivations in Agda

- In Agda, rules are implicit.

- The rule

  \[
  f : A \to B \\
a : A 
  \]

  corresponds to the following:

  Assume we have introduced:

  ```
  g :: D \to E, d :: D. 
  ```

  and want to solve the goal

  ```
  \{! !\} :: E. 
  ```

  `exampleSimpleDerivation4.agda`
Derivations in Agda (Cont.)

- Then we can fill this goal by typing in \(g \ d\):
  \[
  \{! \ g \ d \ ! \} :: E
  \]
- If we then choose goal-menu **Refine (C-c C-r)**, the system shows:
  \[
  g \ d :: E.
  \]

Refinement

- Assume in the above situation that we don’t know what to insert.
  - We only guess that it has to be of the form \(g\) applied to some arguments.
  - We can see this since the result type of \(g\) is \(E\):
    \[
    \(g : D \rightarrow E\).
    \]
- Then we can insert \(g\) and use menu **Refine (C-c C-r)**
  - The system shows \(g \ \{! \ !\} :: E\).
- We can ask for the type of the new goal \(\{! \ !\}\), and get:
  \[
  \{! \ !\} :: D
  \]
- Now we can solve this goal by filling in \(d\) and using refine:
  \[
  g \ d :: E.
  \]

Judgements

- In ordinary functional programming, it is easy to determine the correctly formed types.
  - In dependent type theory the type structure is richer and more complicated.
- Proof steps are required to conclude that something is a type.

Judgements

- Therefore we have not only the judgement as in functional programming
  \[
  a : A
  \]
  but as well a typing judgement \(A\) is a type, written (as we have already seen)
  \[
  A : \text{Set}
  \]
- Before deriving \(a : A\), where \(A\) is a constant, we first have to show \(A : \text{Set}\).
Equality Judgements

Agda will identify terms which have the same normal form.
E.g. \( s := (\lambda(x : A).x) \) \( r \) and \( r \) will be identified.
If one needs at some place \( r \), one can insert \( s \) instead of \( r \) and vice versa.
In Agda this is done automatically, the user doesn’t see such equalities.
There is not even a direct command available in Agda, which allows to check whether two terms are equal (this could probably be added easily).

Example

Postulate \( A :: \text{Set} \)
Postulate \( a :: A \)
Postulate \( f :: A \to \text{Set} \)
\[ g (a :: A) :: A = a \]
\[ a' :: A = ga \]
\[ p (x :: fa) :: (fa') = \{ ! ! \} \]

\texttt{exampleSimpleEquality2.agda}

Since \( a' = g a = a \), we can solve the goal by using \( x \).

Equality Judgements

When using the simply typed \( \lambda \)-calculus, we could separate the derivation of \( \lambda \)-terms, from reductions.
When using dependent type theory as in Agda, reductions and derivations have to be integrated.
Traditionally, instead of introducing reductions, one introduces in dependent type theory equalities between terms.
Written as
\[ r = s : A \]
for \( r \) and \( s \) are equal elements of set \( A \).

Example

The rule expressing that \( \pi_0(\langle a, b \rangle) \to a \) reads in this style as follows:
\[
\frac{a : A \quad b : B}{\pi_0(\langle a, b \rangle) = a : A}
\]

\( = \) is not directed, so we have as well the rule
\[
\frac{a = b : A}{b = a : A}
\]

We can therefore derive:
\[
\frac{a : A \quad b : B}{\pi_0(\langle a, b \rangle) = a : A}
\]
\[
\frac{a = \pi_0(\langle a, b \rangle) : A}{\pi_0(\langle a, b \rangle) = a : A}
\]
Equality Judgements (Cont.)

- We will have as well equality between types, written as
  \[ A = B : \text{Set} \]
- This is something novel in dependent type theory.
  - In simple type theory, there is only one way of writing a type.
- For instance if we assume \( f : A \rightarrow \text{Set} \), then if \( a = a' : A \) then \( f \ a = f \ a' : \text{Set} \).

(f) Dependent Judgements

- As for the simply typed \( \lambda \)-calculus, in dependent type theory, judgements might depend on a context.
- So we obtain judgements of the form

\[
\begin{align*}
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow A : \text{Set} \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow A = B : \text{Set} \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow s : A \\
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow s = t : A
\end{align*}
\]

Interactive Theorem Proving, CS_336, Lentterm 2004, Sec. 1

Four Judgements

So we have the following 4 types of judgements:

- \( A : \text{Set} \)  
  “\( A \) is a type”.
- \( a : A \)  
  “\( a \) is of type \( A \)”.
- \( A = B : \text{Set} \)  
  “\( A \) and \( B \) are equal types”.
- \( a = b : A \)  
  “\( a \) and \( b \) are equal elements of type \( A \)”.

In Agda, only \( A : \text{Set} \) and \( a : A \) are explicit.

Need for Context Rules

- To derive such judgements requires that we know

\[
\begin{align*}
x_1 : A_1, \ldots, x_n : A_n & \Rightarrow A_1 : \text{Set} \\
x_1 : A_1, x_2 : A_2 & \Rightarrow A_2 : \text{Set} \\
x_1 : A_1, x_2 : A_2 & \Rightarrow A_3 : \text{Set} \\
\cdots
\end{align*}
\]

\[
\begin{align*}
x_1 : A_1, x_2 : A_2, \ldots, x_{n-1} : A_{n-1} & \Rightarrow A_n : \text{Set}
\end{align*}
\]
Context Rule

- Note that we didn’t require derivations as above in the simply typed $\lambda$-calculus, since it was easy to verify whether something is a valid type.
- In case of dependent types $A : Set$ requires a derivation.
- It can be as complicated to derive $A : Set$ as it is to derive a judgement $b : B$.
  One can compute from an unchecked judgement $a : A$ an unchecked type expression $B$ s.t. $a : A$ holds iff $B : Set$ holds.

Interactive Theorem Proving, CS_336, Lentterm 2004, Sec. 1

Five Dependent Judgements

- So we have therefore 5 dependent judgements:
  
  \[
  x_1 : A_1, \ldots, x_n : A_n \Rightarrow A : Set \\
  x_1 : A_1, \ldots, x_n : A_n \Rightarrow A = B \\
  x_1 : A_1, \ldots, x_n : A_n \Rightarrow s : A \\
  x_1 : A_1, \ldots, x_n : A_n \Rightarrow s = t : A \\
  x_1 : A_1, \ldots, x_n : A_n \Rightarrow \text{Context}
  \]

Interactive Theorem Proving, CS_336, Lentterm 2004, Sec. 1

Example

- The assumption rule, which in case of the simply typed $\lambda$-calculus read:
  \[
  \Gamma \Rightarrow x : \sigma \quad \text{(if } x : \sigma \text{ occurs in } \Gamma) 
  \]
  reads in dependent type theory as follows (assuming that $x : A$ occurs in $\Gamma$):
  
  \[
  \Gamma \Rightarrow \text{Context} \quad \Gamma \Rightarrow x : A
  \]

- Similarly we have to deal with the rule introducing constants.
Notations for Judgements, Contexts

- \( \theta \) will in the following denote an arbitrary non dependent judgement, i.e. one of the following:
  - \( A : \text{Set}, \)
  - \( A = B : \text{Set}, \)
  - \( a : A, \)
  - \( a = b : A. \)
- \( \Gamma, \Delta \) will usually denote contexts.
- We have the same notations as before, i.e.
  - \( \Gamma, \Delta \) is the result of concatenating contexts \( \Gamma, \Delta, \)
  - \( \Gamma, x : A \) is the result of extending the context \( \Gamma \) by \( x : A, \)
  - \( \emptyset \) is the empty context.
  - We write for \( \emptyset \Rightarrow \theta \) usually simply \( \theta. \)

Example: Derivation of double

(See exampleDoubleString2.agda.)

- We derive
  
  \[
  \text{double} := (\lambda x. \text{concat } x \ x) : ((x : \text{String}) \rightarrow \text{String}) \]
  
  in Agda, assuming definitions of \( \text{String} \) and \( \text{concat}. \)
- We start with
  
  \[
  \text{double} \ (x :: \text{String})
  \]
  
  \[
  :: \text{String}
  \]
  
  \[
  = \{! !\}
  \]
- We can insert into the goal \( \text{concat}: \)

\[
\text{double} \ (x :: \text{String})
\]

\[
:: \text{String}
\]

\[
= \{! \text{concat } !\}
\]

Contexts in Agda

- In Agda, we have no explicit judgements depending on contexts.
  - Not needed, since we don’t derive judgments using rules directly.
  - However, if we have the open judgement

\[
f \ (x :: B)
\]

\[
:: A
\]

\[
= \{! !\}
\]

- Then we can make use of \( x :: B \) for refining the goal.
- So we have to solve the goal in context \( x :: B. \)
- This context can be shown using goal menu Context.
- See exampleShowContext.agda.

Example: Derivation of double

- When using goal-menu refine, we obtain:

\[
\text{double} \ (x :: \text{String})
\]

\[
:: \text{String}
\]

\[
= \{! !\}
\]

- We can check now using goal-menu Type of Goal (or Type of Goal (unfolded)) that the two new goals require both type \( \text{String}. \)
- We can check using goal-menu Context that the context of both goals contain \( x :: \text{String}. \)
Example: Derivation of double

- We insert $x$ into the first goal and refine:

\[
\text{double } (x :: \text{String}) \\
:: \text{String} \\
= \text{concat } x \{! !\}
\]

- Doing the same with the second goal gives:

\[
\text{double } (x :: \text{String}) \\
:: \text{String} \\
= \text{concat } x x
\]

- We are done.

double in Type Theory

The remaining derivation is as follows (using the above derivation):

\[
\begin{align*}
\text{x : String} & \Rightarrow \text{Context} \\
\text{x : String} & \Rightarrow \text{concat : String} \\n\text{x : String} & \Rightarrow \text{concat : String} \\
\text{x : String} & \Rightarrow \text{concat : String} \\
\text{double := } & \lambda x : \text{String}. \text{concat } x x
\end{align*}
\]

(g) Dependent Types

- Assume we want to assign a type to a sorting function \text{sort} on lists of natural numbers.

- In most programming languages, the type of it is essentially

\[
\text{sort : NatList} \rightarrow \text{NatList}
\]

for the type of lists of natural numbers \text{NatList}.

- In dependent type theory, we can demand more correctness, namely that its type is

\[
\text{sort : NatList} \rightarrow \text{SortedList}
\]

- We assume some notion of \text{NatList} (list of natural numbers).
SortedList

What is SortedList?
- An element of SortedList is a list which is sorted.
- It is a pair \( \langle l, p \rangle \) s.t.
  - \( l \) is a NatList.
  - \( p \) is a proof or verification that \( l \) is sorted:
    - \( p : \text{Sorted}(l) \).

Sort Lists (Cont.)

An element of \( \text{Sorted}(l) \) will be a \textbf{proof} that \( l \) is sorted.
- If \( l \) is \textit{sorted}, then \( \text{Sorted}(l) \) will be provable, and therefore \textbf{will have an element}.
  - It is possible to write a program which computes an element of \( \text{Sorted}(l) \).
- If \( l \) is \textit{not sorted}, then \( \text{Sorted}(l) \) will have no proof and it will therefore \textbf{no element}.
  - Then it is not possible to write a program which computes an element of \( \text{Sorted}(l) \).

Sorted Lists

For the moment, ignore what is meant by \( \text{Sorted}(l) \) as a type.
- Only important: Sorted depends on \( l \).
  - \( \text{Sorted}(l) \) is a predicate expressed as a type.
- Elements of SortedList are pairs \( \langle l, p \rangle \) s.t.
  - \( l : \text{NatList} \).
  - \( p : \text{Sorted}(l) \).
- Sorted\( (l) \) is a dependent type.

The Dependent Product

Then the pair \( \langle l, p \rangle \) will be an element of
\[
\text{SortedList} := (l : \text{NatList}) \times \text{Sorted}(l).
\]
- SortedList is the type of pairs \( \langle l, p \rangle \) s.t.
  - \( l : \text{NatList} \),
  - \( p : \text{Sorted}(l) \).
  - called the \textbf{dependent product}
- sort : \( \text{NatList} \rightarrow ((l : \text{NatList}) \times \text{Sorted}(l)) \) expresses:
  - sort converts lists into sorted lists.
The Dependent Function Type

- From a sorting function we know more:
  - It takes a list and converts it into a sorted list with the same elements.
- Assume a type (or predicate) \text{EqElements}(l,l') standing for
  - \( l \) and \( l' \) have the same elements.

(\text{Architecture}) Dependent Types in Programming

Dependent types are often \textbf{needed in programming}, even if no verification is needed.

We give some examples:

- In Java, a relatively big library of \textbf{"collection classes"} is available.
  - Provides implementations of lists, sets, hash tables, etc.
  - It would be nice to have \textbf{"lists of type} \( A \)."
  - However \textbf{this is a dependent type}, depending on a type \( A \).
  - Cannot be expressed in Java.
  - Instead, in Java only \textbf{lists of elements of type} \textbf{Object} are available.

The Dependent Function Type

- A refined version of \text{sort} has type:

\[ (l : \text{NatList}) \rightarrow ((l' : \text{NatList}) \times \text{Sorted}(l') \times \text{EqElements}(l, l')) \]

- \text{sort}(l) is a list, which is sorted and has the same elements.
- \text{sort} is a program, which takes a list and returns a sorted list with the same elements.

The type of \text{sort} is an instance of the dependent function type:

- The result type depends on the arguments.
Example

Assume a class `StudentEntry`.

If we have a list `listOfStudentEntries`, and add to it an element `studentEntry` of type `StudentEntry`, this element will first be converted (upcasted) to type `Object`.

If we retrieve an element (e.g. the first element) of `listOfStudentEntries`, we obtain an element of `Object`.

If it was originally a `StudentEntry`, we can cast this element down to `StudentEntry`.

However, whether we have an element of `StudentEntry`, cannot be determined at compile time, only at run time.

Polymorphism

What is needed is a weak form of dependent types, called **polymorphism**.

Types might depend on other types but not on elements of types.

In C++, this form of dependency is available (called **templates**).

One writes for instance `List<A>` for lists of type `A`.

In Java, it might be available in the next release 1.5.

In C#, it will be available in the next release.

In Haskell and ML it is available.

E.g. \( \lambda x. x : \alpha \Rightarrow \alpha \rightarrow \alpha \), which means that \( \lambda x. x \) is of type \( \alpha \rightarrow \alpha \) for every type \( \alpha \).

Dependent Types in Programming

**Matrix multiplication** is an operation, which takes three natural numbers \( n, m, k \), an \( n \times m \)-matrix and an \( m \times k \)-matrix, and has as result an \( n \times k \)-matrix.

The type of this function is a **dependent type**: The types of \( n \times m \)-matrices, of \( m \times k \)-matrices and of \( n \times k \)-matrices depend on \( n, m, k \).

Matrix Multiplication

Usually, this problem is solved by

- taking matrices which are big enough and restricting the operation to \( n \times m, m \times k \) and \( n \times k \) sub-matrices,
- waste of memory
- or by dynamically allocating arrays.
- This means memory allocation has to be done at run time.
- In both solutions, checking that the dimensions are in accordance has to be done at **run-time**.
Type of Matrix Multiplication

Let $N$ be the type of natural numbers (i.e. $0, 1, \ldots$; $N$ will be introduced later).

Let $\text{Mat}(n, m)$ be the type of $n \times m$-matrices. (Will be introduced later).

Then matrix multiplication has type

$$(n : N) 
\rightarrow (m : N) 
\rightarrow (k : N) 
\rightarrow \text{Mat}(n, m) 
\rightarrow \text{Mat}(m, k) 
\rightarrow \text{Mat}(n, k)$$

Type of Matrix Multiplication (Cont.)

A shorter notation for this type is

$$(n, m, k : N) 
\rightarrow \text{Mat}(n, m) 
\rightarrow \text{Mat}(m, k) 
\rightarrow \text{Mat}(n, k)$$

Dependent Types in Programming

Digital Components.

A digital component (e.g. a logic gate) with $n$ inputs and $m$ outputs can be considered as a function $\text{Bool}^n \rightarrow \text{Bool}^m$.

In general such a component is a triple consisting of

- $n$, the number of inputs,
- $m$, the number of outputs,
- a function $f : \text{Bool}^m \rightarrow \text{Bool}^m$.

The type of $f$ depends on $n$ and $m$, an example of a dependent type.

Predicates are dependent types.

See the types of sort above.
Dependent Types in Linguistics

- **Aarne Ranta** has used dependent types in **linguistics**: In a sentence like “The man goes home”, the predicate
  in grammar the predicate is “what is said about the subject of a sentence”; it consists essentially of the verb and what completes the verb;
  here it is “goes home” depends on, whether the subject
  in grammar the subject is the noun or its equivalent about which a sentence is predicated; here it is “The man” is **singular** (then the predicate is “goes home”) or **plural** (then the predicate is “go home”).

Dependent Types in Linguistics

- Aarne Ranta constructed **grammars based on dependent types**, and used them for translating sentences between different languages automatically.

**2. The Logical Framework**

(a)
(b)
(c)
(d)
(e)
(f)
Four Kinds of Rules

For each type construction we have usually 4 kinds of rules:

1. Formation Rules.
2. Introduction Rules.
3. Elimination Rules.

Additionally there are equality versions of the formation, introduction and elimination rules.

Logical Framework

Preliminarily, we will be using type theory without the logical framework.

For instance, below we will introduce

\[ \text{List}(A) : \text{Set} \]

for any \( A : \text{Set} \), the set of lists of elements of \( A \).

(1) Formation Rules

The formation rules introduce new types.

Each type construction has one such rule.

The conclusion of such a rule will have the form:

\[ C(a_1, \ldots, a_n) : \text{Set} \]

where \( C \) is a type-constructor,

\( a_1, \ldots, a_n \) are its arguments.

\( n = 0 \) is possible, in which case we write \( C \) instead of \( C() \).

Later, we will introduce higher levels Type, Kind, . . . .

Then we have formation rules with conclusion

\[ C(a_1, \ldots, a_n) : \text{Type} \text{ or } \text{Kind, etc.} \]

Logical Framework

Until we have introduced the full logical framework, it doesn’t make sense to talk about \( \text{List} \) itself, which would have type

\[ \text{List} : \text{Set} \to \text{Set} \]

The problem is that \( \text{Set} \to \text{Set} \) doesn’t make sense without the logical framework.

The full logical framework is conceptually more difficult, that’s why we delay its introduction.
Logical Framework

When working later with the full logical framework, constructors will have their own type.

Then what is now \( C(a_1, \ldots, a_n) \)

will be written as \( C \ a_1 \ldots a_n \),

namely the application of the constructor \( C \) to arguments \( a_1, \ldots, a_n \).

For instance we will write then \( \text{List} \ A \) instead of \( \text{List}(A) \).

Example 1: The Set of Lists

\[
\begin{align*}
A : \text{Set} & \quad \text{(List-F)} \\
\text{List}(A) : \text{Set} & \\
\end{align*}
\]

- The type-constructor is \( \text{List} \).
- \( \text{List}(A) \) is the type of lists of type \( A \).

Logical Framework

Agda has the logical framework built in, so in Agda \( \text{List} \) will be a function \( \text{Set} \rightarrow \text{Set} \), in Agda notation:

\[
\text{List} \quad (A :: \text{Set}) \\
:: \text{Set} \\
= \ldots
\]

Ex. 2: The Set of Natural Numbers

Formation rule for the type of natural numbers:

\[
N : \text{Set} \quad \text{(N-F)}
\]

- The type-constructor is \( N \).
- Note that the formation rule for \( N \) has 0 premises (therefore the fraction bar is omitted).
Ex. 3: The Non-Dependent Product

- Formation rule for the non-dependent product:

\[
\frac{A : \text{Set} \quad B : \text{Set}}{A \times B : \text{Set}} \quad (\times - \text{F})
\]

- \(A \times B\) stands for \((\times)(A, B)\).
- The type-constructor is \((\times)\).

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 2(a) 236

Example 2: \((\times)\)

- Agda syntax for introducing the non-dependent product:

\[
(\times) \quad (A :: \text{Set}) \\
(B :: \text{Set}) \\
:: \text{Set} \\
= \ldots
\]

- \(\ldots\) is an Agda definition of this type (more about this later).
- \((\times)\) is ASCII symbol 215 (not the letter x).
- There are as well predefined versions of the product (and of the function type) in Agda (see later).

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 2(a) 238

Formation Rules in Agda

- The formation of a type is usually done by introducing a constant of a certain type.
- Example 1:

\[
\text{List} \quad (A :: \text{Set}) \\
:: \text{Set} \\
= \ldots
\]

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 2(a) 237

(2) Introduction Rules

- The introduction rule introduces elements of a type.
- The conclusion of such a rule will have the form

\[C(a_1, \ldots, a_n) : A\]

where

- \(A\) is a type introduced by the corresponding formation rule,
- \(C\) is a constructor or term-constructor,
- \(a_1, \ldots, a_n\) are terms.

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 2(a) 239
Introduction Rule, Example 1a

The set $\text{NatList}$ of lists of natural numbers with formation rule

$$\text{NatList} : \text{Set} \quad (\text{NatList-F})$$

has two introduction rules:

$$\begin{align*}
\text{nil : NatList} & \quad (\text{N-I-nil}) \\
n : N \quad l : \text{NatList} & \quad (\text{NatList-I-cons}) \\
\text{cons}(n, l) : \text{NatList}
\end{align*}$$

Introduction Rule, Example 1b

We generalise the previous example to lists of arbitrary type.

Lists of type $A$ have two introduction rules:

$$\begin{align*}
A : \text{Set} \quad & (\text{List-I-nil}) \\
\text{nil}_A : \text{List}(A) \\
a : A \quad l : \text{List}(A) & \quad (\text{List-I-cons}) \\
\text{cons}_A(a, l) : \text{List}(A)
\end{align*}$$

In case of the rule for nil, we needed the premise $A : \text{Set}$ in order to guarantee that we can form the type $\text{List}(A)$.

In case of the rule for cons, this premise is implicit in the premise $a : A$.

Conflicting Constructors

We shouldn’t use the same constructors for different sets. So if we want to use both $\text{NatList}$ and $\text{List}(A)$, we have to choose a notation like $\text{natnil}$ instead of $\text{nil : N}$, similarly for cons.

We will usually ignore this distinction, if it doesn’t cause confusion.

Example 2: Natural Numbers.

The natural numbers $N$ can be considered as being formed from two operations:

- $0$,
- $S$ where $S(n) = n + 1$.

Using these two operations we can form $0$, $S(0) = 1$, $S(1) = 2$, \ldots and therefore all natural numbers.

So the constructors of $N$ are $0$ and $S$.

The introduction rules of $N$ are:

$$\begin{align*}
0 : N & \quad (\text{N-I}_0) \\
n : N \quad S(n) : N & \quad (\text{N-I}_S)
\end{align*}$$
Canonical Elements

- **Canonical elements** of a type are those introduced by an introduction rule.
- Canonical elements therefore always start with a constructor.
- **Examples:**
  - 0, $S(2 + 3)$ in case of $N$.
  - Here 2 stands for $S(S(0))$ and 3 for $S(S(S(0)))$.
  - nil, cons$(1 + 1$, concat cons$(0$, nil) nil) in case of NatList.

Non-Canonical Elements

- Terms can usually be reduced further.
  - Example:
    $$2 + 3 = 2 + S(2) \rightarrow S(2 + 2).$$
- The underlying reduction system is essentially a term rewriting system combined with the $\lambda$-calculus.
- Therefore we can apply reductions to subterms.
- A term is a **non-canonical element** of a type, if it reduces to a canonical element of that type.
- Each element of a type (depending on the empty context) in dependent type theory will either be a canonical or a non-canonical element of that type.
- Consequence of the normalisation theorem.

Non-Canonical Elements

- E.g. $2 + 3$ is a non-canonical element of $N$, since $S(2 + 2)$ is a canonical element of $N$.
- However, we have $x : N \Rightarrow x : N$
- and $x$ doesn’t reduce to a canonical element of $N$.
- However, if we substitute for $x$ any closed element of $N$, we get a canonical or non-canonical element of $N$.

Constructors in Agda

- In Agda the constructor $C$ of type $A$ is written as $C@A$.
- If $A$ can be inferred automatically, we can replace the above by $C@_$. 
- As type-constructors, in Agda constructors are as in dependent type theory with the logical framework, i.e. we have
  $$\begin{align*}
  \text{nil@List N} &:: \text{List N} \\
  \text{cons@List N} &:: (n :: N) \\
  &\quad \rightarrow (l :: \text{List N}) \\
  &\quad \rightarrow \text{List N}
  \end{align*}$$
Constructors in Agda

Since notations like nil@ (List N) are usually too cumbersome, it is better to introduce abbreviations:

\[
\begin{align*}
nil &: \text{List N} \\
&= \text{nil@} \\
\text{cons} &: (n :: \text{N}) \\
&= (l :: \text{List N}) \\
&:: \text{List N} \\
&= \text{cons@}_n l
\end{align*}
\]

Note that the above introduces nil, cons for List N, and not for the general case List A for any type A. (That would require an extra argument A : Set.)

Example 2: Addition in N

\[
\begin{align*}
n : \text{N} & \quad m : \text{N} \quad (N-\text{El}_+) \\
\frac{n + m : \text{N}}{n + m : \text{N}}
\end{align*}
\]

Equality rules will express

\[
\begin{align*}
n + 0 &= n. \\
n + S(m) &= S(n + m).
\end{align*}
\]

The equality rules show that n is only a parameter, we are eliminating the second argument m.

Proceeding like this would require one elimination rule for each function from N we want to define.

Instead we will later introduce one general elimination rule, which will allow to introduce all functions we expect to be definable, including all primitive-recursive ones.

(3) Elimination Rules

Elimination rules allow to take an element of a type and compute from it an element of another type.

Example 1: The introduction rule for the non-dependent product is

\[
\begin{align*}
a : A & \quad b : B \\
\langle a, b \rangle &: A \times B
\end{align*}
\]

The elimination rules are the first and second projections:

\[
\begin{align*}
c : A \times B & \quad (\times-\text{El}_0) \\
\pi_0(c) &: A \\
\pi_1(c) &: B \\
c : A \times B & \quad (\times-\text{El}_1) \\
\pi_0(\langle a, b \rangle) &= a, \\
\pi_1(\langle a, b \rangle) &= b.
\end{align*}
\]

Elimination vs. Introduction Rules

Elimination rules invert the introduction rules.

In case of A × B, the canonical elements are of the form \langle a, b \rangle for a : A, b : B.

A non-canonical element of type A × B must reduce to a canonical element.
Elimination in Agda

- Elimination for built-in types has special notation.
- For user defined types, elimination is realized by case distinction.
- Example: Definition of addition in \(\mathbb{N}\):

\[
(+) \quad (n, m : \mathbb{N}) \\
\quad :: \mathbb{N} \\
= \text{case } m \text{ of} \\
\quad \{ \\
\quad \quad (Z) \rightarrow n; \\
\quad \quad (S \, m') \rightarrow S \, (n + m') \}\]

Equality Rules

- In case of \((+), (\text{Red})\) are the reductions
  \[n + 0 \rightarrow n.\]
  \[n + S(m) \rightarrow S(n + m).\]
  Note that the second argument is the argument which we are “eliminating”.
- So the computation of \(0 + (1 + 1)\) is as follows:

\[
0 + (1 + 1) = 0 + (S(0) + S(0)) \rightarrow 0 + S(S(0 + 0)) \rightarrow S(0 + S(0 + 0))
\]

The result is already in canonical form (but not in normal form since it can be reduced further).

(4) Equality Rules

- The canonical element for an element, which is the result of an elimination, can always be computed as follows:
  - Reduce the element to be eliminated to canonical form.
  - Then make one reduction step (Red).
  - The result will be a canonical or non-canonical element of the target type.
  - Reduce it to canonical form.
- For instance in case of \(A \times B\), (Red) are the reductions
  \[\pi_0((a, b)) \rightarrow a.\]
  \[\pi_1((a, b)) \rightarrow b.\]
Example (Equality Rule)

The first equality rule for $A \times B$ is as follows:

$$
\begin{align*}
\pi_0(\langle a, b \rangle) &= a : A \\
\pi_0(\langle a, b \rangle) &= b : B
\end{align*}
$$

($\times\text{-Eq}_0$)

In the first judgement we can derive $\pi_0(\langle a, b \rangle) : A$ as follows:

$$
\begin{align*}
a : A &
\quad b : B \\
\langle a, b \rangle : A \times B \\
\pi_0(\langle a, b \rangle) &= A
\end{align*}
$$

($\times\text{-I}$)

So it is derived by first introducing $\langle a, b \rangle$ and then eliminating it immediately.

The equality rule explains how to reduce that element (namely to $a : A$).

Example (Equality Rule, Cont)

The second equality rule for $\times$ is similar:

$$
\begin{align*}
a : A &
\quad b : B \\
\pi_1(\langle a, b \rangle) &= b : B
\end{align*}
$$

($\times\text{-Eq}_{11}$)

Example 2 (Equality Rule)

The first equality rule for $+$ is as follows:

$$
\begin{align*}
n : N \\
n + 0 &= n : N
\end{align*}
$$

($\text{N-Eq}_{+,0}$)

$n + 0 : N$ can be derived by first introducing $0 : N$ and then by eliminating it using $+$.

(The right hand side is an axiom, the left hand side has to be concluded using some derivation.)

$$
\begin{align*}
n : N &
\quad 0 : N \\
n + 0 &= N
\end{align*}
$$

($\text{N-El}_+$)

The equality rule explain how to reduce $n + 0$.

Example 3 (Equality Rule)

The second equality rule for $+$ is a s follows:

$$
\begin{align*}
n : N &
\quad m : N \\
n + S(m) &= S(n + m) : N
\end{align*}
$$

($\text{N-Eq}_{+,S}$)

$n + S(m) : N$ can be derived by first introducing $S(m) : N$ and then by eliminating it using $+$:

$$
\begin{align*}
m : N &
\quad S(m) : N \\
n + S(m) &= N
\end{align*}
$$

($\text{N-El}_+$)
Equality Rules in Agda

- Equality Rules in Agda are **implicit**.
- The notation for elimination however indicates already how the reductions take place.

\[
(+) \quad (n, m :: N) :: N = \text{case } m \text{ of}\{ (Z) \to n; \ (S \ m') \to S \ (n + m')\}
\]

- Functions corresponding to elimination are defined by telling how elimination operates.

Equality Versions of the Rules

- We have equality versions of the formation, introduction, and elimination rules.
- These express: if we replace the terms in the premises by equal ones, we obtain equal results.
- Example: Equality version of the formation rule for List:

\[
A = B : \text{Set} \quad \frac{A = B : \text{Set}}{\text{List}(A) = \text{List}(B)} \quad (\text{List-F}^=)
\]

- Example: Equality version of the formation rule for \(N\) (degenerated):

\[
N = N : \text{Set} \quad (\text{N-F}^=)
\]

Equality Versions of Rules

- Example: Equality version of the introduction rules for List (rule for nil is degenerated):

\[
\begin{array}{c}
A : \text{Set} \\
\text{nil}_A = \text{nil}_A : \text{List}(A)
\end{array}
\]

\[
\begin{array}{c}
a = a' : A \\
l = l' : \text{List}(A)
\end{array}
\]

\[
\text{cons}_A(a, l) = \text{cons}_A(a', l') : \text{List}(A)
\]

- Example: Equality version of the elimination rule for \((+), N\):

\[
\begin{array}{c}
n = n' : N \\
m = m' : N
\end{array}
\]

\[
\frac{n + m = n' + m'}{N} \quad (\text{N-El}^=)
\]

- The equality versions of the rules in questions can be formed in a **straight-forward way**, once one knows the non-equality version.
- We will often not mention them.
- In Agda they are **implicit** (part of the reduction machinery).
(b) Nondep. Funct. Type and Product

We introduce in the following non-dependent versions of the product and the function type.

The Non-Dependent Product

- **Formation Rule**
  \[
  A : \text{Set} \quad B : \text{Set} \quad A \times B : \text{Set} \quad (\times - F)
  \]

- **Introduction Rule**
  \[
  a : A \quad b : B \quad \langle a, b \rangle : A \times B \quad (\times - I)
  \]

- **Elimination Rules**
  \[
  c : A \times B \quad \pi_0(c) : A \quad c : A \times B \quad \pi_1(c) : B \quad (\times - \text{El}_0)
  \]
  \[
  \langle a, b \rangle = \langle a', b' \rangle : A \times B \quad (\times - \text{El}_1)
  \]

- **Equality Rules**
  \[
  a : A \quad b : B \quad \pi_0(\langle a, b \rangle) = a : A \quad (\times - \text{Eq}_0)
  \]
  \[
  \pi_1(\langle a, b \rangle) = b : B \quad (\times - \text{Eq}_1)
  \]

The \(\eta\)-Rule

The \(\eta\)-rule does not fit into the above schema:

\[
\frac{c : A \times B}{c = \langle \pi_0(c), \pi_1(c) \rangle : A \times B} \quad (\times - \eta)
\]

Equality Versions of the \(\times\)-Rules

- **Equality Version of the Formation Rule**
  \[
  A = A' : \text{Set} \quad B = B' : \text{Set} \quad A \times B = A' \times B' : \text{Set} \quad (\times - \text{F} =)
  \]

- **Equality Version of the Introduction Rule**
  \[
  a = a' : A \quad b = b' : B \quad \langle a, b \rangle = \langle a', b' \rangle : A \times B \quad (\times - \text{I} =)
  \]

- **Equality Versions of the Elimination Rules**
  \[
  c = c' : A \times B \quad \pi_0(c) = \pi_0(c') : A \quad (\times - \text{El}_0 =)
  \]
  \[
  \pi_1(c) = \pi_1(c') : B \quad (\times - \text{El}_1 =)
  \]
**The Product in Agda**

- In Agda, the product is represented as a record type.
- Assume we have introduced $A, B :: \text{Set}$.
- Then we can introduce the record type

\[
D :: \text{Set} = \text{sig}\{a :: A; b :: B\}
\]

- One can introduce longer record types as well, e.g.

\[
\text{sig}\{a :: A; b :: B; c :: C; e :: E\}
\]

**The Product in Agda**

\[
D = \text{sig}\{a :: A; b :: B\}
\]

- Elements of a record type are introduced as follows:
  - Assume we have $a' :: A, b' :: B$.
  - Then we can introduce in the above situation

\[
c :: D = \text{struct}\{a = a'; b = b'\} :: D
\]

- Unfortunately, the built-in product (record type) in Agda does not behave very well.
  - This is due to the fact that Agda doesn’t support the $\eta$-rule.
  - In this setting $\eta$-equality would assert that if

\[
c :: \text{sig}\{a :: A; b :: B\}
\]

then

\[
c = \text{struct}\{a = c.a; b = c.b\}
\]

- In most cases one can avoid this, by using the inductively defined $\Sigma$-type, which will be treated later.
Problem of the $\eta$-Rule

The exact problem of the missing $\eta$-rule is as follows:

- Assume we have $s : \text{sig}\{a :: N; b :: N\}$.
- If we then make case distinction on $s.a : N$, we know in case of $s.a = Z$ not that $s = \text{struct}\{a = Z; b = s.b\}$.
- This is since we don’t know without the $\eta$-rule, whether $s = \text{struct}\{a = s.a; b = s.b\}$.
- If we had the $\eta$-rule we would get $s = \text{struct}\{a = s.a; b = s.b\} = \text{struct}\{a = Z; b = s.b\}$.
- If we use instead of $\text{sig}$ the inductively defined product, we can make case distinction on an element, e.g. $s = p\ a\ b$, and if we then make case distinction on $a$, we get in case $a = Z$, that $s = p\ Z\ b$.

Expensiveness of the $\eta$-Rule

The reason for not having the $\eta$-rule is that it is computationally expensive.

- Whether we can expand $s$ to $\text{struct}\{a = s.a; b = s.b\}$ depends on, whether $s : \text{sig}\{a :: A; b :: B\}$.
- All other reductions can be applied to terms without knowing the exact type of the term – in case of $\eta$, we need to know this type.
- So, in order to allow $\eta$-rule as part of the reduction system, we need to carry along with each term (and as well with each subterm) its type, which requires an enormous additional overhead.

Indentation Sensitivity

For $\text{sig}$, $\text{struct}$ and some other constructs to be introduced later (let, case, packages, probably some more) there are two versions for determining the scope of the construct:

- Using “{” and “}”.
  - The scope is what is enclosed in those brackets.
  - The selections must be separated by “;”.
  - In case of let (see below), “{” and “}” can be omitted.
  - For instance $\text{struct}\{a = a0; b = b0\} :: \text{sig}\{a :: A; b :: B\}$

Using indentation.

- The scope are the following lines which are indented more than the keyword “struct”, “sig” etc.
- The first characters in each line must be indented in the same way as the first line following the “struct”, “sig” etc.
- What forms the definition, must be more indented than the rest.
- Example:

$$\begin{align*}
\text{sig} \\
  a :: A \\
  b :: B
\end{align*}$$

is the indentation sensitive version of $\text{sig}\{a :: A; b :: B\}$.

Both versions are equivalent.
### Indentation Sensitivity

- Internally, first the indentation sensitive version is translated to a version using curly brackets and then it is type checked.
- Error messages refer to the version using curly brackets.
- One has to interpret such error messages as if one had actually written a version using curly brackets, even if one is using indentation sensitivity.

---

### Intro

- Usually, we don’t have to spell out

\[
\text{struct}\{a = a'; b = b'\}
\]

in full.
- Assume we have a goal, the type of which is a sig-type, e.g.

\[
\begin{align*}
AB & :: \text{Set} = \text{sig}\{a :: A; b :: B\} \\
ab & :: AB = \{! !\}
\end{align*}
\]

---

### Let expressions in Agda

- When introducing products, let expressions are often useful. They allow to introduce temporary variables.
- Let-expressions have the form

\[
\begin{align*}
\text{let } a_1 & :: A_1 \\
& = s_1 \\
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
\;
Let expressions in Agda (Cont.)

- This means that we introduce new local constants $a_1 :: A_1 = s_1$,
  $a_2 :: A_2 = s_2$,
  $\ldots$,
  $a_n :: A_n = s_n$,
which can now be used locally.

- $s_i$ can refer to all $a_j$ defined before, including $a_i$ itself, i.e. it can refer to $a_0, \ldots, a_i$.

- Reference to $a_i$ might result in non-termination; termination will be discussed below.

Simple Example

The following function computes $(n + n) \ast (n + n)$ for $n :: N$:

$$f (n :: N) :: N$$

$$= \text{let } m :: N = n + n \text{ in } m \ast m$$

See exampleLetExpression.agda

Note that this version is computationally cheaper than the function computing directly $(n + n) \ast (n + n)$:

- Using let, $n + n$ is computed only once,
- without let, we have to compute it twice.

Example Using the Product

As an example we want to define in Agda, depending on functions $a_\_c : A \rightarrow C$, $b_\_d : B \rightarrow D$ and an element $ab : A \times B$ and element $f(ab, a_\_c, b_\_d) : C \times D$.

- This means that $f$ is a function which takes arguments $a_\_c$, $b_\_d$ and $ab$ as above and returns an element of $C \times D$. 

Let expressions in Agda (Cont.)

- If we are in a goal, we can use the goal menu Make let expression.
- We have to insert into the goal the variables, separated by blanks, e.g. “$a_1 \ a_2 \ \cdots \ a_n$”.
- Agda will construct a template of the form:

$$\text{let } a_1 :: \{! !\} = \{! !\}$$

$$a_2 :: \{! !\} = \{! !\}$$

$$\ldots$$

$$a_n :: \{! !\} = \{! !\}$$

$$\text{in } \{! !\}$$
Example Using the Product

So the type of $f$ in Agda is as follows:

$$AB :: \text{Set} = \text{sig}\{a :: A; b :: B\}$$

$$CD :: \text{Set} = \text{sig}\{c :: C; d :: D\}$$

$$f \quad (a\_c :: A \rightarrow C)$$

$$\quad (b\_d :: B \rightarrow D)$$

$$\quad (ab :: AB)$$

$$:: CD$$

$$= \{! !\}$$

Example Using the Product

The idea for this function is as follows:

- We first project $ab$ to elements $a : A$, $b : B$.
- Then we apply $a\_c$ to $a$ and obtain an element $c : C$, and $b\_d$ to $b$ and obtain an element $d : D$.
- Finally we form the pair $(b, d)$.

Agda Code for the Above

$$AB :: \text{Set} = \text{sig}\{a :: A; b :: B\}$$

$$CD :: \text{Set} = \text{sig}\{c :: C; d :: D\}$$

$$f \quad (a\_c :: A \rightarrow C)$$

$$\quad (b\_d :: B \rightarrow D)$$

$$\quad (ab :: AB)$$

$$:: CD$$

$$= \text{let} a :: A = ab.a$$

$$\quad b :: B = ab.b$$

$$\quad c' :: C = a\_c a$$

$$\quad d' :: D = b\_d b$$

$$\text{in struct}\{c = c'; d = d'\}$$

See exampleLetExpressionSig.agda.
Open Expression

Open abbreviates the unpacking of pairs.

Assume

\[ e :: \text{sig}\{a_1 :: A_1; a_2 :: A_2; \ldots\} \]

Then

\[ \text{open } e \text{ use } x_1 :: A_1 = a_1, \]

\[ x_2 :: A_2 = a_2, \]

\[ \ldots \]

\[ \text{in } \ldots \]

is equivalent to

\[ \text{let } x_1 :: A_1 = e.a_1; \]

\[ x_2 :: A_2 = e.a_2; \]

\[ \ldots \]

\[ \text{in } \ldots \]

So the above example reads

\[
\begin{align*}
f(a_c :: A \rightarrow C) \\
(b_d :: B \rightarrow D) \\
(ab :: AB) \\
:: CD
\end{align*}
\]

= open ab

use a :: A = a,

b :: B = b

in let c' :: C = a_c a

d' :: D = b_d b

in struct\{c = c'; d = d'\}

See exampleOpenExpression1.agda.

Open Expression

If we want to use the selectors directly, i.e. assuming

\[ e :: \text{sig}\{a_1 :: A_1; a_2 :: A_2; \ldots\} \]

to use the names \(a_1, a_2, \ldots\), we can write more briefly

\[ \text{open } e \text{ use } a_1, a_2, \ldots \]

\[ \text{in } \ldots \]

So the above example reads more briefly

\[
\begin{align*}
f(a_c :: A \rightarrow C) \\
(b_d :: B \rightarrow D) \\
(ab :: AB) \\
:: CD
\end{align*}
\]

= open ab use a, b

in let c' :: C = a_c a

d' :: D = b_d b

in struct\{c = c'; d = d'\}

See exampleOpenExpression2.agda.
**Termination Checker**

- The type checker in Agda allows recursive definitions. For instance, the following passes the type checker:

  \[ a :: A = a \]

- Necessary, since for instance the definition of + is necessarily recursive, i.e. will make use of +:

  \[
  (+) \ (n, m :: N) \\
  :: N \\
  = \text{case } m \text{ of} \\
  \quad Z \rightarrow n \\
  \quad S \ m' \rightarrow n + m'
  \]

**Termination Check (Cont.)**

- In order to avoid mistakes, Agda has a builtin termination checker: Menu **Check Termination**.

- One should **use this command at the end of a session**, in order to avoid **black hole recursion**.

- **If the termination check succeeds**, all programs checked will **terminate**.

- **If the termination check fails**, it might still be the case that all programs **terminate**. (One cannot write a universal termination checker, since the Turing halting problem is undecidable).

**Termination Check**

- The termination check can avoid non-obvious mistakes.

- Assume for instance we have defined \( a :: A \) and \( b :: B \), and \( AB = \text{sig}\{a :: A; b :: B\} \).

- If we define

  \[ ab :: AB = \text{struct}\{a = a; b = b\} \]

  we have

  - **not defined an element** \( ab \) s.t. \( ab.a = a \) and \( ab.b = b \),
  - **but defined an element** \( ab \) s.t. \( ab.a = ab.a \) and \( ab.b = ab.b \).

  That is a black-hole recursion, and will not pass the termination checker.

**Termination Check**

- Especially, when later proving theorems, we don’t know whether what we have proved is true unless we have used the termination checker.

  - Any formula \( A \) (which is just a type), whether it is true or false, can be shown using black-hole recursion by using the following Agda code:

    \[ a :: A = a \]

- Only if a theorem has passed the termination checker, we know it holds.
The Non-Dependent Function Type

**Formation Rule**

\[
A : \text{Set} \quad B : \text{Set} \quad (\rightarrow -F)
\]

\[
A \rightarrow B : \text{Set}
\]

**Introduction Rule**

\[
x : A \Rightarrow b : B \\
(\lambda x : A.b) : A \rightarrow B
\]

\[
(\rightarrow -I)
\]

**Elimination Rule**

\[
f : A \rightarrow B \\
a : A
\]

\[
f a : B
\]

\[
(\rightarrow -El)
\]

**Equality Rule**

\[
x : A \Rightarrow b : B \\
a : A
\]

\[
(\lambda x : A.b) a = b[x := a] : B
\]

\[
(\rightarrow -Eq)
\]

---

\**\(\alpha\)**-Equivalence

- As for the simply typed \(\lambda\)-calculus, terms which differ in the choice of bound variables are identified:
  - E.g. \(\lambda x : A.x\) and \(\lambda y : A.y\) are identified.
  - E.g. \(\lambda x : N.x + x\) and \(\lambda y : N.y + y\) are identified.
  - A similar rule applies to bound variables in types (see later).

---

\**\(\beta\)**-Reduction

- \(b[x := a]\) was as for the simply typed \(\lambda\)-calculus the result of substituting in \(b\) every occurrence of variable \(x\) by the term \(a\) (after renaming of bound variables as usual).

- The equality rule is a symmetric version of \(\beta\)-reduction

\[
(\lambda x : A.b) a \rightarrow b[x := a]
\]

---

\**\(\eta\)**-Rule

Again the \(\eta\)-rule does not fit into the above schema:

\[
f : A \rightarrow B
\]

\[
f = (\lambda x : A.f x) : A \rightarrow B
\]

\[
(\rightarrow -\eta)
\]
Equality Versions of the \(\rightarrow\)-Rules

**Equality Version of the Formation Rule**

\[
A = A' : \text{Set} \quad B = B' : \text{Set} \quad \quad \quad (\rightarrow - F=)
\]

**Equality Version of the Introduction Rule**

\[
x : A \Rightarrow b = b' : B \\ \lambda x : A.b = \lambda x : A.b' : A \rightarrow B \quad \quad \quad (\rightarrow - I=)
\]

**Equality Version of the Elimination Rule**

\[
f = f' : A \rightarrow B \quad a = a' : A \\ f \ a = f' \ a' : B \quad \quad \quad (\rightarrow - \text{El}=)
\]

### The Function Type in Agda

- In Agda one writes \(A \rightarrow C\) for the nondependent function type. We write on our slides \(\rightarrow\) instead of \(\rightarrow\).
- There are two ways of introducing an element of \(A \rightarrow C\):
  - We can write
    \[
    f \quad (x :: A) \\
    :: C \\
    = \cdots
    \]
    Requires the \(\cdots\) is an element of type \(C\), possibly making use of \(x\).
  - The above introduces
    \[
    f :: A \rightarrow C
    \]

### The Function Type in Agda (Cont.)

- Alternatively, one can use the \(\lambda\)-notation:
  - Remember that \(\\backslash\) is used instead of \(\lambda\) in Agda.
  - In our slides we will use \(\lambda\).
  - The above can be rewritten as
    \[
    f :: A \rightarrow C \\
    = \lambda(x :: A) \rightarrow \cdots
    \]

### Equivalence of the two Notations

- Both ways of introducing functions are equivalent.
- One can check this by defining two versions:
  \[
  \text{postulate} \quad A :: \text{Set} \quad \\
  f :: A \rightarrow A \rightarrow A \\
  = \lambda(a, b :: A) \rightarrow a \quad \\
  g (a, b :: A) :: A \\
  = a
  \]
- See exampleEquivalenceLambdaNotations.agda
Equivalence of the two Notations

- If one then introduces an arbitrary goal, e.g.

\[
\text{Test} :: A \rightarrow A \rightarrow A
= \{! !\}
\]

inserts into it the value \( f \), uses goal-menu

Compute WHNF, (for compute weak head normal form), and does the same with \( g \), one obtains twice the same result, namely

\[
\lambda(a, b :: A) \rightarrow a
\]

- This is a general method for evaluating terms (e.g. for evaluating the result of a applying a function to some arguments).

\[\text{Interactive Theorem Proving, CS\_336, Lentterm 2004, Sec. 2(b)}\]

\[295b\]

\[\lambda\text{-Notation in Agda}\]

- In most cases, it is easier to use the first way of introducing \( \lambda \)-terms.

- However, \( \lambda \)-notation allows to introduce anonymous functions (i.e. functions without giving them names):

A typical example from functional programming is the map function, which applies a function to each element of a list:

\[
\text{map } (\lambda(x::N) \rightarrow \text{S } x)
(\text{cons two } (\text{cons three } \text{nil}))
\]

The result would be

cons three (cons four nil)

See exampleMapAppliedToList.agda.

Abbreviations

- We can write

\[
\lambda(n,m::N) \rightarrow \cdots
\]

instead of

\[
\lambda(n::N) \rightarrow \lambda(m::N) \rightarrow \cdots
\]

Abbreviations (Cont.)

- Similarly we can write

\[
f \ n,m::N)
\vdash N
= \cdots
\]

instead of

\[
f \ n::N)
(m::N)
\vdash N
= \cdots
\]

See exampleMapAppliedToList.agda.
The Function Type in Agda (Cont.)

- **Application** has the same syntax as in the rules of dependent type theory: If we have
  \[ f :: A \to B , \]
  \[ a :: A , \]
  then we can conclude \( f \ a :: B \).
- And we have that
  \[ (\lambda(x :: A) \to b) \ a \]
  and
  \[ b[x := a] \]
  are identified.

\(\eta\)-Rule in Agda (Cont.)

- In Agda syntax, the \(\eta\)-rule would state that if
  \[ f :: A \to B \]
  then
  \[ f = \lambda(x :: A) \to f \ x . \]
- \(\eta\)-rule is **computationally expensive** and therefore **not implemented**.
  (This is similar to what was said about the \(\eta\)-rule for the product).
- The lack of the \(\eta\)-rule causes sometimes problems.

Intro

- When introducing \(\lambda\)-terms, there is no need to introduce the \(\lambda\)-term by hand.
- Assume for instance the following goal:
  \[ f :: A \to A \]
  \[ = \{! !\} \]
- If the position is inside the goal, we can use the goal-menu **Intro**, in order to obtain a template for the \(\lambda\)-term:
  \[ f :: A \to A \]
  \[ = \lambda(h :: \{! !\}) \to \{! !\} \]

Solve

- We usually don’t have to introduce the types of the \(\lambda\)-variables, they can usually be solved automatically using Agda-menu **Solve**.
- Applied in the above situation we obtain:
  \[ f :: A \to A \]
  \[ = \lambda(h :: A) \to \{! !\} \]
- It is recommended to rename the automatically generated variable \(h\) to something more meaningful:
  \[ f :: A \to A \]
  \[ = \lambda(a :: A) \to \{! !\} \]
Solve

In some cases, the types inferred can be simplified. For instance if we start with

\[ g :: ((A \to A) \to A) \to A \]
\[ = \lambda (h :: \{! !\}) \to \{! !\} \]

we obtain using solve

\[ g :: ((A \to A) \to A) \to A \]
\[ = \lambda (h :: (h :: (h :: A) \to A) \to A) \to \{! !\} \]

which can be simplified to

\[ g :: ((A \to A) \to A) \to A \]
\[ = \lambda (a_a_a :: (A \to A) \to A) \to \{! !\} \]

Common Contexts

The convention is that all rules can as well be weakened by a common context.

This means that when introducing a rule

\[ \Gamma_1 \Rightarrow \theta_1 \ldots \Gamma_n \Rightarrow \theta_n \]

we implicitly introduce as well the following rules

\[ \Delta, \Gamma_1 \Rightarrow \theta_1 \ldots \Delta, \Gamma_n \Rightarrow \theta_n \]

This convention will not apply to the context rules (Context_0) and (Context_1) (see later).

Example

For instance, the formation rule of \( \times \):

\[ A : \text{Set} \quad B : \text{Set} \]
\[ A \times B : \text{Set} \]

\((\times \text{-}F)\)

can be weakened as follows:

\[ \Gamma \Rightarrow A : \text{Set} \quad \Gamma \Rightarrow B : \text{Set} \]
\[ \Gamma \Rightarrow A \times B : \text{Set} \]

\((\times \text{-}F)\)
Example (Cont.)

Consider the sample derivation (assuming $A : Set$):

$$
\begin{align*}
  & x : A, y : A \Rightarrow y : A \\
  \quad & x : A \Rightarrow (\lambda y : A.y) : A \Rightarrow A \\
  \quad & (\lambda x : A.\lambda y : A.y) : A \Rightarrow A \Rightarrow A
\end{align*}
$$

The first rule used is the rule for $\lambda$-introduction, weakened by the context $x : A$.

The second rule used is the rule for $\lambda$-introduction without any weakening.

Weakening of Axioms

If we have an axiom $\theta$

for any judgement $\theta$

- e.g. $\theta = N : Set$ or $\theta = 0 : N$

and we want to weaken it by context $\Gamma$, we need to make sure that $\Gamma \Rightarrow \text{Context}$ holds.

So we need in the weakened form one additional premise:

$$
\begin{align*}
  & \Gamma \Rightarrow \text{Context} \\
  \quad & \Gamma \Rightarrow \theta
\end{align*}
$$

Example

The formation rule for $N$

$$
N : Set
$$

will be weakened as follows:

$$
\begin{align*}
  & \Gamma \Rightarrow \text{Context} \\
  \quad & \Gamma \Rightarrow N : Set
\end{align*}
$$

(c) Structural Rules
Context Rules

The empty context

\[ \emptyset \Rightarrow \text{Context} \quad (\text{Context}_0) \]

Extending a context

\[
\frac{\Gamma \Rightarrow A : \text{Set}}{\Gamma, x : A \Rightarrow \text{Context}} \quad (\text{Context}_1)
\]

(Where in the last rule \( x \) must not occur in \( \Gamma \)).

The convention that rules can be weakened by a common context does not apply to the rules \( (\text{Context}_0) \) and \( (\text{Context}_1) \).

Example Derivation (Context Rules)

- The following derives \( x : N, y : N, z : N \Rightarrow \text{Context} \)
  (Note that \( N : \text{Set} \) is the same as \( \emptyset \Rightarrow N : \text{Set} \)):

\[
\frac{N : \text{Set}}{x : N \Rightarrow \text{Context}} \quad (\text{Context}_1)
\]
\[
\frac{x : N \Rightarrow N : \text{Set}}{x : N, y : N \Rightarrow \text{Context}} \quad (\text{N-F})
\]
\[
\frac{x : N, y : N \Rightarrow N : \text{Set}}{x : N, y : N, z : N \Rightarrow \text{Context}} \quad (\text{Context}_1)
\]

Example Derivation (Context Rules)

- We assume the following formation rule for the type of natural numbers:

\[ N : \text{Set} \quad (\text{N-F}) \]

With this rule, following the convention on slide ??, we have as well introduced the rules

\[
\frac{\Gamma \Rightarrow \text{Context}}{\Gamma \Rightarrow N : \text{Set}} \quad (\text{N-F})
\]

Assumption Rule

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow \text{Context}}{\Gamma, x : A, \Gamma' \Rightarrow x : A} \quad \text{(Ass)}
\]
Example Deriv. (Assumpt. Rule)

- We extend the derivation of slide ?? to a derivation of
  \[ x : N, y : N, z : N \Rightarrow y : N : \]
  \[
  \begin{array}{c}
  x : N, y : N, z : N \Rightarrow \text{Context} \\
  \text{(Ass)}
  \end{array}
  \]
  \[ x : N, y : N, z : N \Rightarrow y : N \]
- Similarly we can derive \( x : N, y : N, z : N \Rightarrow z : N : \)
  \[
  \begin{array}{c}
  x : N, y : N, z : N \Rightarrow \text{Context} \\
  \text{(Ass)}
  \end{array}
  \]
  \[ x : N, y : N, z : N \Rightarrow z : N \]

Assumption Rule in Agda

- When we define a function:
  \[
  f \ (a :: A) :: B
  \]
  \[ = \{! !\} \]
  we can make use of \( a :: A \) when solving the goal \( \{! !\} \).
- This is an application of the assumption rule:
  When solving \( \{! !\} \) we essentially define
  \[ a :: A \text{ an element } \{! !\} :: B. \]

Example Deriv. (Assumpt. Rule) (Cont.)

- The full derivation of first judgement on the previous slide is as follows:
  \[
  \begin{array}{c}
  N : \text{Set} \\
  \text{(Context}_1\text{)}
  \end{array}
  \]
  \[
  \begin{array}{c}
  x : N \Rightarrow \text{Context} \\
  \text{(N-F)}
  \end{array}
  \]
  \[ x : N, y : N \Rightarrow N : \text{Set} : \text{(Context}_1\text{)} \]
  \[
  \begin{array}{c}
  x : N, y : N \Rightarrow \text{Context} \\
  \text{(N-F)}
  \end{array}
  \]
  \[ x : N, y : N, z : N \Rightarrow N : \text{Set} : \text{(Context}_1\text{)} \]
  \[
  \begin{array}{c}
  x : N, y : N, z : N \Rightarrow \text{Context} \\
  \text{(Ass)}
  \end{array}
  \]
  \[ x : N, y : N, z : N \Rightarrow y : N \]

Assumption Rule in Agda (Cont.)

- The above corresponds to a derivation
  \[
  a : A \Rightarrow \{! !\} : B
  \]
  \[ (\lambda a : A.\{! !\}) : A \rightarrow B \] \( \rightarrow \text{-I} \)
- If \( B \) is equal to \( A \) we can use the assumption rule directly
  \[
  a : A \Rightarrow \lambda a : A.a : A \rightarrow \text{A} \]
  \[ (\rightarrow \text{-I}) \]
  in order to solve this goal.
Assumption Rule in Agda (Cont.)

More generally we might in the derivation of $a : A \Rightarrow \{ ! ! \} : B$ make anywhere use of $a : A$, as long as this is in the context.

\[
\begin{align*}
\vdots \\
\frac{a : A \Rightarrow a : A}{(\text{Ass})} \\
\vdots \\
\frac{a : A \Rightarrow s : B}{(\lambda a : A.s) : A \rightarrow B} & (\rightarrow -I)
\end{align*}
\]

Weakening Rule

\[
\frac{\Gamma, \Gamma' \Rightarrow \theta}{\Gamma, \Delta, \Gamma' \Rightarrow \text{Context}} \quad \text{(Weak)}
\]

$\theta$ stands for an arbitrary non-dependent judgement.

This rule allows to add an additional context piece ($\Delta$) to the context of a judgement.

The judgement $\Gamma, \Gamma' \Rightarrow \theta$ is weakened by $\Delta$.

Assumption Rule in Agda (Cont.)

Similarly, when solving the goal

\[f :: A \rightarrow B\]

\[= \lambda(a :: A) \rightarrow \{ ! ! \}\]

in $\{ ! ! \}$ we can make use of $a :: A$.

In fact when solving the above, we implicitly use the rule

\[
\frac{a : A \Rightarrow \{ ! ! \} : B}{(\lambda a : A.\{ ! ! \}) : A \rightarrow B} & (\rightarrow -I)
\]

So we have to solve $a : A \Rightarrow \{ ! ! \} : B$ in order to derive

\[(\lambda a : A.\{ ! ! \}) : A \rightarrow B\]

Weakening Rule (Cont.)

Remark: One can in fact show that the weakening rule can be weakly derived.

Weakly derived means: whenever the assumptions of the rule can be derived in the complete set of rules we provide, then as well the conclusion.

However, this can’t be derived from the premise the conclusion directly.

An exception is when we additionally assume some judgements for instance $A : \text{Set}$ (corresponding to “postulate” in Agda).

Then $\Gamma \Rightarrow A : \text{Set}$ doesn’t follow without the weakening rule.
Example Deriv. (Weak. Rule)

We derive \( a : A, b : B \Rightarrow a : A \) from global assumptions

\[
\begin{array}{c}
A : \text{Set}, B : \text{Set}:
\end{array}
\]

\[
\begin{array}{c}
A : \text{Set} \quad (\text{Context}_1) \\
a : A \Rightarrow \text{Context} \quad (\text{Ass}) \\
a : A \Rightarrow a : A
\end{array}
\]

\[
\begin{array}{c}
B : \text{Set} \quad (\text{Context}_1) \\
a : A \Rightarrow B : \text{Set} \quad (\text{Weak}) \\
a : A, b : B \Rightarrow \text{Context} \quad (\text{Weak})
\end{array}
\]

\[
\begin{array}{c}
a : A, b : B \Rightarrow a : A
\end{array}
\]

Example Deriv.2 (Weak. Rule)

We derive \( x : A \rightarrow (B \times C), a : A \Rightarrow x : A \rightarrow (B \times C) \)

from global assumptions \( A : \text{Set}, B : \text{Set}, C : \text{Set} : \)

\[
\begin{array}{c}
B : \text{Set} \quad (\times -F) \\
A : \text{Set} \quad B \times C : \text{Set} \quad (\rightarrow -F) \\
A \rightarrow (B \times C) : \text{Set} \quad (\text{Context}_1) \\
x : A \rightarrow (B \times C) \Rightarrow \text{Context} \quad (\text{Weak})
\end{array}
\]

\[
\begin{array}{c}
x : A \rightarrow (B \times C) \Rightarrow A : \text{Set} \quad (\text{Context}_1) \\
x : A \rightarrow (B \times C), a : A \Rightarrow \text{Context} \quad (\text{Ass})
\end{array}
\]

General Equality Rules

**Reflexivity**

\[
\begin{array}{c}
A : \text{Set} \quad (\text{Refl}_\text{Set}) \\
a : A \\
a = a : A \quad (\text{Refl}_\text{Elem})
\end{array}
\]

(Reflexivity can be weakly derived, except for global assumptions).

**Symmetry**

\[
\begin{array}{c}
A = B : \text{Set} \quad (\text{Sym}_\text{Set}) \\
B = A : \text{Set} \\
a = b : A \quad (\text{Sym}_\text{Elem}) \\
b = a : A
\end{array}
\]

General Equality Rules (Cont.)

**Transitivity**

\[
\begin{array}{c}
A = B : \text{Set} \quad B = C : \text{Set} \quad (\text{Trans}_\text{Set}) \\
A = C : \text{Set} \\
a = b : A \quad b = c : A \quad (\text{Trans}_\text{Elem})
\end{array}
\]

**Transfer**

\[
\begin{array}{c}
a : A \\
a = B : \text{Set} \quad (\text{Transfer}_\text{Set}) \\
a : B \\
a = b : A \\
a = b : B \quad (\text{Transfer}_\text{Elem})
\end{array}
\]
Example Deriv. (Gen. Equal. Rules)

In the previous derivation, the most complicated step was:

\[
\begin{align*}
    & \quad \text{N:Set} \quad \text{(Context)} \\
    & y : N \Rightarrow Context \\
    & y : N \Rightarrow N : \text{Set} \\
    & y : N, x : N \Rightarrow Context \quad \text{(Ass)} \\
    & y : N, x : N \Rightarrow x : N \\
    & y : N \Rightarrow (\lambda x : N. x) \ y = y : N \\
    & y : N \Rightarrow y = (\lambda x : N. x) \ y : N \\
    & (\lambda y : N. y + 0) = (\lambda y : N. (\lambda x : N. x) \ y) : N \Rightarrow N \quad (-1=) \quad \text{(TransElem)}
\end{align*}
\]

This is an example of the equality rule for the non-dependent function type (slide ??):

\[
\begin{align*}
    x : A \Rightarrow b : B & \quad a : A \\
    (\lambda x : A. b) \ a = b[x := a] : B \quad (\rightarrow \text{Eq})
\end{align*}
\]

with \( A := B := N \), \( b := x \), \( a := y \).

Therefore \( b[x := a] = y \).

This instance of the rule was weakened by an additional context \( y : N \).

Example Deriv. (Gen. Equal. Rules)

Note that from the premises of that rule

\[
\begin{align*}
    & y : N, x : N \Rightarrow x : N \quad y : N \Rightarrow y : N \\
    & y : N \Rightarrow (\lambda x : N. x) \ y = y : N \\
\end{align*}
\]

we can derive using the introduction and elimination rule

\[
\begin{align*}
    & y : N \Rightarrow (\lambda x : N. x) \ y : N \\
\end{align*}
\]

as follows:

\[
\begin{align*}
    & y : N, x : N \Rightarrow x : N \\
    & y : N \Rightarrow (\lambda x : N. x) \ N \Rightarrow N \\
    & y : N \Rightarrow (\lambda x : N. x) \ y : N \\
\end{align*}
\]

Example Deriv. (Gen. Equal. Rules)

The equality rule expresses how the function \( \lambda x : N. x \) applied to \( y \) is evaluated as follows:

- We evaluate the body of the function \((x)\) by setting for \( x \) the argument of the function \((y)\).
- This is the same as substituting in the body for \( x \) the argument of the function, i.e. \( y \).
- This explains how the detour above of first introducing and then eliminating an expression can be reduced (namely to \( y \) or in general to \( b[x := a] \)).
Substitution Rules

The following rules can be weakly derived:

**Substitution 1**

\[ \frac{\Gamma, x : A, \Gamma' \Rightarrow \theta}{\Gamma, \Gamma'[x := a] \Rightarrow \theta[x := a]} \quad (\text{Subst}_1) \]

(\(\Gamma'[x := a]\) is the result of substituting in \(\Gamma'\) all occurrences of \(x\) by \(a\)).

**Substitution 2**

\[ \frac{\Gamma, x : A, \Gamma' \Rightarrow B : \text{Set} \quad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow B[x := a] = B[x := a'] : \text{Set}} \quad (\text{Subst}_2) \]

**Substitution 3**

\[ \frac{\Gamma, x : A, \Gamma' \Rightarrow b : B \quad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow b[x := a] = b[x := a'] : B[x := a]} \quad (\text{Subst}_3) \]

Example Deriv. (Substitution)

\[ \frac{x : N, y : N \Rightarrow x + y : N}{(\text{Ass})} \]

\[ \frac{y : N \Rightarrow 0 + y : N}{(\text{N-1}_a)} \]

\[ \frac{(\lambda y : N. 0 + y) : N \Rightarrow N}{(-\text{I}_1)} \]

---

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 2(c)
The Dependent Product

The dependent product is similar as the non-dependent product (e.g. \( A \times B \)), but now the second set depends on an element of the first set.

The type theoretic notation is
\[ (a : A) \times B \]

Elements of \( (a : A) \times B \) are pairs
\[ \langle a', b' \rangle \]

s.t.
- \( a' : A \)
- \( b' : B[a := a'] \).

Example 1 (Dep. Products)

One example for its use are the set of sorted lists:

- Sorted\((l)\) is a predicate on NatList expressing that \( l \) is sorted.
- An element of
  \[ \text{SortedList} := (l : \text{NatList}) \times \text{Sorted}(l) \]
  is a pair
  \[ \langle l, p \rangle \]
  s.t.
  - \( l : \text{NatList} \),
  - \( p : \text{Sorted}(l) \), i.e. \( p \) is a proof that \( l \) is sorted.
- So elements of \( \text{SortedList} \) are lists \( l \) together with a proof that \( l \) is sorted.

Example 2 (Dep. Products)

Let \( G \) be the set of genders, informally written
\[ G = \{ \text{male, female} \} . \]

Let for \( g : G \) the set \( \text{Names}_g \) be the collection of names of that gender, e.g. informally written
- \( (\text{Names male}) = \{ \text{Tom, Jim} \} \),
- \( (\text{Names female}) = \{ \text{Jill, Sara} \} \).
Example 2 (Dep. Products)

- The **set of names with their gender** is the set of pairs \(\langle g, n \rangle\) s.t. \(g\) is a Gender and \(n : (\text{Names } g)\).
- This set is written as

\[
\text{AllNames} := (g : G) \times (\text{Names } g)
\]

Rules of the Dependent Product

**Formation Rule**

\[
\begin{array}{c}
A : \text{Set} \\
\vdash x : A \Rightarrow B : \text{Set}
\end{array}
\quad \frac{}{(x : A) \times B : \text{Set}} \quad (\times-I)
\]

**Introduction Rule**

\[
\begin{array}{c}
x : A \Rightarrow B : \text{Set} \\
a : A \\
b : B[x := a]
\end{array}
\quad \frac{}{(a, b) : (x : A) \times B} \quad (\times-I)
\]

Example

- Assuming we have defined the set of genders \(G : \text{Set}\) and the set of names \(g : G \Rightarrow (\text{Names } g) : \text{Set}\), we can introduce the set

\[
\text{AllNames} := (g : G) \times (\text{Names } g) : \text{Set}
\]

by using the formation rule:

\[
\begin{array}{c}
G : \text{Set} \\
g : G \Rightarrow (\text{Names } g) : \text{Set}
\end{array}
\quad \frac{}{(g : G) \times (\text{Names } g) : \text{Set}} \quad (\times-I)
\]

Extra Premise in the Introd. Rule

- In the last introduction rule, an **extra premise**

\[
x : A \Rightarrow B : \text{Set}
\]

was required.

- This is required in order to guarantee that we can **form the type** \((x : A) \times B\).

- In case of the non-dependent product, this premise was not necessary:

\[
a : A \text{ and } b : B \text{ indirectly implies that we know } A : \text{Set}
\]

and \(B : \text{Set}\) from which it follows \(A \times B : \text{Set}\).
Example

Furthermore we can introduce

\[ (\text{male, Tom}) : \text{AllNames} \]

as follows:

\[
\frac{g : G \Rightarrow (\text{Names } g) : \text{Set} \quad \text{male} : G \quad \text{Tom} : (\text{Names male})}{(\text{male, Tom}) : (g : G) \times (\text{Names } g)} \quad (\times-\text{El})
\]

Note that we need the premise

\[ g : G \Rightarrow (\text{Names } g) : \text{Set} \]

Otherwise we only know that \((\text{Names male}) : \text{Set}, \) but not that \((\text{Names female}) : \text{Set}.\]

Example

Note that we \textbf{don’t have}

\[ (\text{female, Tom}) : \text{AllNames} \]

since we \textbf{don’t have}

\[ \text{Tom} : (\text{Names female}) \]

So here dependent types prevent errors. In an ordinary programming language without dependent types, we can’t define a corresponding type \text{AllNames} which allows at compile time to define

\[ (\text{male, Tom}) : \text{AllNames} \]

but not

\[ (\text{female, Tom}) : \text{AllNames} \]

Rules of the Dependent Product

Elimination Rules

\[
\frac{c : (x : A) \times B}{\pi_0(c) : A} \quad (\times-\text{El}_0) \\
\frac{c : (x : A) \times B}{\pi_1(c) : B[x := \pi_0(c)]} \quad (\times-\text{El}_1)
\]

Equality Rules

\[
\frac{x : A \Rightarrow B : \text{Set} \quad a : A \quad b : B[x := a]}{\pi_0((a, b)) = a : A} \quad (\times-\text{Eq}_0)
\]

\[
\frac{x : A \Rightarrow B : \text{Set} \quad a : A \quad b : B[x := a]}{\pi_1((a, b)) = b : B[x := a]} \quad (\times-\text{Eq}_1)
\]

Note that the last two rules require the extra premise \(x : A \Rightarrow B : \text{Set} \) (which is not implied by the premises).

Example

In the “Names-example we have that, if \(a : \text{AllNames},\) then \(\pi_0(a) : G\) and \(\pi_1(a) : (\text{Names } \pi_0(a)):\)

\[
\frac{a : (g : G) \times (\text{Names } g)}{\pi_0(a) : G} \quad (\times-\text{El}_0)
\]

\[
\frac{a : (g : G) \times (\text{Names } g)}{\pi_1(a) : (\text{Names } \pi_0(a))} \quad (\times-\text{El}_1)
\]
Rules of the Dependent Product

We have the following $\eta$-rule:

$$c : (x : A) \times B$$

$$c = \langle \pi_0(c), \pi_1(c) \rangle : (x : A) \times C \quad (\times - \eta)$$

- As before, the $\eta$-rule expresses that every element of $(x : A) \times B$ is of the form $(\text{something}_0, \text{something}_1)$.
- The $\eta$-rule cannot be derived, if the element in question is a variable.

Equality Versions of the above

Equality Version of the Formation Rule

$$A = A' : \text{Set} \quad x : A \Rightarrow B = B' : \text{Set} \quad (\times - \text{I}^=)$$

Equality Version of the Introduction Rule

$$x : A \Rightarrow B : \text{Set} \quad a = a' : A \quad b = b' : B[x := a] \quad \langle a, b \rangle = \langle a', b' \rangle : (x : A) \times B \quad (\times - \text{I}^=)$$

Equality Versions of the Elimination Rules

$$c = c' : (x : A) \times B$$

$$\pi_0(c) = \pi_0(c') : A \quad (\times - \text{El}_0^=)$$

$$\pi_1(c) = \pi_1(c') : B[x := \pi_0(c)] \quad (\times - \text{El}_1^=)$$

The Non-Dep. Product as an Abbrev.

- The non-dependent product $A \times B$ can now be seen as an abbreviation for $(x : A) \times B$ for some fresh variable $x$.

- Taking $A \times B$ as an abbreviation, we can see that the rules for the non-dependent product are special cases of the rules for the dependent product.

The Non-Dep. Product as an Abbrev.

- More precisely this can be seen as follows:
  - From $A : \text{Set}$ and $B : \text{Set}$ we can derive $x : A \Rightarrow B : \text{Set}$ using the weakening rule.
  - Therefore the premises of the formation rule for the non-dependent product imply those of the formation rule for the non-dependent product.
  - From a derivation of $a : A$ we can derive $A : \text{Set}$ (we need the concept of presupposition for that, as introduced later).
  - Therefore the premises of the introduction rule for the non-dependent product imply those of the dependent product.
  - Similarly for the elimination, equality and $\eta$-rule.
The Dependent Product in Agda

- In Agda, the record type allows already dependencies of later sets on previous ones:
- Assume $A :: \text{Set}$, and $B :: \text{Set}$, possibly depending on $a :: A$.
- Then we can form $\text{sig}\{a :: A; b :: B\}$.
- Elements of this type can be introduced in the same way as before, i.e. if $a' :: A$ and $b' :: B[a := a']$ then we can form
  $$\text{struct}\{a = a'; b = b'\} :: \text{sig}\{a :: A; b :: B\}.$$ 
- Note that $b' :: B[a := a']$, so the type of $b'$ depends on $a'$.
- Furthermore, if $c :: \text{sig}\{a :: A; b :: B\}$, then $c.a :: A$ and $c.b :: B[a := c.a]$.

The “Names”-Example in Agda

- Although we haven’t introduced yet a notation for algebraic data types, the following is readable for those familiar with Haskell:

  ```agda
data G = male | female

data maleNames = Tom | Jim

data femaleNames = Jill | Sara
```

The “Names”-Example in Agda

- Names
  $$(g :: G) :: \text{Set} = \text{case } g \text{ of }$$
  $$(\text{male}) \rightarrow \text{maleNames}$$
  $$(\text{female}) \rightarrow \text{femaleNames}$$

- AllNames
  $$:: \text{Set} = \text{sig}$$
  $$g :: G$$
  $$n :: \text{Names } g$$

- Note that in the above example we have
  $$\text{Names male} = \text{maleNames} = \text{data Tom | Jim}$$
  $$\text{Names female} = \text{femaleNames} = \text{data Jill | Sara}$$

- Further we have for instance
  $$\text{struct}\{g=\text{male}, n=\text{Tom}\} :: \text{AllNames}$$
  whereas we don’t have
  $$\text{struct}\{g=\text{male}, n=\text{Jill}\} :: \text{AllNames}$$

See exampleAllNames.agda.
The Dependent Function Set

In the presence of dependent types we have as well a dependent function set, where the type of the result depends on the argument of the function.

Notation: \((x : A) \rightarrow B\), for the set of functions \(f\) which map an element \(a : A\) to an element of \(B[x := a]\).

In set-theoretic notation this is:

\[
\{ f \mid \text{f function} \\
\land \text{dom}(f) = A \\
\land \forall a \in A. f(a) \in B[x := a] \}
\]

Example (Dep. Function Set)

Consider the “Names example” as above

\((G : \text{Set} \text{ set of genders, Names } g \text{ set of names for gender } g)\).

Define

select : \((g : G) \rightarrow (\text{Names } g)\)
select male = Tom
select female = Jill

select selects for every gender a name.
select male will be an element of
\((\text{Names male}) = (\text{Names } g)[g := \text{male}]\).

It wouldn’t make sense to say \((\text{select male}) : (\text{Names } g)\), without knowing what \(g\) is.

Example (Dep. Function Set)

Note that for instance we don’t have

\((\lambda g : G. \text{Tom}) : (g : G) \rightarrow (\text{Names } g)\)

since we don’t have

\(((\lambda g : G. \text{Tom}) \text{ female}) : (\text{Names female})\)

Rules of the Dep. Funct. Set

Formation Rule

\[
A : \text{Set} \quad x : A \Rightarrow B : \text{Set} \quad (\rightarrow -F)
\]

\((x : A) \rightarrow B : \text{Set}\)

Introduction Rule

\[
x : A \Rightarrow b : B \quad (\lambda x : A.b) : (x : A) \rightarrow B \quad (\rightarrow -I)
\]
Rules of the Dep. Funct. Set

**Elimination Rule**

\[
\frac{f : (x : A) \to B \quad a : A}{f \ a : B[x := a]} (\to \text{-El})
\]

**Equality Rule**

\[
\frac{x : A \Rightarrow b : B \quad a : A}{(\lambda x : A.b) \ a = b[x := a] : B[x := a]} (\to \text{-Eq})
\]

The \(\eta\)-Rule

The \(\eta\)-rule has a special status:

\[
\frac{f : (x : A) \to B}{f = (\lambda x : A.f \ x) : (x : A) \to B} (\to \text{-}\eta)
\]

- As before, the \(\eta\)-rule expresses that every element of 
  \((x : A) \to B\) is of the form \(\lambda x : A.\text{something}\).
- The \(\eta\)-rule cannot be derived, if the element in question is a variable.

Equality Versions of the above

**Equality Version of the Formation Rule**

\[
\frac{A = A' : \text{Set} \quad x : A \Rightarrow B = B' : \text{Set}}{x : A \Rightarrow B = (x : A') \Rightarrow B' : \text{Set}} (\to \text{-F=})
\]

**Equality Version of the Introduction Rule**

\[
\frac{x : A \Rightarrow b = b' : B}{(\lambda x : A.b) = (\lambda x : A.b') : (x : A) \Rightarrow B} (\to \text{-I=})
\]

**Equality Version of the Elimination Rule**

\[
\frac{f = f' : (x : A) \Rightarrow B \quad a = a' : A}{f \ a = f' \ a' : B[x := a]} (\to \text{-El=})
\]

Non-Dep. Funct. Set as an Abbrev.

- The **non-dependent function type**
  \(A \Rightarrow B\)
  can be regarded as an abbreviation for the
  **dependent function type**
  \((x : A) \Rightarrow B\),
  where \(B\) does not depend on \(x\).
- As for the product one can see that the rules for the
  non-dependent function set are special cases of the
  rules for the dependent function set.
We have seen that the non-dependent function type is written as $A \rightarrow B$ in Agda.

The notation for the dependent function set is $(x :: A) \rightarrow C$.

Internally, Agda has only the dependent function set. That’s why one often sees in code generated by Agda (e.g. when showing context, when using solve) types of the form

$$(h :: A) \rightarrow B$$

where one could use as well

$$A \rightarrow B$$

Elements of $(x :: A) \rightarrow C$ are introduced as before by using

- either $\lambda$-abstraction, i.e.
  $$\lambda(x :: A) \rightarrow t :: (x :: A) \rightarrow B.$$ 
- Requires that $t :: B$ depending on $x :: A$.
- Note that the type $B$ of $t$ depends on $x :: A$.
- or by writing
  $$f(x :: A) :: C = \cdots$$

Elimination is application using the same notation as before.

E.g., if $f :: (x :: A) \rightarrow C$ and $a :: A$, then

$$f[a :: C[x := a]].$$

We can write

$$(n, m :: N) \rightarrow A(n,m)$$

instead of

$$(n :: N) \rightarrow (m :: N) \rightarrow A(n,m)$$

Abbreviations
The “select”-Name Example in Agda

Here follows the code for the select-Name example in Agda, which should be readable for those familiar with Haskell:

```agda
data G = male | female

data maleNames = Tom | Jim

data femaleNames = Jill | Sara
```

The “select”-Name Example in Agda

Names \((g :: G)\)

\[::\] Set
\[=\] case g of
\[\quad (\text{male}) \rightarrow\] maleNames
\[\quad (\text{female}) \rightarrow\] femaleNames

select \((g :: G)\rightarrow\) Names g

\[=\] \(\lambda (g :: G)\rightarrow\) case g of
\[\quad (\text{male}) \rightarrow\] Tom
\[\quad (\text{female}) \rightarrow\] Jill

See `exampleNamesFunction2.agda`.

(e) Derivations vs. Agda Code

In this subsection we look at the relationship between Agda code and the corresponding derivations.

- We consider various examples.
  - **First** we will go through the development of the Agda code.
  - **Then** we will look at, how the corresponding derivations are developed, following each step in the development of the Agda code.

An attempt to define select s.t. select male is not in maleNames, e.g.

\[
\text{select male} = \text{Jill}
\]

or that select female is not in femaleNames, e.g.

\[
\text{select female} = \text{Tom}
\]

will result in a **type error**.
Example 1

We want to derive in Agda
\[ \lambda (a :: A). a :: A \rightarrow A \]

(See example file exampleIdentity.agda)

**Step 1:**
- We need to introduce the type \( A \) first.
- Since we want to have the definition for an arbitrary type \( A \), we postulate (i.e. assume) one type \( A \):
  
  \[
  \text{postulate } A :: \text{Type}
  \]

**Step 2:** We state our goal:

\[
\begin{align*}
  f :: A & \rightarrow A \\
  & = \{! !\}
\end{align*}
\]

Agda is an indentation-sensitive language. The complete definition of \( f \) must be intended otherwise Agda regards this as a new definition.

**Step 3:**
- We want to derive an element of function type \( A \rightarrow A \).
- Elements of the function type \( A \rightarrow A \) are introduced by using \( \lambda \)-terms.
- If introduced as a \( \lambda \)-term, the term in question will be of the form \( \lambda (a :: A) \rightarrow \text{something} \).
- Agda has a command **agda-intro (Intro)** which does this step automatically.
- Has to be executed while the cursor is inside one goal.

**Step 3 (Cont.)**

After executing it we get:

\[
\begin{align*}
  f :: A & \rightarrow A \\
  & = \lambda (h :: \{! !\}) \rightarrow \{! !\}
\end{align*}
\]

(The precise Agda code uses \( \setminus \) instead of \( \lambda \), and \( \rightarrow \) instead of \( \rightarrow \)).
Example 1 (Cont.)

Step 4:
- The first goal, the type of the variable $h$ can be solved automatically.
  Use `agda-solve` (Solve)
- We obtain:

$$f :: A \rightarrow A$$
$$= \lambda(h :: A) \rightarrow \{! !\}$$

Example 1 (Cont.)

Step 4 (Cont)
- We obtain:

$$f :: A \rightarrow A$$
$$= \lambda(a :: A) \rightarrow \{! !\}$$

Example 1 (Cont.)

Step 4 (Cont)
- It is a good idea to rename the variable to something, for instance to $a$:
  This can be done by simple editing.
- We can always edit the current code.
- If one wants to edit parts involving goals, one first has to execute:
  `agda-restart` (Re)Start Agda
  Then one is in a mode where the goals are converted to ordinary symbols and can edit everything.
- At the end of any editing one should execute:
  `agda-load-buffer` (Load Buffer)
  Otherwise the changes will not be known by Agda.

Step 5:
- In order for $\lambda(a :: A) \rightarrow \{! !\}$ to be of type $A \rightarrow A$, $\{! !\}$ must be of type $A$.
- Then this $\lambda$-term computes an element of type $A$ depending on some $a$ of type $A$, which means it is a function of type $A \rightarrow A$.
- So the type of the goal is $A$.
- This can be inspected by using the menu `agda-goalType-of-meta-reduced`
  (Type of goal (unfolded)), which shows the type of the current goal.
  - Has to be executed while the cursor is inside one goal.
  - It shows $A$. 
### Termination Check (Cont.)

Agda has a command `agda-term-check-buffer (Check Termination)`, which checks whether recursive definitions are done properly.

- One should use this command at the end of a session, to avoid black hole recursion.
- If the termination check succeeds, all programs checked will terminate.
- If the termination check fail, it might still be the case that all programs terminate. (One cannot write a universal termination checker, since the Turing halting problem is undecidable).

### Example 1 (Cont.)

**Step 5 (Cont.)**

- We can inspect the context.
- The context contains everything we can use when solving our goal. It contains:
  - $A : \text{Set}$.
  - $f :: A \to A$.
  - $a :: A$.
  - Since we are defining $a$ an element of type $A$ depending on $a :: A$, we can use $a$.

---

**Termination Check**

- On the last slide we had $f :: A \to A$ in the context.
- This appears, since the type checker allows to define functions recursively, independently of whether the recursion terminates or not.
- For the type checker a definition $b :: A = b$ would be legal, although evaluating $b$ doesn’t terminate (black hole recursion).
Example 1, Using Rules

- In **Agda step 1** we postulated $A :: \text{Set}$. This corresponds in the rule system, that we can assume $A : \text{Set}$, i.e. can write this down without any derivation.

- In **Agda step 2** we stated our goal:

$$ f :: A \rightarrow A = \{ ! ! \} $$

In terms of rules this means that we want to derive something of type $A \rightarrow A$. We write for this something $d_0$ and get as conclusion of our derivation:

$$ d_0 : A \rightarrow A $$

**Example 1, Using Rules (Cont.)**

- In **Agda step 3** and **4** we replaced $\{ ! ! \}$ by $\lambda(a :: A) \rightarrow \{ ! ! \}$:

$$ f :: A \rightarrow A = \lambda(a :: A) \rightarrow \{ ! ! \} $$

In terms of rules this means that we replace $d_0$ by $\lambda(a :: A).d_1$ which is derived by an introduction rule:

$$ a : A \Rightarrow d_1 : A \Rightarrow (\lambda a : A.d_1) : A \rightarrow A $$

**Example 1, Using Rules (Cont.)**

- In **Agda step 5** we replaced $\{ ! ! \}$ in $\lambda(a :: A) \rightarrow \{ ! ! \}$ by $a$:

$$ f :: A \rightarrow A = \lambda(a :: A) \rightarrow a $$

In terms of rules this means that we replace $d_1$ by $a$. $a : A \Rightarrow a : A$ follows by an assumption rule:

$$ \frac{}{a : A \Rightarrow a : A} (\lambda a : A.a) : A \rightarrow A $$

The assumption rule will be discussed later.

Essentially it allows to derive if $x : B$ occurs in the context that $x : B$ holds.

**Example 2**

- We consider a derivation of

$$ \lambda(x :: (A \rightarrow A) \rightarrow A).x (\lambda(a :: A) \rightarrow a) :: ((A \rightarrow A) \rightarrow A) \rightarrow A $$

(See example exampleSampleDerivation2.agda).

**Step 1:**

- We postulate $A$:

$$ \text{postulate } A :: \text{Type} $$

- We state our goal:

$$ f :: ((A \rightarrow A) \rightarrow A) \rightarrow A = \{ ! ! \} $$
Example 2 (Cont.)

**Step 2:**
- The type of the goal is a function type, and we can use `agda-intro (Intro)`: we obtain
  \[
  f :: ((A \to A) \to A) \to A \\
  = \lambda (h :: \{! \}) \to \{! \}
  \]
- Using `agda-solve (Solve)` we obtain:
  \[
  f :: ((A \to A) \to A) \to A \\
  = \lambda (h :: (A \to A) \to A) \to \{! \}
  \]

**Step 2 (Cont.):**
- We rename the variable `h` to `x` and use `agda-load-buffer (Load Buffer)` so that Agda realizes this change:
  \[
  f :: ((A \to A) \to A) \to A \\
  = \lambda (x :: (A \to A) \to A) \to \{! \}
  \]

Example 2 (Cont.)

**Step 3:**
- The type of the new goal is `A`, which is the result type of the function we are defining.
- The context contains `f` (for recursive definitions), `A` and `x`.
- `x` is a function of result type `A`. Applying it to its argument would have as result the type of the goal in question.

**Step 3 (Cont.):**
- Therefore we type into the goal `x` and use `agda-refine (Refine)`. Agda will then apply `x` to as many goals as needed in order to obtain an element of the desired type.
- In our case it is one (of type `A \to A`).
- We obtain
  \[
  f :: ((A \to A) \to A) \to A \\
  = \lambda (x :: (A \to A) \to A) \to x \{! \}
  \]
Example 2 (Cont.)

Step 4:
- The type of the new goal is $A \to A$.
- This is since $x :: (A \to A) \to A$ needs to be applied to an element of type $A \to A$ in order to obtain an element of type $A$.
- We try `agda-intro (Intro)` and obtain:

$$f ::= ((A \to A) \to A) \to A$$
$$= \lambda(x :: (A \to A) \to A) \to x (\lambda(h :: \{! !\}) \to \{! !\})$$

Example 2 (Cont.)

Step 4 (Cont.)
- We rename $h$ by $a$, reload the buffer, and obtain:

$$f ::= ((A \to A) \to A) \to A$$
$$= \lambda(x :: (A \to A) \to A) \to x (\lambda(a :: A) \to \{! !\})$$

Example 2 (Cont.)

Step 5
- The new goal has type $A$.
- The complete expression $\lambda(a :: A) \to \{! !\}$ should have type $A \to A$, so $\{! !\}$ must have type $A$.
- The context contains $A :: \text{Set}$, $f$, $x$ and $a$.
- We can use both $x$ and $a$ here.
- There is usually more than one solution for proceeding in Agda.
- This means that we sometimes have to backtrack and try a different solution.
Example 2 (Cont.)

Step 5 (Cont.)

We try \( a :: A \). After inserting it and using \texttt{agda-refine (Refine)} we obtain the following and are done.

\[
\begin{align*}
  f &:: ((A \rightarrow A) \rightarrow A) \rightarrow A \\
  &= \lambda(x :: (A \rightarrow A) \rightarrow A) \rightarrow x \ (\lambda(a :: A) \rightarrow a)
\end{align*}
\]

---

Example 2, Using Rules

Postulating \( A :: \text{Set} \) corresponds to assuming \( A : \text{Set} \) in the rules without deriving it.

Stating the goal means that we have as last line of the derivation:

\[
d_0 : ((A \rightarrow A) \rightarrow A) \rightarrow A
\]

---

Example 2, Using Rules

The next step in the Agda-derivation was to replace the goal by

\[
\lambda(x :: (A \rightarrow A) \rightarrow A) \rightarrow \{! !\}.
\]

This corresponds to replacing \( d_0 \) by \( \lambda(x :: (A \rightarrow A) \rightarrow A) \). \( d_1 \) and having as last step an introduction rule:

\[
\frac{x :: (A \rightarrow A) \rightarrow A \Rightarrow d_1 : A}{(\lambda x : ((A \rightarrow A) \rightarrow A).d_1) :: ((A \rightarrow A) \rightarrow A) \rightarrow A}
\]

The left top judgement can be derived by an \textit{assumption rule} (more about this later).
Example 2, Using Rules

We then used intro on the goal which was then replaced by $\lambda (x :: A) \rightarrow \{! ! \}$. This corresponds to replacing $d_2$ by $\lambda x : A.d_3$ which can be introduced by an introduction rule:

$$
\frac{x : (A \rightarrow A) \rightarrow A \Rightarrow x : (A \rightarrow A) \rightarrow A}{\quad x : (A \rightarrow A) \rightarrow A \Rightarrow \lambda x : A.d_3 : A \rightarrow A}
$$

Finally we used refine with $a$, which replaced the goal by $a$.

This corresponds to replacing $d_3$ by $a$.

$$
\frac{x : (A \rightarrow A) \rightarrow A \Rightarrow x : (A \rightarrow A) \rightarrow A}{\quad x : (A \rightarrow A) \rightarrow A \Rightarrow \lambda a : A.a : A \rightarrow A}
$$

The right hand derivation can again be derived by an assumption rule (more about this later).

Example 3

We derive an element of type

$$A \rightarrow B \rightarrow AB$$

where $AB$ is the product of $A$ and $B$.

(See exampleProductIntro.agda).

Example 3 (Cont.)

Step 1:

We postulate types $A$, $B$:

postulate $A :: \text{Set}$
postulate $B :: \text{Set}$

We introduce the product of $A$, $B$:

This will be a record with element $a : A$, $b : B$.

$$AB :: \text{Set} = \text{sig}\{a :: A; b :: B\}$$
Example 3 (Cont.)

Step 2:
- Our goal is:

\[ f :: A \rightarrow B \rightarrow AB \]

\[ = \{! !\} \]

Step 3:
- We use intro.
  - An element of \( A \rightarrow B \rightarrow AB \) will be of the form

\[ \lambda (a' :: A) \rightarrow \lambda (b' :: B) \rightarrow \{! !\} \]

which is introduced by two introduction steps.
- Agda will immediately carry out both of them.
- We choose to use \( a' \) instead of \( a \), \( b' \) instead of \( b \), since \( a, b \) are used as labels of \( AB \).

Step 3 (Cont)
- After applying intro we get

\[ f :: A \rightarrow B \rightarrow AB \]

\[ = \lambda (h :: \{! !\}) \rightarrow \lambda (h' :: \{! !\}) \rightarrow \{! !\} \]

- After applying agda-solve and renaming of variables we get

\[ f :: A \rightarrow B \rightarrow AB \]

\[ = \lambda (a' :: A) \rightarrow \lambda (b' :: B) \rightarrow \{! !\} \]

Step 4:
- The new goal is of type \( AB \) which is a record type. An element of it can be introduced by an introduction rule.
- Elements of type \( AB \) introduced by the introduction principle will have the form

\[ \text{struct}\{a = \{! !\}; b = \{! !\}\} \]
Example 3 (Cont.)

**Step 4 (Cont.):**
- When using intro we get:

\[
\begin{align*}
  & f :: A \to B \to AB \\
  & = \lambda (a' :: A) \to \lambda (b' :: B) \to \text{struct}\{a = \{! \};} \\
  & \quad b = \{! \}\}
\end{align*}
\]

Example 3 (Cont.)

**Step 5 (Cont):**
- We insert \( a \), use refine and solve the first goal:

\[
\begin{align*}
  & f :: A \to B \to AB \\
  & = \lambda (a' :: A) \to \lambda (b' :: B) \to \text{struct}\{a = a'\}; \\
  & \quad b = \{! \}\}
\end{align*}
\]

Example 3 (Cont.)

**Step 5:**
- The first goal has as context:
  - \( A, B : \text{Set} \),
  - \( AB : \text{Set} \),
  - \( f : A \to B \to AB \),
  - \( a' : A \),
  - \( b' : B \),
  - \( a : A \),
  - \( b : B \).
- \( a : A, b : B \) are the projections of the record we are defining, which might be used recursively.
- Using \( a \) and \( b \) would in our example result in non-termination.

Example 3 (Cont.)

**Step 6:**
- Similarly we can solve the second one:

\[
\begin{align*}
  & f :: A \to B \to AB \\
  & = \lambda (a' :: A) \to \lambda (b' :: B) \to \text{struct}\{a = a';} \\
  & \quad b = b'\}
\end{align*}
\]
Example 3, Using Rules

The definition of $AB$ means that $AB$ abbreviates $A \times B$, which can be derived as follows (using assumptions $A : \text{Set}, B : \text{Set}$):

\[
\frac{A : \text{Set} \quad B : \text{Set}}{A \times B : \text{Set}}
\]

We won’t use this however, since it is required for the assumption rules only, the treatment of which will be delayed until later.

Example 3, Using Rules (Cont.)

Stating the goal corresponds to having as last line of the derivation:

\[
d_0 : A \rightarrow B \rightarrow (A \times B)
\]

Using intro means that we replace $d_0$ by $\lambda a' : A. \lambda b' : B. d_1$, which is introduced by two introduction rules:

\[
\frac{a' : A, b' : B \Rightarrow d_2 : A \quad a' : A, b' : B \Rightarrow d_3 : B}{a' : A, b' : B \Rightarrow \langle d_2, d_3 \rangle : A \times B}
\]

\[
\frac{a' : A \Rightarrow (\lambda b' : B. \langle d_2, d_3 \rangle) : B \rightarrow (A \times B)}{(\lambda a' : A. \lambda b' : B. \langle d_2, d_3 \rangle) : A \rightarrow B \rightarrow (A \times B)}
\]

Example 3, Using Rules (Cont.)

Solving the goals by refining them with $a'$, $b'$ means that we replace $d_2$ by $b$, $d_3$ by $c$:

\[
\frac{a' : A, b' : B \Rightarrow a' : A \quad a' : A, b' : B \Rightarrow b' : B}{a' : A, b' : B \Rightarrow \langle a', b' \rangle : A \times B}
\]

\[
\frac{a' : A \Rightarrow (\lambda b' : B. \langle a', b' \rangle) : B \rightarrow (A \times B)}{(\lambda a' : A. \lambda b' : B. \langle a', b' \rangle) : A \rightarrow B \rightarrow (A \times B)}
\]

The premises require an assumption rule (which will use the derivation of $A \times B$), see later for details.
Example 4

We derive an element of type

\[(A \rightarrow BC) \rightarrow A \rightarrow B\]

where \(BC\) is the product of \(B\) and \(C\). (See exampleProductElim.agda).

Example 4 (Cont.)

Step 1:
- We postulate types \(A, B, C\):
  - \(A :: \text{Set}\)
  - \(B :: \text{Set}\)
  - \(C :: \text{Set}\)
- We introduce the product of \(B, C\):
  \[BC :: \text{Set} = \text{sig}\{b :: B; c :: C\}\]

Example 4 (Cont.)

Step 2:
- Our goal is:
  \[f :: (A \rightarrow BC) \rightarrow A \rightarrow B = \{! !\}\]

Example 4 (Cont.)

Step 3:
- We use intro and get (after using agda-solve and renaming of variables):
  \[f :: (A \rightarrow BC) \rightarrow A \rightarrow B = \lambda(x :: A \rightarrow BC) \rightarrow \lambda(a :: A) \rightarrow \{! !\}\]
Example 4 (Cont.)

Step 4:
- The context has no element with result type $B$ (except of $f$ which results in a circular definition).
- However, $x$ has function type with result type $BC$, which can be projected to $B$.
- We introduce first an element of type $BC$ by a let-expression, and then derive from it the desired element of type $B$:

Step 5:
- We insert as type of variable $bc$ the type $BC$ (using refine) and obtain:

$$f :: (A \to BC) \to A \to B$$
$$= \lambda(x :: A \to BC) \to \lambda(a :: A) \to \text{let bc :: BC}$$
$$= \{! !\}$$

in \{! !\}

Example 4 (Cont.)

Step 4 (Cont.):
- Using agda-let (Make let expression) with variable $bc$ we obtain:

$$f :: (A \to BC) \to A \to B$$
$$= \lambda(x :: A \to BC) \to \lambda(a :: A) \to \text{let bc :: } \{! !\}$$
$$= \{! !\}$$

in \{! !\}

Step 6:
- For solving the first goal (definition of $bc$) we can refine $x$, which has as result type $BC$.

$$f :: (A \to BC) \to A \to B$$
$$= \lambda(x :: A \to BC) \to \lambda(a :: A) \to \text{let bc :: BC}$$
$$= x \{! !\}$$

in \{! !\}
Example 4 (Cont.)

Step 7:
The new goal can be solved by refining it with variable $a$:

$$f :: (A \rightarrow BC) \rightarrow A \rightarrow B$$

$$= \lambda(x :: A \rightarrow BC) \rightarrow \lambda(a :: A) \rightarrow \text{let } bc :: BC \rightarrow x \cdot a$$

$$\text{in } \{! !\}$$

Example 4 (Cont.)

In our rule calculus we don’t introduce a let construction (we could add this).

In order to get close to the derivations, we omit in the Agda derivation the let expression, and replace in the body of it $bc$ by its definition $(x \cdot a)$.

We get

$$f :: (A \rightarrow BC) \rightarrow A \rightarrow B$$

$$= \lambda(x :: A \rightarrow BC) \rightarrow \lambda(a :: A) \rightarrow (x \cdot a).b$$

Example 4 (Cont.)

Step 8:
Currently, Agda doesn’t have any direct support for refining $bc$ to an element of type $B$.

We have to do this by hand, insert $bc.b$, choose refine and obtain:

$$f :: (A \rightarrow BC) \rightarrow A \rightarrow B$$

$$= \lambda(x :: A \rightarrow BC) \rightarrow \lambda(a :: A) \rightarrow \text{let } bc :: BC \rightarrow x \cdot a$$

$$\text{in } bc.b$$

Example 4, Using Rules

Using rules we start with our goal

$$d_0 : (A \rightarrow (B \times C)) \rightarrow A \rightarrow B$$
Example 4, Using Rules (Cont.)

In Agda, we then replace the goal corresponding to $d_1$ by $(x \ a).b$.

In our rule calculus, this reads $\pi_0(x \ a)$.

This can be introduced by two applications of elimination rules:

$\quad x : A \rightarrow (B \times C), a : A \Rightarrow (x : A \rightarrow (B \times C)) : a : A \\
\quad x : A \rightarrow (B \times C), a : A \Rightarrow (x : A \rightarrow (B \times C)) : a \Rightarrow a : A \\
\quad x : A \rightarrow (B \times C), a : A \Rightarrow x : a : B \times C \\
\quad x : A \rightarrow (B \times C), a : A \Rightarrow \pi_0(x \ a) : B \\
\quad x : A \rightarrow (B \times C) : (\lambda x : (A \rightarrow (B \times C)), \lambda a : A. \pi_0(x \ a)) : A \Rightarrow B \\
\quad (\lambda x : (A \rightarrow (B \times C)), \lambda a : A. \pi_0(x \ a)) : (A \rightarrow (B \times C)) : A \Rightarrow B$ \\

The two initial judgements can be introduced by assumption rules.

Presuppositions (Cont.)

A : Set and $x : A \Rightarrow B : Set$ are presuppositions of the judgement

$x : A, y : B \Rightarrow C : Set$.
Presuppositions (Cont.)

- $A : \text{Set}$ and $B : \text{Set}$ are presuppositions of the judgement $A \rightarrow B : \text{Set}$. And of the judgement $A \times B : \text{Set}$.

- The next slide shows the presuppositions of judgements.

### Presuppositions

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : A \Rightarrow \text{Context}$</td>
<td>$\Gamma \Rightarrow A : \text{Set}$</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow A : \text{Set}$</td>
<td>$\Gamma \Rightarrow \text{Context}$</td>
</tr>
<tr>
<td>$\Gamma \Rightarrow A = B : \text{Set}$</td>
<td>$\Gamma \Rightarrow A : \text{Set}$, $\Gamma \Rightarrow B : \text{Set}$</td>
</tr>
</tbody>
</table>

Interactive Theorem Proving, CS 336, Lentterm 2004, Sec. 2(f)
Presuppositions

Furthermore, presuppositions of presuppositions of

\[ \Gamma \Rightarrow \theta \]

are as well presuppositions of

\[ \Gamma \Rightarrow \theta . \]

Example of Presuppositions

\[ x : A, y : B \Rightarrow a = b : (z : C) \times D \]

presupposes:

- \( \emptyset \Rightarrow \) Context,
- \( A : \) Set,
- \( x : A \Rightarrow \) Context,
- \( x : A \Rightarrow B : \) Set,
- \( x : A, y : B \Rightarrow \) Context,
- \( x : A, y : B \Rightarrow C : \) Set,
- \( x : A, y : B, z : C \Rightarrow \) Context,
- \( x : A, y : B, z : C \Rightarrow D : \) Set,
- \( x : A, y : B \Rightarrow (z : C) \times D : \) Set,
- \( x : A, y : B \Rightarrow a : (z : C) \times D ,\)
- \( x : A, y : B \Rightarrow b : (z : C) \times D. \)

Remark on \( A \rightarrow B, A \times B \)

- Note that \( A \rightarrow B \) is an abbreviation for \((x : A) \rightarrow B\) for some fresh \( x \).
- Similarly \( A \times B \) is an abbreviation for \((x : A) \times B\) for some fresh \( x \).
- Therefore the presupposition of \( A \rightarrow B : \) Set (which abbreviates \( \emptyset \Rightarrow A \rightarrow B : \) Set) are:
  - \( \emptyset \Rightarrow \) Context,
  - \( A : \) Set,
  - \( x : A \Rightarrow \) Context,
  - \( x : A \Rightarrow B : \) Set.

(g) The Full Logical Framework

- We would like to add operations on types, such as
  \[ \text{prod} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \]
  which should take two sets and form the product of it.
- The problem is that for this we need
  \[ \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} : \text{Set} \]
  and our rules allow this only if we had
  \[ \text{Set} : \text{Set} \]
Set

Adding

Set : Set

as a rule results however in an inconsistent theory:

- using this rule we can prove everything, especially false formulas.

The corresponding paradox is called Girard's paradox.

Set (Cont.)

Instead we introduce a new level on top of Set called Type.

- So besides judgements \( A : \text{Set} \) we have as well judgements of the form
  \[
  A : \text{Type}
  \]

One rule will especially express

\[
\text{Set} : \text{Type}
\]

- Elements of Type are types, elements of Set are small types.

Jean-Yves Girard

We add rules asserting that if \( A : \text{Set} \) then \( A : \text{Type} \).

Further we add rules asserting that Type is closed under the dependent function type and product.

Since \( \text{Set} : \text{Type} \) we get therefore (by closure under the function type)

\[
\text{Set} \rightarrow \text{Set} \rightarrow \text{Set} : \text{Type}
\]

and we can assign to \( \text{prod} \) above the type

\[
\text{prod} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}
\]

(The definition of \( \text{prod} \) will be given later.)
Set (Cont.)

- However, we cannot use \texttt{prod} in order to form the product of two sets, ie. we cannot introduce

\[
\text{prod } \texttt{Set Set} : \texttt{Set}
\]

since \texttt{Set : Set} does not hold.

Rules for Set

Formation Rule for Set

\[
\text{Set} : \text{Type}
\]

Every Set is a Type

\[
\begin{array}{c}
A : \text{Set} \\
\hline
A : \text{Type}
\end{array}
\]

Closure of Type

- Further we add rules stating that \texttt{Type} is closed under the dependent function type and the dependent product:

Closure of Type under the dependent product

\[
\begin{array}{c}
A : \text{Type} \quad x : A \Rightarrow B : \text{Type} \\
\hline
(x : A) \times B : \text{Type}
\end{array}
\]

Closure of Type under the dependent function type

\[
\begin{array}{c}
A : \text{Type} \quad x : A \Rightarrow B : \text{Type} \\
\hline
(x : A) \rightarrow B : \text{Type}
\end{array}
\]
Nondependent Case

A special case of the above rule is the closure under the non-dependent function type and product. This rule can be derived (e.g. from the premises one can derive using the other rules the conclusion).

Closure of Type under the non-dependent product

\[
\frac{A : \text{Type} \quad B : \text{Type}}{A \times B : \text{Type}} \quad (\timesFType)
\]

Closure of Type under the non-dependent function type

\[
\frac{A : \text{Type} \quad B : \text{Type}}{A \rightarrow B : \text{Type}} \quad (\rightarrowFType)
\]

Equality Versions of the Rules

Formation Rule for Set

\[
\text{Set} = \text{Set} : \text{Type} \quad (\text{SetIsType}^=)
\]

Every Set is a Type

\[
\frac{A = B : \text{Set}}{A = B : \text{Type}} \quad (\text{Set2Type}^=)
\]

Equality Versions of the Rules

Closure of Type under the dependent product

\[
\frac{A = A' : \text{Type} \quad x : A \Rightarrow B = B' : \text{Type}}{(x : A) \times B = (x : A') \times B' : \text{Type}} \quad (\timesFType)
\]

Closure of Type under the dependent function type

\[
\frac{A = A' : \text{Type} \quad x : A \Rightarrow B = B' : \text{Type}}{(x : A) \rightarrow B = (x : A') \rightarrow B' : \text{Type}} \quad (\rightarrowFType)
\]

Similarly for the non-dependent versions of the above.

Context Rules

The types in the contexts, which were before only elements of Set, can now be as well elements of Type.

Therefore we need an additional context rule

\[
\frac{\Gamma \Rightarrow A : \text{Type}}{\Gamma, x : A \Rightarrow \text{Context}} \quad (\text{Context1Type})
\]
Example: prod

We can now introduce \( \text{prod} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \):

First we derive \( X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set} \):

\[
\begin{align*}
\text{Set} & : \text{Type} \\
\frac{X : \text{Set}}{X : \text{Set} \Rightarrow \text{Context}} \\
\frac{X : \text{Set} \Rightarrow \text{Set} : \text{Type}}{X : \text{Set}, Y : \text{Set} \Rightarrow \text{Context}} \\
\frac{X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set}}{X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set}}
\end{align*}
\]

Similarly we derive \( X : \text{Set}, Y : \text{Set} \Rightarrow Y : \text{Set} \).

Example: prod (Cont.)

Now we can derive our desired judgement:

\[
\begin{align*}
X : \text{Set}, Y : \text{Set} & \Rightarrow X : \text{Set} & X : \text{Set}, Y : \text{Set} & \Rightarrow Y : \text{Set} \\
X : \text{Set}, Y : \text{Set} & \Rightarrow X \times Y : \text{Set} \\
X : \text{Set} & \Rightarrow (\lambda Y : \text{Set}.X \times Y) : \text{Set} \rightarrow \text{Set} \\
(\lambda X, Y : \text{Set}.X \times Y) : \text{Set} & \rightarrow \text{Set} \rightarrow \text{Set}
\end{align*}
\]

So define

\[
\text{prod} := \lambda X, Y : \text{Set}.X \times Y
\]

Hierarchies of Types

- If one wants to form

\[
\text{prod}' : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} ,
\]

one needs to have a further level \( \text{Kind} \) above \( \text{Type} \), s.t.

\[
\text{Type} : \text{Kind} .
\]

- Then

\[
\text{Type} \rightarrow \text{Type} \rightarrow \text{Type} : \text{Kind} .
\]
**Rules for Type as a Kind**

**Type is a Kind**

$$\text{Type} : \text{Kind}$$

**Every Type is a Kind**

$$\frac{A : \text{Type}}{A : \text{Kind}} \quad \text{(Type2Kind)}$$

---

**Closure of Kind**

**Closure of Kind under the dependent product**

$$\frac{A : \text{Kind} \quad x : A \Rightarrow B : \text{Kind}}{(x : A) \times B : \text{Kind}} \quad \text{(\times_f^{\text{Kind}})}$$

**Closure of Kind under the dependent function type**

$$\frac{A : \text{Kind} \quad x : A \Rightarrow B : \text{Kind}}{(x : A) \rightarrow B : \text{Kind}} \quad \text{(-_f^{\text{Kind}})}$$

Plus equality versions of the above rules.

---

**Context Rules**

- Again, the context rules have to be expanded:

$$\frac{\Gamma \Rightarrow A : \text{Kind}}{\Gamma, x : A \Rightarrow \text{Context}} \quad \text{(Context_1^{\text{Kind}})}$$

---

**Hierarchies of Types (Cont.)**

- This can be iterated further, forming

  $$\text{Type} = \text{Type}_1, \text{Kind} = \text{Type}_2, \text{Type}_3, \text{Type}_4 \cdots$$

- Agda has a hierarchy of types built in, written as 

  $$\#0$$ (which is `Set`), $$\#1$$ (which is `Type`), $$\#2$$ (in the rule calculus called `Kind`), $$\#3$$ etc.

- So we have

  $$\#0 = \text{Set} : \text{Type},$$
  $$\#0 = \text{Set} : \#2, \#1 = \text{Type} : \#2,$$
  $$\#0 = \text{Set} : \#3, \#1 = \text{Type} : \#3, \#2 : \#3,$$
  $$\#0 = \text{Set} : \#4, \#1 = \text{Type} : \#4, \#2 : \#4, \#3 : \#4,$$
  etc.
3. Data Types

(a) The Set of Booleans

**Formation Rule**

| Bool : Set |

**Introduction Rules**

| tt : Bool | ff : Bool |

**Elimination Rule**

\[
\begin{array}{c}
C : \text{Bool} \rightarrow \text{Set} \\
ic : C \quad ec : C \\
\text{cond} : \text{Bool}
\end{array}
\]

\[
\text{CaseBool} C \ ic \ ec \ cond : C \ cond
\]

**The Set of Booleans (Cont.)**

**Equality Rules**

\[
\begin{array}{c}
C : \text{Bool} \rightarrow \text{Set} \\
\end{array}
\]

\[
\begin{array}{c}
ic : C \ tt \\
ec : C \ ff
\end{array}
\]

\[
\text{CaseBool} C \ ic \ ec \ tt = ic : C \ tt
\]

\[
\text{CaseBool} C \ ic \ ec \ ff = ec : C \ ff
\]
Remarks

- In the above
  - \( \texttt{tt} \) stands for true, \( \texttt{ff} \) stands for false.
  - \( \texttt{ic} \) stands for "if-case", \( \texttt{ec} \) for "else-case".
  - \( \texttt{cond} \) for "condition".

Remarks (Cont.)

- The argument \( C \) above has no computational content.
  - It is not needed in order to compute
    \[
    \text{Case}_{\text{Bool}} \ C \text{ \texttt{ic} \text{ \texttt{ec} \texttt{tt}}} \quad \text{and} \quad \text{Case}_{\text{Bool}} \ C \text{ \texttt{ic} \text{ \texttt{ec} \texttt{ff}}}.
    \]
  - \( C \) is only needed in order to get to allow decidable type checking:
    - In the presence of arguments like this we can decide whether a judgement \( a : B \) is derivable.

Remarks (Cont.)

- We can write the elimination rule in a more compact but less readable way:
  - \[
  \text{Case}_{\text{Bool}} : (C : \text{Bool} \rightarrow \text{Set}) \rightarrow \text{Case}_{\text{Bool}} \ C \text{ \texttt{ic} \texttt{ec} \texttt{tt} \rightarrow (\texttt{ec} : C \text{ \texttt{ff} \rightarrow (\texttt{cond} : \text{Bool} \rightarrow C \text{ \texttt{cond}}.}
  \]
  - \( \texttt{tt}, \texttt{ff} \) are the constructors of \( \text{Bool} \).

Remarks (Cont.)

- The argument \( C \) above has no computational content.
  - It is not needed in order to compute
    \[
    \text{Case}_{\text{Bool}} \ C \text{ \texttt{ic} \text{ \texttt{ec} \texttt{tt}}} \quad \text{and} \quad \text{Case}_{\text{Bool}} \ C \text{ \texttt{ic} \text{ \texttt{ec} \texttt{ff}}}.
    \]
  - \( C \) is only needed in order to get to allow decidable type checking:
    - In the presence of arguments like this we can decide whether a judgement \( a : B \) is derivable.
Remarks (Cont.)

Notice that we then get for $C : \text{Bool} \to \text{Set}$,

\[ iC : C \text{tt}, ec : C \text{ff} \]

\[ f := \text{Case}_{\text{Bool}} C \ ic \ ec \]
\[ : (\text{cond} : \text{Bool}) \to C \text{cond} \]

\[ f \text{tt} = \text{Case}_{\text{Bool}} C \ ic \ ec \text{tt} = ic : C \text{tt}, \]
\[ f \text{ff} = \text{Case}_{\text{Bool}} C \ ic \ ec \text{ff} = ec : C \text{ff}. \]

So we obtain functions from $\text{Bool}$ into other sets without having to write $\lambda(\text{b} : \text{Bool}) \ldots$.

That’s why we choose the argument to eliminate from as the last one.

Remarks (Cont.)

This is similar to the definition of for instance $(\cdot)$ in curried form in Haskell

\[ (+) : \text{int} \to \text{int} \to \text{int}. \]
\[ (+) \ 3 \text{ is the function which takes an integer and adds to it 3.} \]
\[ \text{Shorter than writing } \lambda x.3 + x. \]

Remarks (Cont.)

Note that we have the following order of the arguments of $\text{Case}_{\text{Bool}}$:

- First we have the set into which we eliminate.
- Then follow the cases, one for each constructor.
- Finally we put the element which we are eliminating.

In some sense $\text{Case}_{\text{Bool}}$ is a “then _else _if ” – the condition (if . . .) is the last one.

Example: AND

We want to introduce conjunction

\[ \text{AND} : \text{Bool} \to \text{Bool} \to \text{Bool}. \]

This will be of the form

\[ \text{AND} = \lambda b, c : \text{Bool}. t \]

for some term $t$.

$t$ will be defined by case distinction on $b$, so we get

\[ \text{AND} = \lambda b, c. \text{Case}_\text{Bool} C \ e \ f \ b \]

for some $e, f$. 

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 3(a)
Example: AND

\[ \text{AND} = \lambda b, c. \text{CaseBool } C \ e \ f \ b \]

- \( C \) will be the set into which we are eliminating, depending on a Boolean value.
- It need to be an element of \( \text{Bool} \rightarrow \text{Set} \).
- Therefore we have \( C = \lambda d : \text{Bool}. D \) for some \( D \) which might depend on \( d \).
- The set, into which we are eliminating, is always the same, namely \( \text{Bool} \).
- So \( D = \text{Bool} \) and therefore we have
  \[ C = \lambda d : \text{Bool}. \text{Bool} \].

Two Meanings of Elements of Set

- All elements \( A \) of \( \text{Set} \) have these two meanings:
  - They can be used as terms, which are elements of the type \( \text{Set} \).
  - The corresponding judgements are \( A : \text{Set} \), \( A = A' : \text{Set} \).
  - And they can be used as sets, which have elements.
  - The corresponding judgements are \( a : A \) and \( a = a' : A \).

Example: AND

- Note that in
  \[ \lambda d : \text{Bool}. \text{Bool} \]
  \( \text{Bool} \) occurs in two different meanings:
  - The first occurrence is that of a set.
  - \( d \) is chosen here as an element of that set.
  - The second occurrence is that as an element of another type, namely \( \text{Set} \).
  - So here \( \text{Bool} \) is a term.

Example: AND

- So
  \[ \text{AND} = \lambda b, c. \text{CaseBool} \ (\lambda d : \text{Bool}. \text{Bool}) \ e \ f \ b \]
  for some \( e, f \).

  For conjunction we have:
  - If \( b \) is true then
    \[ b \land c = \text{tt} \land c = c \]
    So the if-case \( e \) above is \( c \).
  - If \( c \) is false then
    \[ b \land c = \text{ff} \land c = \text{ff} \]
    So the else-case \( f \) above is \( \text{ff} \).
**Example: AND**

- In total we define therefore

\[ \text{AND} = \lambda(b, c : \text{Bool}). \text{Case}_\text{Bool} (\lambda(d : \text{Bool}). \text{Bool}) \ c \ ff \ b \]

: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}

- We verify the correctness of this definition:

\[ \text{AND} \ tt \ c = \text{Case}_\text{Bool} (\lambda(d : \text{Bool}). \text{Bool}) \ c \ ff \ tt = c. \]

as desired.

\[ \text{AND} \ ff \ c = \text{Case}_\text{Bool} (\lambda(d : \text{Bool}). \text{Bool}) \ c \ ff \ ff = ff. \]

Correct as desired.

---

**Select Example**

- Then we can define the function

\[ \text{SelectBool} : (b : \text{Bool}) \rightarrow \text{Names} \ b \]

\[ \text{SelectBool} \ tt = \text{Tim} \]

\[ \text{SelectBool} \ ff = \text{Sara} \]

as follows:

\[ \text{SelectBool} = \text{Case}_\text{Bool} \ \text{Names} \ \text{Tim} \ \text{Sara} \]

- Note that by using twice the \( \eta \)-rule we get that

\[ \text{SelectBool} \]

\[ = \lambda b : \text{Bool}. \text{Case}_\text{Bool} (\lambda d : \text{Bool}. \text{Names} \ d) \ \text{Tim} \ \text{Sara} \ b \]

---

**Select Example**

- Assume we have introduced in type theory

\[ \text{Names} : \text{Bool} \rightarrow \text{Set} , \]

\[ \text{Names} \ tt = \text{MaleNames} , \]

\[ \text{Names} \ ff = \text{FemaleNames} . \]

---

**Select Example**

- We verify the correctness of \( \text{SelectBool} \):

\[ \text{SelectBool} \ tt = \text{Case}_\text{Bool} \ \text{Names} \ \text{Tim} \ \text{Sara} \ tt = \text{Tim} , \]

\[ \text{SelectBool} \ ff = \text{Case}_\text{Bool} \ \text{Names} \ \text{Tim} \ \text{Sara} \ ff = \text{Sara} . \]
Derivation of AND

We derive in the following \( \text{AND} : \text{Bool} \to \text{Bool} \to \text{Bool} \).

We write \( \text{Bool} \), if it

- is a type in \textit{boldface red},
- and if it is a term, in \textit{italic blue}.

First we derive
\[
\begin{align*}
&b : \text{Bool}, c : \text{Bool} \Rightarrow \lambda(d : \text{Bool}).\text{Bool} : \text{Bool} \to \text{Set}: \\
& \quad \text{Bool} : \text{Set} \\
& \quad \frac{}{b : \text{Bool} \Rightarrow \text{Context}} \\
& \quad \frac{}{b : \text{Bool} \Rightarrow \text{Bool} : \text{Set}} \\
& \quad \frac{}{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Context}} \\
& \quad \frac{}{b : \text{Bool}, c : \text{Bool} \Rightarrow \text{Bool} : \text{Set}} \\
& \quad \frac{}{b : \text{Bool}, c : \text{Bool}, d : \text{Bool} \Rightarrow \text{Context}} \\
& \quad \frac{}{b : \text{Bool}, c : \text{Bool}, d : \text{Bool} \Rightarrow \text{Bool} : \text{Set}} \\
& \quad \frac{}{b : \text{Bool}, c : \text{Bool} \Rightarrow \lambda(d : \text{Bool}).\text{Bool} : \text{Bool} \Rightarrow \text{Set}}
\end{align*}
\]
Derivation of AND

Using part of the proof above, we derive

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow c : (\lambda(d : \textit{Bool}). \textit{Bool}) \ \text{tt} \]

\[ \ldots \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{Context} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow c : \textit{Bool} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow (\lambda(d : \textit{Bool}). \textit{Bool}) \ \text{tt} : \text{Set} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow (\lambda(d : \textit{Bool}). \textit{Bool}) \ \text{tt} \]

We derive

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{ff} : (\lambda(d : \textit{Bool}). \textit{Bool}) \ \text{ff} \]

\[ \ldots \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{Context} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{ff} : \textit{Bool} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow (\lambda(d : \textit{Bool}). \textit{Bool}) \ \text{ff} : \text{Set} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{ff} : (\lambda(d : \textit{Bool}). \textit{Bool}) \ \text{ff} \]

Finally we obtain our judgement (we stack the premises of the rule because of lack of space):

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow (\lambda(d : \textit{Bool}). \textit{Bool}) : \textit{Bool} \rightarrow \text{Set} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{ff} : (\lambda(d : \textit{Bool}). \textit{Bool}) : \text{ff} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{ff} : (\lambda(d : \textit{Bool}). \textit{Bool}) \rightarrow \text{ff} \]

Derivation of AND

Finally we obtain our judgement (we stack the premises of the rule because of lack of space):

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow (\lambda(d : \textit{Bool}). \textit{Bool}) : \textit{Bool} \rightarrow \text{Set} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{ff} : (\lambda(d : \textit{Bool}). \textit{Bool}) : \text{ff} \]

\[ b : \textit{Bool}, c : \textit{Bool} \Rightarrow \text{ff} : (\lambda(d : \textit{Bool}). \textit{Bool}) \rightarrow \text{ff} \]

Elimination into Type

We can extend add elimination and equality rules, having as result \textit{Type}:

\textbf{Elimination Rule into Type}

\[ C : \textit{Bool} \rightarrow \textit{Type} \quad \textit{ic} : C \ \text{tt} \quad \textit{ec} : C \ \text{ff} \quad \text{cond} : \textit{Bool} \]

\[ \text{Case}_{\textit{Bool}}^{\textit{Type}} C \ \text{ic} \ \text{ec} \ \text{cond} : C \ \text{cond} \]

\textbf{Equality Rules into Type}

\[ C : \textit{Bool} \rightarrow \textit{Type} \quad \textit{ic} : C \ \text{tt} \quad \textit{ec} : C \ \text{ff} \]

\[ \text{Case}_{\textit{Bool}}^{\textit{Type}} C \ \text{ic} \ \text{ec} \ \text{tt} = \text{ec} : C \ \text{tt} \]

\[ C : \textit{Bool} \rightarrow \textit{Type} \quad \textit{ic} : C \ \text{tt} \quad \textit{ec} : C \ \text{ff} \]

\[ \text{Case}_{\textit{Bool}}^{\textit{Type}} C \ \text{ic} \ \text{ec} \ \text{ff} = \text{ec} : C \ \text{ff} \]
Elimination into Type (Cont.)

We can extend this into an elimination rule into Kind or other higher types.

Bool in Agda (Cont.)

The definition of Bool above is treated in Agda as an abbreviation for the following three more fundamental Agda definitions:

```agda
Bool :: Set
  = data tt | ff

tt :: Bool
  = tt@Bool

ff :: Bool
  = ff@Bool
```

Bool in Agda

We introduce `Bool` by simply listing its constructors (similarly to Haskell syntax):

```agda
data Bool = tt | ff
```

This introduces as well constants

```agda
tt :: Bool
ff :: Bool
```

With this syntax, each constructor can occur at most once in a data type,

i.e. we cannot define a second type having constructor `tt",

e.g. for defining `True` (which is used later):

```agda
data True = tt
```

Notation for Constructors

The official notation for a constructor of a set `A` is

```agda
C@A
```

The notation `C@A` is, what is displayed, when evaluating expressions in Agda.

This notation is necessary, since a constructor might belong to different sets.

For instance we can introduce both

```agda
Bool :: Set = data tt | ff
True :: Set = data tt
```

In this situation we need to be able to distinguish between `tt@Bool` and `tt@True` in order to get decidable type checking.
**Notation for Constructors**

Forcing the user to use different names for constructors doesn’t help since we will later introduce as well sets of the form

\[ D \ (a :: A) \]
\[ :: \text{Set} \]
\[ = \text{data } C \cdot \cdot \cdot \]

Now \( C\text{@}(D\ a) \) can be a constructor of \( D\ a \) for any \( a : A \).

Using \( C \) alone would cause problems with decidable type checking.

If Agda can resolve the type itself, one can write \( C\text{@}_\) instead of \( C\text{@}A \).

**Notation for Constructors**

However the abbreviation

\[ \text{data } A = C | D | \cdots \]

can be used only if

- one is defining a set \( A \) (and not a type \( A \)),
- and if the set one is defining has no parameter.
- So it cannot be use in order to define

\[ D \ (a :: A) \]
\[ :: \text{Set} \]
\[ = \text{data } C \cdot \cdot \cdot \]

**Bool in Agda (Cont.)**

It is still recommended to avoid the use of the symbol \( \text{@} \), and for instance define in the above situation by hand

\[ C \ (a :: A) \]
\[ :: D \]
\[ = C\text{@}_ \]

The abbreviation

\[ \text{data } \text{Bool} = \text{tt} | \text{ff} \]

does this automatically.
Bool in Agda (Cont.)

- The definition of `Bool` as above *doesn’t prevent the definition of another set* with constructors `tt` or `ff`.
- This syntax is the only one allowed, if one defines a set using the **data keyword depending on arguments**. More about this later.

Internally, `tt` will always be represented as `tt@Bool`, similarly for `ff`.

So Agda **evaluates `tt` to `tt@Bool`**.

This can be seen when using of the agda methods for evaluating a term.

Evaluation of terms in Agda

- Agda has several methods for evaluating expressions:
  - `agda-compute-whnf`, “Compute weak head normal form”,
  - `agda-compute-whnfs`, “Compute weak head normal form strict”,
  - `agda-nfC`, “Compute to a depth”,
  - `agda-nfC100`, “Compute to depth 100”.

The above mentioned methods can be executed (directly or by using the goal-menu), while in a goal.

- An expression typed into the goal will be taken as default input to that function.
- But that can be modified.

The methods follow different evaluating strategies.

- **Compute weak head normal form** reduces a term until it starts with a constructor or the outer most function doesn’t reduce any further, even if its arguments are evaluated.
- **Compute to depth 100** seems to work best in most cases.
Elimination in Agda is **based on case distinction.** Assume we want to define
\[ f : \text{Bool} \rightarrow \text{Bool}, \text{s.t.} \]
\[ f \; \text{tt} = \text{ff}, \]
\[ f \; \text{ff} = \text{tt}. \]
So we have the goal:

\[
f \; (x :: \text{Bool}) :: \text{Bool} = \{! !\}
\]

We can then type into the goal \( x \) and choose the menu item **“agda-case”**.
This introduces a case distinction by the constructor used for introducing \( x \):
\( x \) could have been introduced as \( \text{tt} \) or \( \text{ff} \).
The goal expands to:

\[
f \; (x :: \text{Bool}) :: \text{Bool} = \text{case } x \text{ of} \]
\[
\text{tt} \rightarrow \{! !\} \]
\[
\text{ff} \rightarrow \{! !\}
\]
Case Distinction (Cont.)

- Now we can solve the new goals by inserting
  - \texttt{ff} into the first one,
  - \texttt{tt} into the second one.
- We obtain a function:

\[
\begin{align*}
  f \ (x :: \text{Bool}) \\
  :: \text{Bool} \\
  = \text{case } x \text{ of} \\
  \ (\text{tt}) & \rightarrow \text{ff} \\
  \ (\text{ff}) & \rightarrow \text{tt}
\end{align*}
\]

- \(f \ x\) is the \textit{negation of} \(x\).

Testing the Defined Function

- So we
  - type in a dummy goal:
    \[
    \textit{test} :: \text{Set} \\
    = \{! !\}
    \]
  - move to the new goal
  - choose \textit{compute weak head normal form strict} or another evaluation method of Agda,
  - and type into the mini-buffer \(f \texttt{tt}\).
- The result shown is \(\texttt{ff@}\).

Testing the Defined Function

- We can test our function by using one of the evaluation methods of Agda, e.g.
  \textit{compute weak head normal form strict}.
- We have to create a goal for this.
  - The reduction machinery is \textit{context dependent}.
  - The context depends on where in the buffer we are.
  - See the above example where \(x\) was depending on the goal \texttt{tt} or \texttt{ff}.
  - Not every place in the buffer is a good place.
  - \textit{Good places for context are goals}, and that's the only place where Agda allows us to \textit{compute the weak head normal form of expressions}.

(b) The Finite Sets

- \textit{Bool} can be generalised to sets having \(n\) elements (\(n\) a fixed natural number):

  Formation Rule
  \[
  \text{Fin}_n :: \text{Set}
  \]

  Introduction Rules
  \[
  A^n_k : \text{Fin}_n
  \]

  (for \(k = 0, \ldots, n - 1\))
Rules for $\text{Fin}_n$

Elimination Rule

\[ C : \text{Fin}_n \rightarrow \text{Set} \]
\[ s_0 : C A^n_0 \]
\[ s_1 : C A^n_1 \]
\[ \ldots \]
\[ s_{n-1} : C A^n_{n-1} \]
\[ a : \text{Fin}_n \]
\[ \text{Case}_n C s_0 \ldots s_{n-1} a : C a \]

The Finite Sets (Cont)

Equality Rules

\[ C : \text{Fin}_n \rightarrow \text{Set} \]
\[ s_0 : C A^n_0 \]
\[ s_1 : C A^n_1 \]
\[ \ldots \]
\[ s_{n-1} : C A^n_{n-1} \]
\[ \text{Case}_n C s_0 \ldots s_{n-1} A^n_k = s_k : C A^n_k \]

(for $k = 0, \ldots, n - 1$).

Remark: Note that we have just introduced infinitely many rules (for each $n \in \mathbb{N}$ and $k = 0, \ldots, n - 1$).

Omitting Premises in Equality Rules

Since the premises of the equality rule can in most cases be determined from the introduction and elimination rules, we will usually omit them, when writing down equality rules.

So we write for instance for the previous rule:

\[ \text{Case}_n C s_0 \ldots s_{n-1} A^n_k = s_k : C A^n_k \]

We sometimes even omit the type:

\[ \text{Case}_n C s_0 \ldots s_{n-1} A^n_k = s_k \]

More Compact Elimination Rules

\[ \text{Case}_n : (C : \text{Fin}_n \rightarrow \text{Set}) \]
\[ \rightarrow (s_0 : C A^n_0) \]
\[ \rightarrow \ldots \]
\[ \rightarrow (s_{n-1} : C A^n_{n-1}) \]
\[ \rightarrow (a : \text{Fin}_n) \]
\[ \rightarrow C a \]
Elimination into Type

- Similarly as for Bool we can write down elimination rules, where:
  - \( C : \text{Fin}_n \rightarrow \text{Type} \) (instead of \( C : \text{Fin}_n \rightarrow \text{Set} \)).
- This can be done for all sets defined later as well.

Rules for True

- True is the special case \( \text{Fin}_n \) for \( n = 1 \) (we write true for \( A_0^1 \)):
  - Formation Rule: \( \text{True} : \text{Set} \)
  - Introduction Rules: \( \text{true} : \text{True} \)
  - Elimination Rule: \( C : \text{True} \rightarrow \text{Set} \quad c : C \text{ true} \quad t : \text{True} \)
    \[
    \text{Case}_\text{True} \ c \ t : C \ t
    \]
  - Equality Rule: \( C : \text{True} \rightarrow \text{Set} \quad c : C \text{ true} \)
    \[
    \text{Case}_\text{True} \ c \ 	ext{true} = c : C \text{ true}
    \]

Rules for True (Cont.)

- Case\(\text{True} is \text{computationally not very interesting.}\)
  - Case\(\text{True} c \) is the untyped function \( \lambda x. c \).
  - However, in Agda we might not be able to derive
    \[
    \lambda(t : \text{True}).c : (t : \text{True}) \rightarrow C \ t
    \]
- From a logic point of view, it expresses:
  - From an element of \( C \text{ true} \) we obtain an element of \( C \ t \) for every \( t : \text{True} \).
  - So there is no \( C : \text{True} \rightarrow \text{Set} \) s.t. \( C \text{ true} \) is inhabited, but \( C \ x \) is not inhabited for some other \( x : \text{True} \).
  - This means that all elements of \( x \) of type \( \text{True} \) are indistinguishable from \( \text{true} \), i.e. they are identical to \( \text{true} \).
  - This equality is called \text{Leibnitz equality}.

Formulae as Types

- In type theory, formulas are certain types.
  - A formula expressed as a type, is type-theoretically true, if it has an element.
  - The elements of such a type are proofs of this formula.
  - Therefore \textbf{Truth in type theory means provability}.
  - True has exactly one proof, and corresponds therefore to the always always true formula.
Rules for False

False is the special case $\text{Fin}_n$ for $n = 0$:

**Formation Rule**

\[
\text{False} : \text{Set}
\]

**There is no Introduction Rule**

**Elimination Rule**

\[
C : \text{False} \rightarrow \text{Set} \quad f : \text{False} \\
\text{CaseFalse } f : C \ f
\]

**There is no Equality Rule**

False

- CaseFalse expresses: from an element $f$ of False we obtain an element of any set (which might depend on $f$).
- Considered as a formula, this means: from a proof of False we obtain a proof of every other formula.
- I.e. False implies everything.
- In logic this principle is called “Ex falsum quodlibet” (from the absurdity follows anything).
- E.g. A false formula like “0 = 1” or “Swansea lies in Germany” implies everything.

False (Cont.)

- False has no elements.
- It is the formula, which is always false, since it has no proofs.
- Often called falsum or absurdity.

- CaseFalse has no computational meaning, since there is no element it can be applied to.
- Applies of course only if we are working in a terminating type theory.
- If we had full recursion, we could define $f : \text{False}$ by $f = f$.
- However that $f$ doesn’t reduce to canonical form.
- That’s why it’s important to carry out the termination check in Agda, otherwise one obtains for instance elements of False.
**Finite Sets in Agda**

- **Finite sets** can be introduced by giving **one constructor for each element**. E.g.
  
  ```agda
data Colour = blue | red | green
```

- With this we obtain `red :: Colour`

---

**False in Agda**

- In Agda we can define the empty set as a “data”-set with **no constructors**:
  
  ```agda
data False =
```

- If we want to solve
  
  ```agda
g  (x :: False) :: Bool = { ! ! }
```

  we can insert into the goal `x` and choose menu-item “agda-case”.

---

**Finite Sets in Agda (Cont.)**

- Elimination is done via case distinction.

- In the “Colour” example above for instance, we can define

  ```agda
is_red  (c :: Colour) :: Bool = case c of
  (red)  → tt
  (green) → ff
  (blue)  → ff
  ```
Example for the Use of False

Assume the **type of trees**:

```haskell
data Tree = oak | pine | spruce
```

Below we will show, how to introduce a function

```haskell
IsConifer :: Tree → Set
```

s.t.

- IsConifer oak = False
- IsConifer pine = True
- IsConifer spruce = True

Example for the Use of False

If we want to define a function from trees, which are conifers, into another set, we can do so by requiring an additional argument "IsConifer":

```haskell
f (t :: Tree) (p :: IsConifer t) :: A = case t of
  oak → case p of { }
  pine → ⋯
  spruce → ⋯
```

Example for the Use of False

In order to use \( f \) we have to **know** that \( t \) is a conifer, i.e. we have to provide an argument \( p \) which expresses the fact that we know this.

Note that we **don’t have to invent a result** of \( f \) in case \( t \) is an oak tree.

Example 2 for the Use of False

Similarly we can introduce a **stack**, together with a predicate

```haskell
NonEmpty :: Stack → Set
```

s.t.

- NonEmpty s = False
  
  if \( s \) is the empty stack.

Now we can define

```haskell
pop (s :: Stack) (p :: NonEmpty s) :: Stack = ⋯
```
Example 2 for the Use of False

- Again we don’t have to provide a result, in case s is empty.

True in Agda

- The definition of True in Agda is straightforward:
  data True = true

- Case distinction will require to solve the case true:

\[ g \ (x :: \text{True}) \]
\[ :: \ \text{Bool} \]
\[ = \ \text{case } x \ \text{of} \]
\[ (\text{true}) \rightarrow \{! \} \]

(c) Atomic Formulae

Full title of this section: Atomic formulae and the Traffic Light Example.

- We have introduced two formulae:
  - True, the always true formula.
    - Corresponds to truth value \( \text{tt} : \text{Bool} \).
  - False, the always false formula.
    - Corresponds to truth value \( \text{ff} : \text{Bool} \).

Atomic Formulae

- A formula expressing equality between two elements of \( \text{Fin}_n \) (for fixed \( n \)) can now be introduced as follows:
  - Define a function
    \[ \text{Eq}_{\text{in, Bool}} : \text{Fin}_n \rightarrow \text{Fin}_n \rightarrow \text{Bool} \]
    s.t.
    \[ \text{Eq}_{\text{in, Bool}} A^n_i A^n_j = \text{true} \]
    \[ \text{Eq}_{\text{in, Bool}} A^n_i A^n_j = \text{false} \text{ for } i \neq j \]
  - \( \text{Eq}_{\text{in, Bool}} \) can be defined easily (for fixed \( n \)) by case distinction on its two arguments.
Atomic Formulae

Now apply an operation

\[ \text{atom} : \text{Bool} \to \text{Set} \]

which maps the truth value to the corresponding formula, i.e. define now

\[ \text{Eq}_n : \text{Fin}_n \to \text{Fin}_n \to \text{Set} \]
\[ \text{Eq}_n x y = \text{atom}(\text{Eq}_{n, \text{Bool}} x y) \]

atom can be defined as follows:

\[ \text{atom} = \text{Case}_{\text{Type}}^{\text{Type}} (\lambda b : \text{Bool}.\text{Set}) \text{ True False} \]

atom in Agda

\[
\begin{align*}
\text{atom} \ (b :: \text{Bool}) \\
:: \text{Set} \\
&= \text{case} \\n&\quad b \ of \\n&\quad (tt) \ \to \ \text{True} \\n&\quad (ff) \ \to \ \text{False}
\end{align*}
\]

atom is defined s.t.

\[
\begin{align*}
\text{atom} \ \text{tt} \ &= \ \text{True} \\
\text{atom} \ \text{ff} \ &= \ \text{False}
\end{align*}
\]

So we get for \( \text{Eq}_n \) above

\[
\begin{align*}
\text{Eq}_n A^n_i A^n_j &= \text{True} \\
\text{Eq}_n A^n_i A^n_j &= \text{False} \ \text{for} \ i \neq j
\end{align*}
\]

So

- \( \text{Eq}_n A^n_i A^n_j \) is inhabited, has a proof, is true;
- for \( i \neq j \), \( \text{Eq}_n A^n_i A^n_j \) is not inhabited, has not a proof, is false.
Decidable Predicates

- In general, atom allows us to define **decidable predicates** on sets.
- A predicate is **decidable** if it can be decided by a Boolean valued function.
- E.g. **equality on natural numbers** is decidable, since we can define a function \( \text{Eq}_{\text{N}, \text{Bool}} : \text{N} \rightarrow \text{N} \rightarrow \text{Bool} \) which decides it.
- Equality on **functions** \( \text{N} \rightarrow \text{N} \) is **undecidable**, since we cannot define such a function – in order to check equality between \( f \) and \( g \) we need to check equality between \( f \, n \) and \( g \, n \) for all \( n : \text{N} \).

Decidable Predicates (Cont.)

- Assume we have a **set of states** of a system \( A \).
  - E.g. the set of states a railway controller can choose.
- Assume we have a function \( f : A \rightarrow \text{Bool} \).
  - E.g. \( f \, a \) means: **state a is safe**.
- Let now \( g : A \rightarrow \text{Set}, \, g \, a = \text{atom}(f \, a) \).
  - If \( f \, a \) is **true** (e.g. \( a \) is safe), \( g \, a \) is **inhabited**.
  - If \( f \, a \) is **false** (e.g. \( a \) is unsafe), \( g \, a \) is **not inhabited**.
- Now, the existence of a \( h : (a : A) \rightarrow g \, a \) means:
  - For all \( a : A \) we have \( g \, a \) is **inhabited**, i.e. for all \( a : A, \, f \, a \) is **true**,
  - e.g. for all \( a : A, \, a \) is **safe**.

The Traffic Light Example

- Assume a **road crossing**, controlled by **traffic lights**:
- Assume from each direction \( A, \, A', \, B, \, B' \) there is one traffic light,
  - but \( A \) and \( A' \) always coincide, similarly \( B \) and \( B' \).
The Set of Physical States

- For simplicity assume that each traffic light is either red or green:
  
data Colour = red | green

- The set of physical states of the system is given by a pair, determining the colour of A (and therefore as well A') and of B (and B')
  
  \[
  \text{Phys\_State :: Set = sig}
  \]
  
  sigA :: Colour
  
  sigB :: Colour

---

The Set of Control States

- The set of control states is a set of states of the system, a controller of the system can choose.
  
  - Each of these states should be safe.
  - In our example, all safe states will be captured (this can usually be only achieved in small examples).

- A complete set of control states consists of:
  
  - Allred – all signals are red.
  - Agreen – signal A (and A') is green, signal B is red.
  - Bgreen – signal B is green, signal A is red.

---

Control States to Physical States

- We define the state of signals A, B depending on a control state:
  
  \[
  \text{toSigA (s :: Control\_State) :: Colour = case s of}
  \]
  
  (Allred) → red
  
  (Agreen) → green
  
  (Bgreen) → red
Control States to Physical States

\[
toSigB \ (s :: \text{Control\_State}) \\
:: \text{Colour} \\
= \text{case } s \text{ of} \\
\quad \text{(Allred)} \rightarrow \text{red} \\
\quad \text{(Agreen)} \rightarrow \text{red} \\
\quad \text{(Bgreen)} \rightarrow \text{green}
\]

Safety Predicate

- We define now \textbf{when a physical state is safe}:
  - It is \textbf{safe iff not both signals are green}.
  - We define now a corresponding predicate \textbf{directly}, without defining first a Boolean function.
  - We first define a predicate depending on two signals:

\[
\text{CorAux} \ (a, b :: \text{Colour}) \\
:: \text{Set} \\
= \text{case } a \text{ of} \\
\quad \text{(red)} \rightarrow \text{True} \\
\quad \text{(green)} \rightarrow \text{case } b \text{ of} \\
\quad \quad \text{(red)} \rightarrow \text{True} \\
\quad \quad \text{(green)} \rightarrow \text{False}
\]

Safety Predicate (Cont.)

- Now we define:

\[
\text{Cor} \ (s :: \text{Phys\_State}) \\
:: \text{Set} \\
= \text{CorAux} \ s.\text{sigA} \ s.\text{sigB}
\]

\textbf{Remark}: In some cases in order to define a function from some \textbf{product} (i.e. a \textbf{sig-set}) into some other set, it is better first to \textbf{introduce an auxiliary function}, depending on the components of that product.

- In the current example this wouldn’t have caused problems, but in more complex examples it does (due to the lack of the $\eta$-rule).
Safety of the System

Now we show that all control states are safe:

\[
\text{cor\_proof} \ (s :: \text{Control\_State}) \\
:: \text{Cor}(\text{phys\_state} \ s) = \text{case} \ s \ \text{of} \\
\quad \text{(Allred)} \rightarrow \text{true} \\
\quad \text{(Agreen)} \rightarrow \text{true} \\
\quad \text{(Bgreen)} \rightarrow \text{true}
\]

Safety of the System (Cont.)

The first element \text{true} was an element of \text{Cor}(\text{phys\_state Allred}), which reduces to \text{True}.
Similar for the other two elements.
This works only because each control state corresponds to a correct physical state.
If this hadn’t been the case, we would have gotten instances where the goal to solve is \text{False}, which we can’t solve.

Safety of the System (Cont.)

If one makes a mistake which results in an unsafe situation
\[\text{e.g. sets toSigB Agreen = green,}\]
then in the last step we obtain one goal of type \text{False}.
Then we can’t solve this goal directly and cannot prove the correctness.
(We could in Agda solve this goal by using full recursion.
\[\text{e.g. solve this goal as cor\_proof Agreen,}\]
but this would be rejected by the termination check.)

(d) The Disjoint Union of Sets

The disjoint union \[A + B\] of two sets \[A\] and \[B\] is the union of \[A\] and \[B\],
but defined in such a way that we can decide whether an element of this union is originally from \[A\] or \[B\].
This is distinguished by having constructors \[\text{inl} : A \rightarrow A + B\] and \[\text{inr}\].
Elements from \[a : A\] are inserted into \[A + B\] as \[\text{inl} \ a : A + B\],
elements from \[b : B\] are inserted into \[A + B\] as \[\text{inr} \ b : A + B\].
\[\text{inl}\] stands for “in-left”, \[\text{inr}\] for “in-right”.
If we have \[a : A\] and \[a : B\], then \[a\] is represented both as \[\text{inl} \ a\] and \[\text{inr} \ a\] in \[A + B\].
Disjoint Union

Informally, if

\[ A = \{1, 2\} \]

and

\[ B = \{3, 4, 5\} \]

then

\[ A + B = \{\text{inl}(1), \text{inl}(2), \text{inr}(3), \text{inr}(4), \text{inr}(5)\} \]

Each element of \( A + B \) is

- either of the form \( \text{inl}(a) \) for some \( a : A \)
- or of the form \( \text{inr}(b) \) for \( b : B \).

Comparison with the Product

Note that, if \( A \) is empty, then

\[ A + B = \{\text{inr}(b) \mid b : B\} \]

which has a copy of each element of \( B \).

\( A \times B \) is empty, since we cannot form a pair \( p(a, b) \) where \( a : A \) and \( b : B \), since there is no element \( a : A \).

Rules for \( A + B \)

- **Formation Rule**
  
  \[
  \frac{A : \text{Set} \quad B : \text{Set}}{A + B : \text{Set}}
  \]

- **Introduction Rules**
  
  \[
  \frac{A : \text{Set} \quad B : \text{Set} \quad a : A}{\text{inl} \ A \ B \ a : A + B} \]
  
  \[
  \frac{A : \text{Set} \quad B : \text{Set} \quad b : B}{\text{inr} \ A \ B \ b : A + B} \]
Rules for \( A + B \)

**Elimination Rules**

\[
\begin{align*}
A &: \text{Set} \\
B &: \text{Set} \\
C &: (A + B) \rightarrow \text{Set} \\
cl &: (a : A) \rightarrow C \ (\text{inl } A B a) \\
cr &: (b : B) \rightarrow C \ (\text{inr } A B b) \\
d &: A + B \\
\end{align*}
\]

\[
\text{Case}_+ A B C \ cl \ cr \ d : C \ d
\]

**Equality Rules**

\[
\begin{align*}
\text{Case}_+ A B C \ cl \ cr \ (\text{inl } A B a) &= cl a : C \ (\text{inl } A B a) \\
\text{Case}_+ A B C \ cl \ cr \ (\text{inr } A B b) &= cr b : C \ (\text{inr } A B b)
\end{align*}
\]

(cl, cr stand for “case left”, “case right”).

---

**Logical Framework Version**

A more compact notation is:

- \((+) : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}\), written infix.
- \(\text{inl} : (A, B : \text{Set}) \rightarrow A \rightarrow (A + B)\).
- \(\text{inr} : (A, B : \text{Set}) \rightarrow B \rightarrow (A + B)\).
- \(\text{Case}_+ : (A, B : \text{Set}) \rightarrow (C : (A + B) \rightarrow \text{Set}) \rightarrow ((a : A) \rightarrow C \ (\text{inl } A B a)) \rightarrow ((b : B) \rightarrow C \ (\text{inr } A B b)) \rightarrow (d : A + B) \rightarrow C \ d\).

---

**Disjoint Union in Agda**

The disjoint union can be defined as a “data”-set having two constructors \(\text{inl} \) (in-left) and \(\text{inr} \) (inright):

\[
\begin{align*}
(+) &: (A :: \text{Set}) \\
(B :: \text{Set}) \\
:: &: \text{Set} \\
= &: \text{data inl}(a :: A) | \text{inr}(b :: B)
\end{align*}
\]

---

**Disjoint Union in Agda (Cont.)**

The notation \((+)\) means, that \(+\) can be used infix.

Now we have, if \(A, B :: \text{Set}\):

- \(\text{inl}@((A + B) :: A \rightarrow (A + B))\)
- \(\text{inr}@((A + B) :: B \rightarrow (A + B))\)

This can be checked using the menu “infer type” in a dummy goal.

Note that we cannot assign a type to \(\text{inl}@_\) or \(\text{inr}@_\).

\((+)\) cannot be defined using the abbreviated data notation (which would be of the form data \((+) = \cdots\).
Disjoint Union in Agda (Cont.)

- Elimination is again represented by case distinction. So if we want to define for $A, B :: \text{Set}$ for instance:

$$f \ (c :: A + B) :: \text{Bool} = \{! !\}$$

we can type into the goal $c$ and choose menu “agda-case”.

Use of Concrete Disjoint Sets

- It is usually more convenient to define concrete disjoint unions directly with more intuitive names for constructors, e.g.

```agda
data Plant = tree(t :: Tree) | flower(f :: Flower)
```

- Now one can define for instance:

```agda
isFlower \ (p :: Plant) :: \text{Bool} = \text{case } p \text{ of}

(tree t) → \text{ff}

(flower f) → \text{tt}
```

Disjoint Union in Agda (Cont.)

- We obtain

$$f \ (c :: A + B) :: \text{Bool} = \text{case } c \text{ of}

(inl a) → \{! !\}

(inr b) → \{! !\}$$

and insert into the first goal e.g. true and the second one false.

(e) The $\Sigma$-Set

- The $\Sigma$-set is a second version of the dependent product of two sets.

- It depends on

  - a set $A$,

  - and a second set $B$ depending on $A$, i.e. on $B : A \rightarrow \text{Set}$.

- Similar to the standard product $(x : A) \times (B x)$.

- In Agda

  - $(x : A) \times (B x)$ is a built-in construct,

  - the $\Sigma$-set is introduced by the user using a constructor, similar to the previous sets.

  - The $\Sigma$-set behaves sometimes better than the standard product.
Rules for \( \Sigma \)

**Formation Rule**

\[
A : \text{Set} \quad B : A \rightarrow \text{Set} \\
\Sigma A B : \text{Set}
\]

**Introduction Rule**

\[
\begin{align*}
A : \text{Set} \\
B : A \rightarrow \text{Set} \\
a : A \\
b : B a \\
p A B a b : \Sigma A B
\end{align*}
\]

**Elimination Rule**

\[
\begin{align*}
A : \text{Set} \\
B : A \rightarrow \text{Set} \\
C : (\Sigma A B) \rightarrow \text{Set} \\
c : (a : A) \rightarrow (b : B a) \rightarrow C (p A B a b)) \\
d : \Sigma A B \\
\text{Case}_\Sigma A B C c d : C d
\end{align*}
\]

**Equality Rule**

\[
\text{Case}_\Sigma A B C c (p A B a b) = c a b : C (p A B a b)
\]

The \( \Sigma \)-Set using the Log. Framew.

- The more compact notation is:

\[
\begin{align*}
\Sigma : (A : \text{Set}) \\
&\rightarrow (A \rightarrow \text{Set}) \\
&\rightarrow \text{Set} . \\
p : (A : \text{Set}) \\
&\rightarrow (B : A \rightarrow \text{Set}) \\
&\rightarrow (a : A) \\
&\rightarrow (B a) \\
&\rightarrow \Sigma A B .
\end{align*}
\]
The $\Sigma$-Set and the Dep. Prod.

- Both the $\Sigma$-set and the dep. product have similar introduction rules.
- For the $\Sigma$-set, the constructors have additional arguments $A$, $B$ necessary for bureaucratic reasons only.
- One can define the projections $\pi_0$, $\pi_1$ using $\text{Case}_\Sigma$:
  $$\pi_0 = \text{Case}_\Sigma A B (\lambda x.A) (\lambda(y:B)x).x$$
  $$\pi_1 = \text{Case}_\Sigma A B (\lambda x.B \pi_0(x)) (\lambda(y:B)x).y$$
- On the other hand, from $\pi_0$, $\pi_1$ we can define $\text{Case}_\Sigma$ as follows:
  $$\lambda A, B, s, d. s \pi_0(d) \pi_1(d).$$

The $\Sigma$-Set in Agda

- $\Sigma$ can be defined as a “data”-set with a constructor, e.g.
  $$\text{Sigma} \quad (A :: \text{Set})$$
  $$\quad (B :: A \rightarrow \text{Set})$$
  $$\quad :: \quad \text{Set}$$
  $$\quad = \quad \text{data} \ p \ ((a :: A) \ (b :: Ba))$$
- Elimination uses $\text{case-distinction}$:
  $$f \quad (c :: \text{Sigma} \ A \ B)$$
  $$\quad :: \quad D$$
  $$\quad = \quad \text{case} \ c \ of$$
  $$\quad \quad (p \ a \ b) \quad \rightarrow \quad \cdots$$

The $\Sigma$-Set and the Dep. Prod.

- However the dependent product has the $\eta$-rule (which is however not implemented in Agda).
- Because of the lack of $\eta$-rule, $\Sigma$ works usually better than the dependent product in Agda.
- I personally don’t use the dependent product of Agda much.

The $\Sigma$-Set in Agda (Cont.)

- Again one usually defines concrete $\Sigma$-sets more directly.
- Example: Assume we have defined
  - a set Plant\_Group for groups of plants (e.g. “tree”, “flower”),
  - depending on $g :: \text{Plant\_Group}$, sets
    Plants\_in\_group $g$) for plants in that group.
- The set of plants can then be defined as
  $$\text{data} \ \text{Plant} = \text{plant} \ (g :: \text{Plant\_Group})(pg :: \text{Plants\_in\_group} \ g)$$
The Σ-Set in Agda (Cont.)

- Not surprisingly, for elimination we use case distinction, e.g.:

\[
\begin{align*}
f : (p :: \text{Plant}) \\
\quad :: \text{Plant}\_\text{group} \\
\quad = \text{case } p \text{ of} \\
\quad \quad (\text{plant } g \ pg) \rightarrow g
\end{align*}
\]

(f) Formulae in Dep. Type Theory

- We have seen how to represent atomic decidable formulae.
- Now treatment of complex formulae constructed using logical connectives.

Conjunction

- \(A \land B\) is true iff both \(A\) is true and \(B\) is true.
- Therefore a proof of \(A \land B\) consists of a proof of \(A\) and a proof of \(B\).
  - It is therefore a pair \(\langle p, q \rangle\) consisting of a proof \(p\) of \(A\) and a proof \(q\) of \(B\).
- Therefore the set of proofs of \(A \land B\) is the set of pairs of elements of \(A\) and \(B\), i.e. \(A \times B\).
- We can identify \(A \land B\) with \(A \times B\).

Conjunction (Cont.)

- With this identification, the introduction rule for the product allows to form a proof of \(A \land B\) from a proof of \(A\) and a proof of \(B\):

\[
\frac{p : A \quad q : B}{\langle p, q \rangle : A \land B}
\]

- This means that we can derive \(A \land B\) from \(A\) and \(B\).
Conjunction and Natural Ded.

In so called natural deduction, one has rules for deriving and eliminating formulas formed using the standard connectives.

There the rule for introducing proofs of $A \land B$ is

$$
\begin{array}{c}
A \\
\hline
A \land B
\end{array}
$$

The type theoretic introduction rule corresponds exactly to this rule.

Example 1

For instance, assume we want to prove that a function $\text{sort}$ from lists to lists is a sorting algorithm.

Then we have to show that for every list $l$ the application of $\text{sort}$ to $l$ is sorted, and has the same elements of $l$.

In order to show this, one would assume a list $l$ and show

- first that $\text{sort} \ l$ is sorted,
- then, that $\text{sort} \ l$ has the same elements as $l$,
- and finally conclude that it fulfills the conjunction of both properties.

The last operation uses the introduction rule for $\land$.

Conjunction (Cont.)

The elimination rule for $\land$ allows to project a proof of $A \land B$ to a proof of $A$ and a proof of $B$:

$$
\begin{array}{c}
p : A \land B \\
\hline
\pi_0(p) : A \\
\pi_1(p) : B
\end{array}
$$

This means that we can derive from $A \land B$ both $A$ and $B$.

This corresponds to the natural deduction elimination rule for $\land$:

$$
\begin{array}{c}
A \land B \\
\hline
A \\
B
\end{array}
$$

Example 2

Assume we have defined a function $f$, which takes a list of natural numbers $l$, a proof that $l$ is sorted, and a natural number $n$, and returns the Boolean value $\texttt{tt}$ or $\texttt{ff}$ indicating whether $n$ is in this list or not.

Assume now a sorting function $\text{sort}$ from lists of natural numbers to natural numbers, plus a proof that it is a sorting function, i.e. that $\text{sort} \ l$ is sorted and has the same elements as $l$ for every list $l$.

We want to apply $f$ to $\text{sort} \ l$ and need therefore a proof that $\text{sort} \ l$ is sorted.

We have that the conjunction of “$\text{sort} \ l$ is sorted” and “$\text{sort} \ l$ has the same elements as $l$” holds.

Using the elimination rule for $\land$ one can conclude the desired property, that $\text{sort} \ l$ is sorted.
Example 3

Assume a proof of $A \land B$.
We want to show $B \land A$.
By $\land$-elimination we obtain from $A \land B$ that $B$ holds.
Similarly we conclude that $A$ holds.
Using $\land$-introduction we conclude $B \land A$.
In natural deduction, this proof is as follows:

\[
\frac{A \land B}{B} \quad \frac{A \land B}{A} \\
\frac{B}{B \land A}
\]

Below we will see how to show this in Agda.

Disjunction

$A \lor B$ is true iff $A$ is true or $B$ is true.
Therefore a proof of $A \lor B$ consists of a proof of $A$ or a proof of $B$, plus the information which one.
It is therefore an element $\text{inl } p$ for a proof $p : A$ or an element $\text{inr } q$ for a proof $q : B$.
Therefore the set of proofs of $A \lor B$ is the disjoint union of $A$ and $B$, i.e. $A + B$.
We can identify $A \lor B$ with $A + B$.

Disjunction (Cont.)

With this identification, the introduction rules for $+$ allows to form a proof of $A \lor B$ from a proof of $A$ or from a proof of $B$.

$A : \text{Set} \quad B : \text{Set} \quad p : A$

\[
\frac{\text{inl } A \; B \; p : A + B}{p : A + B}
\]

$A : \text{Set} \quad B : \text{Set} \quad p : B$

\[
\frac{\text{inr } A \; B \; p : A + B}{p : A + B}
\]

Omitting the premises $A, B : \text{Set}$ and omitting them as arguments of $\text{inl}$ and $\text{inr}$ (which is needed only for bureaucratic reasons) we get:

\[
\frac{p : A}{\text{inl } p : A + B}
\]
\[
\frac{p : B}{\text{inr } p : A + B}
\]

Disjunction (Cont.)

This means that we can derive $A \lor B$ from $A$ and from $B$.
This is what is expressed by the natural deduction introduction rules for $\lor$:

\[
\frac{A}{A \lor B} \quad \frac{B}{A \lor B}
\]
Example 1

- Assume we want to show that every prime number is equal to 2 or odd.
- In order to show this one assumes a prime number.
  - If it is 2, it is trivially equal to 2.
  - Using the introduction rule for \( \lor \) one concludes that it is equal to 2 or odd.
  - Otherwise, one argues (using some proof) that it is odd.
  - Using the introduction rule for \( \lor \) one concludes again that it is equal to 2 or odd.

Disjunction (Cont.)

The **elimination rule** for \( + \) allows to form from an element of \( A + B \) an element of any set \( C \) provided we can compute such an element from \( A \) and from \( B \):

\[
\begin{align*}
A &: \text{Set} \\
B &: \text{Set} \\
C &: (A \lor B) \to \text{Set} \\
sl &: (a : A) \to C \ (\text{inl} \ A \ B \ a) \\
sr &: (b : B) \to C \ (\text{inr} \ A \ B \ b) \\
d &: A \lor B \\
\text{Case}_+ A B C sl sr d &: C d
\end{align*}
\]

Disjunction (Cont.)

Omitting the dependency of \( C \) on \( A \lor B \) and omitting the bureaucratic premises and arguments \( A, B \) and \( C \) we get:

\[
\begin{align*}
d &: A \lor B \\
sl &: A \to C \\
sr &: B \to C
\end{align*}
\]

This means that we can **derive from** \( A \lor B \) **a formula** \( C \), if we can derive \( C \) from \( A \) and from \( B \).

Disjunction (Cont.)

This is what is expressed by the **natural deduction elimination rules** for \( \lor \):

\[
\begin{align*}
A & \\
B & \\
\vdots & \\
A \lor B & \\
C & \\
C &
\end{align*}
\]

(Note that in the natural deduction elimination rule, from the premise “\( C \) derivable from \( A \)” we obtain “\( A \to C \)”, similarly for “\( C \) derivable from \( B \)” we get \( B \to C \).)
Example 2

- Assume we want to show that every prime number is equal to 2, equal to 3, or \( \geq 5 \).
- We want to make use of the proof above that every prime number is equal to 2 or odd.
- We assume a prime number.
  - We know that it is equal to 2 or odd.
  - In case it is equal to 2 we conclude that it is equal to 2, equal to 3, or \( \geq 5 \).
  - In case it is odd, we conclude using the fact that it is prime and 1 is not prime, that it is equal to 3 or \( \geq 5 \). Therefore it is equal to 2, equal to 3, or \( \geq 5 \).
  - Now from the elimination rule for \( \lor \) we conclude that the prime number chosen is equal to 2, equal to 3, or \( \geq 5 \).

Example 3

- Assume a proof of \( A \lor B \).
- We want to show \( B \lor A \).
  - We have \( A \lor B \).
  - From assumption \( A \) we obtain \( A \) and therefore by \( \lor \)-introduction \( B \lor A \).
  - From assumption \( B \) we obtain \( B \) and therefore by \( \lor \)-introduction \( B \lor A \).
  - By \( \lor \)-elimination we obtain from these three premises \( B \lor A \) without any premises.

Example 3 (Cont.)

- In natural deduction, this proof is as follows (we write \( A_1, \ldots, A_n \vdash B \) for \( B \) follows under assumptions \( A_1, \ldots, A_n \)):

\[
\begin{align*}
A \lor B & \quad \frac{A \vdash A}{B \lor A} \quad \frac{B \vdash B}{B \lor A}
\end{align*}
\]

- Below we will see how to show this in Agda.

Implication

- We write temporarily \( \supset \) for logical implication, in order to distinguish it from the function type \( \to \).
  - Below we see that \( \supset \) can be identified with \( \to \).
  - \( A \supset B \) is true iff, whenever \( A \) is true then \( B \) is true.
  - Therefore if there is a proof of \( A \), there must be a proof of \( B \).
  - Therefore a proof of \( A \supset B \) is a function, which takes a proof of \( A \) and computes a proof of \( B \).
  - Therefore the set of proofs of \( A \supset B \) is the function type \( A \to B \).
  - We can identify \( A \supset B \) with \( A \to B \).
Implication (Cont.)

With this identification, the **introduction rule for** \( \rightarrow \) allows to form a proof of \( A \supset B \) from a proof of \( B \) depending on a proof \( p \) of \( A \):

\[
\frac{p : A \Rightarrow q : B}{\lambda(p : A).q : A \supset B}
\]

This means that, if we, **from assumptions** \( p:A \) can prove \( B \)

(i.e. we can make use of a context \( p : A \) for proving \( q : B \))

then we can derive \( A \supset B \) without assuming \( p:A \).

Example

We extend the proof that from a proof of \( A \lor B \) it follows \( B \lor A \) to a proof of

\[
(A \lor B) \rightarrow (B \lor A)
\]

The previous proof can be easily transformed into a proof of \( A \lor B \vdash B \lor A \).

By \( \rightarrow \)-introduction, it follows \( (A \lor B) \rightarrow (B \lor A) \).

Implication (Cont.)

This is what is expressed by the **natural deduction introduction rule for** \( \supset \):

\[
\begin{array}{c}
A \\
\vdash \\
\vdash \\
B \\
\hline
A \supset B
\end{array}
\]

Example

The complete proof in natural deduction is as follows is as follows.

\[
\begin{array}{c}
A \lor B \vdash A \lor B \\
A \vdash A \\
B \vdash B \\
\hline
A \lor B \vdash A \lor B \\
B \vdash B \lor A \\
\hline
A \lor B \vdash B \lor A \\
\hline
(A \lor B) \rightarrow (B \lor A)
\end{array}
\]
Implication (Cont.)

- The elimination rule for \( \supset \) allows to apply a proof \( p \) of \( A \supset B \) to a proof of \( q \) of \( A \) in order to obtain a proof of \( B \):

\[
\begin{align*}
  p : A \supset B & \quad q : A \\
  \hline
  p \quad q : B
\end{align*}
\]

- This means that we can derive from \( A \supset B \) and \( A \) that \( B \) holds.

- This is what is expressed by the natural deduction elimination rule for \( \supset \):

\[
A \supset B \quad A \quad \vdash B
\]

Example

- A proof in natural deduction is as follows:

\[
\begin{align*}
A, A \rightarrow B \vdash A \rightarrow B & \quad A, A \rightarrow B \vdash A \\
\hline
A, A \rightarrow B \vdash B & \\
\hline
A \vdash (A \rightarrow B) \rightarrow B & \\
\hline
A \rightarrow (A \rightarrow B) \rightarrow B
\end{align*}
\]

Example

- Assume we want to show \( A \rightarrow (A \rightarrow B) \rightarrow B \).

- We can show this as follows:
  - From assumptions \( A \) and \( A \rightarrow B \) we can conclude \( A \rightarrow B \).
  - From assumptions \( A \) and \( A \rightarrow B \) we can conclude as well \( A \).
  - Using the elimination rule for \( \rightarrow \), we conclude that under the same assumptions we get \( B \).
  - Using the introduction rule for \( \rightarrow \) we conclude from assumption \( A \) that \( (A \rightarrow B) \rightarrow B \) holds.
  - Using again the introduction rule for \( \rightarrow \) we conclude that \( A \rightarrow (A \rightarrow B) \rightarrow B \) holds without any assumptions.

Negation

- \( \neg A \) has the same meaning as \( A \supset \bot \) (where \( \bot \) is absurdity or the set False):
  - If there is no proof of \( A \), then we can prove \( A \supset \bot \).
  - If from any proof of \( A \) we can create a proof of absurdity, then there cannot be a proof of \( A \), \( A \) must be false.

- Therefore we can identify \( \neg A \) with \( A \rightarrow \text{False} \).
Universal Quantification

Since we have many types, we have to write when using quantifiers explicitly the type, the bound variable is ranging over:

We write therefore \( \forall x : A . B \), \( \exists x : A . B \).

\( \forall x : A . B \) is true iff, for all \( x : A \) there exists a proof of \( B \) (with that \( x \)).

Therefore a proof of \( \forall x : A . B \) is a function, which takes an \( x : A \) and computes an element of \( B \).

Therefore the set of proofs of \( \forall x : A . B \) is the dependent function type \( (x : A) \rightarrow B \).

We can identify \( \forall x : A . B \) with \( (x : A) \rightarrow B \).

Universal Quantification (Cont.)

With this identification, the introduction rule for the dependent function type allows to form a proof of \( \forall x : A . B \) from a proof of \( B \) depending on an element \( x : A \):

\[
\frac{x : A \Rightarrow p : B}{\lambda(x : A) . p : (\forall x : A . B)}
\]

This means that, if we, from \( x : A \) can prove \( B \), then we get a proof of \( \forall x : A . B \) which doesn’t depend on \( x : A \).

Universal Quantification (Cont.)

This is what is expressed by the natural deduction introduction rule for \( \forall \):

\[
\frac{B}{\forall x : A . B}
\]

where

- \( x \) might not occur free in any assumption of the proof.
- This is guaranteed in type theory, since \( x : A \) must be the last element of the context, so any other assumptions must be located before it and can therefore not depend on \( x : A \).

Universal Quantification (Cont.)

The conclusion will no longer depend on free variables \( x \).
- This corresponds in type theory to the fact that \( x : A \) does no longer occur in the context of the conclusion.
Example

- Assume one wants to show that for every natural number \( n \), \( n + 0 == n \).
- In order to show this one assumes a natural number \( n \) and shows then that \( n + 0 == n \).
- then using the introduction rule for \( \forall \) one concludes \( \forall n : \mathbb{N}.n + 0 == n \).
- In natural deduction, this proof is as follows (where the prove of \( n + 0 == n \) is not carried out):

\[
\begin{align*}
\frac{n + 0 == n}{\forall n : \mathbb{N}.n + 0 == n}
\end{align*}
\]

Universal Quantification (Cont.)

- This is what is expressed by the natural deduction elimination rule for \( \forall \).
- For the simple languages used in natural deduction, there is no need to derive that \( a : A \); in more complex type theories we have to carry out this derivation.

\[
\begin{array}{c}
\forall x : A.B \\
a : A
\end{array}
\quad \Rightarrow
\begin{array}{c}
B[x := a]
\end{array}
\]

Example

- Assume a proof of \( \forall n : \mathbb{N}.0 + n == n \).
- We want to conclude that \( \forall n, m : \mathbb{N}.0 + (n + m) == (n + m) \).
- This can be done as follows:
  - One assumes \( n, m : \mathbb{N} \).
  - Then one can conclude \( n + m : \mathbb{N} \).
  - Using \( \forall n : \mathbb{N}.0 + n == n \) and the elimination rule for \( \forall \) one concludes \( 0 + (n + m) == (n + m) \) under assumption \( n, m : \mathbb{N} \).
  - Now using the introduction rule for \( \forall \) twice it follows \( \forall n, m : \mathbb{N}.0 + (n + m) == (n + m) \).

\[
\begin{array}{c}
\forall x : A.B \\
av : A
\end{array}
\quad \Rightarrow
\begin{array}{c}
B[x := a]
\end{array}
\]

Universal Quantification (Cont.)

- The elimination rule for the dependent function type allows to apply a proof \( p \) of \( \forall x : A.B \) to an element \( a : A \) in order to obtain a proof of \( B[x := a] \):

\[
\begin{array}{c}
p : (\forall x : A.B) \\
a : A
\end{array}
\quad \Rightarrow
\begin{array}{c}
p a : B[x := a]
\end{array}
\]

- This means that we can derive from \( \forall x : A.B \) and an element of \( a : A \) that \( B[x := a] \) holds.
Example

In natural deduction, this proof is written as follows:

$$\forall n : N. 0 + n == n \quad \frac{n : N, m : N \vdash n : N \quad n : N, m : N \vdash m : N}{n : N, m : N \vdash 0 + (n + m) == (n + m)}$$

Existential Quantification

- \(\exists x : A.B\) is true iff there exists an \(a : A\) such that \(B[x := a]\) is true.
- Therefore a proof of \(\exists x : A.B\) is a pair \(\langle a, p \rangle\) consisting of an element \(a : A\) and a proof \(p\) of \(B[x := a]\).
- Therefore the set of proofs of \(\exists x : A.B\) is the dependent product \((x : A) \times B\).
- We can identify \(\exists x : A.B\) with \((x : A) \times B\).

Example (Cont.)

- With this identification, the introduction rule for the dependent product allows to form a proof of \(\exists x : A.B\) from an element \(a : A\) and a proof \(p : B[x := a]\):

  $$\begin{align*}
  a : A & \quad p : B[x := a] \\
  \langle a, p \rangle \in (\exists x : A.B) &
  \end{align*}$$

- This is what is expressed by the natural deduction introduction rule for \(\exists\):

  $$\begin{align*}
  a : A & \quad B[x := a] \\
  \exists x : A.B &
  \end{align*}$$

Example

- Assume we want to show \(\forall n : N. \exists m : N. m > n\).
  - In order to prove this one assumes first \(n : N\).
  - Then one concludes \(S n : N\) and \(S n > n\).
  - Using the introduction rule for \(\exists\) one concludes \(\exists m : N. m > n\) under the assumption \(n : N\).
  - Using the introduction rule for \(\forall\) one concludes \(\forall n : N. \exists m : N. m > n\).
Example

In natural deduction, this proof reads as follows:

\[
\begin{align*}
\frac{n : N \vdash n : N}{n : N \vdash S\ n : N} & \quad \frac{n : N \vdash S\ n > n}{\forall n : N . \exists m : N . m > n}
\end{align*}
\]

Therefore the rule in natural deduction follows from the type theoretic rules:

\[
\begin{align*}
x : A & \\
B & \\
\vdots & \\
\vdots & \\
\exists x : A . B & \quad C
\end{align*}
\]

where the conclusion does not depend on \( x : A \) and \( B \).

Existential Quantification (Cont.)

The elimination rule for the dependent product allows to project a proof \( p \) of \( \exists x : A . B \) to an element \( \pi_0(p) : A \) and proof \( \pi_1(p) : B[x := \pi_0(p)] \).

This kind of rule works only if we have explicit proofs.

From this we can derive a rule which is essentially that used in natural deduction (in which one doesn't have explicit proofs):

Assume:
\- \( C \) : Set, which does not depend on \( x : A \),
\- \( p \) : (\( \exists x : A . B \)) and
\- \( x : A , y : B \Rightarrow c : C \).
\- Then we have \( c[x := \pi_0(p) , y := \pi_1(p)] : C \), not depending on \( x : A \) or \( y : B \).

Therefore the rule in natural deduction follows from the type theoretic rules:

\[
\begin{align*}
x : A & \\
B & \\
\vdots & \\
\vdots & \\
\exists x : A . B & \quad C
\end{align*}
\]

where the conclusion does not depend on \( x : A \) and \( B \).

Example

Assume we have shown
\( \forall n : N . \exists m : N . m > n \land \text{Prime}(m) \).

We want to show that for all \( n \) there exist two primes above it, i.e.
\( \forall n : N . \exists m : N . \exists k : N . m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \).

We can derive this as follows:

Assume \( n : N \).
We have \( \exists m : N . m > n \land \text{Prime}(m) \).
So assume \( m : N \) and \( m > n \land \text{Prime}(m) \).
We have as well \( \exists k : N . k > m \land \text{Prime}(k) \).
So assume \( k : N \) and \( k > m \land \text{Prime}(k) \).
Example

Then we can conclude

\[ m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

and therefore as well

\[ \exists m, k : \mathbb{N}. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

Now by \( \exists \)-elimination twice follows

\[ n : \mathbb{N} \vdash \exists m, k : \mathbb{N}. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

without assuming \( m, k \) as above.

By \( \forall \)-introduction follows

\[ \forall n : \mathbb{N}. \exists m, k : \mathbb{N}. m > k \land k > n \land \text{Prime}(m) \land \text{Prime}(k) \]

Example

First step: Under the global assumption

\[ n : \mathbb{N}, m : \mathbb{N}, m > n \land \text{Prime}(m), k : \mathbb{N}, k > m \land \text{Prime}(k) \]

we prove the following

\[ m : \mathbb{N} \quad \exists k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \]

\[ \frac{m : \mathbb{N} \quad \exists k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)}{\exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)} \]

So we have shown

\[ n : \mathbb{N}, m : \mathbb{N}, m > n \land \text{Prime}(m), k : \mathbb{N}, k > m \land \text{Prime}(k) \vdash \exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \]

Example

Second step: Under the assumption

\[ n : \mathbb{N}, m : \mathbb{N}, m > n \land \text{Prime}(m) \]

we can conclude

\[ \exists k : \mathbb{N}. k > m \land \text{Prime}(k) \]

and then conclude by \( \exists \)-elimination and Step 1

\[ \frac{\exists k : \mathbb{N}. k > m \land \text{Prime}(k)}{\exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)} \]

\[ \frac{\exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)}{\exists m, k : \mathbb{N}. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k)} \]
Example

Third step: Again we can conclude
\[ n : N \vdash \exists m : N. m > n \land \text{Prime}(m) \]
and then conclude by \( \exists \)-elimination and Step 2
\[ n : N \vdash \exists m : N. m > n \land \text{Prime}(m) \]
\[ n : N \vdash \exists m, k : N. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \]
\[ \forall n : N. \exists m, k : N. m > n \land k > m \land \text{Prime}(m) \land \text{Prime}(k) \]

Constructive Logic (Cont.)

We can derive as well a function which depending on
\[ p : A + B \text{ decides whether } p = \text{inl}(a) \text{ or } p = \text{inr}(b) \].
Therefore we can decide, from a proof of a disjunction,
which of the disjuncts holds.

Now:
- Any function in type theory is recursive.
- We cannot decide the Turing Halting problem, i.e.
we cannot decide for a Turing machine whether it
halts or not.
- Therefore we cannot prove in type theory
\[ \forall x : \text{Turing\_Machine}. (x \text{ halts} \lor \neg (x \text{ halts})) \]

Constructive Logic (Cont.)

In classical logic we can prove the above, since we
can derive \( A \lor \neg A \) (tertium non datur) for any formula \( A \).

In type theory, this law cannot hold, unless we don’t
want that all programs can be evaluated.
- The logic of type theory is intuitionistic
  (constructive) logic, in which \( A \lor \neg A \) and \( \neg \neg A \rightarrow A \)
don’t hold for all formulae \( A \).
Constructive Logic (Cont.)

In classical logic,

- $\exists x : A.B$ is equivalent to $\neg \forall x : A.\neg B$,
- $A \lor B$ is equivalent to $\neg (\neg A \land \neg B)$.

If we take decidable atomic formulae only and replace $\exists x : A.B$ and $A \lor B$ by the above formulae, then all formulae provable in classical logic are derivable.

This requires $(\neg \neg A) \rightarrow A$, which can be shown for all formulae built from decidable atomic formulae using $\neg, \rightarrow, \land, \lor$.

The formula $A \lor \neg A$ translates into $\neg (\neg A \land \neg \neg A)$, which trivially holds, since $\neg A$ and $\neg \neg A$ implies $\bot$.

In this sense, type theory contains classical logic, but is richer, since it has as well so called strong disjunction and existential quantification.

Constructive Logic (Cont.)

Proof (using classical logic) of

$\exists x : A.B \leftrightarrow (\neg \forall x : A.\neg B) :$

We have classically:

$\neg \neg A \rightarrow A :$

If $A$ is true, then $\neg \neg A \rightarrow A$ holds.
If $A$ is false, then $\neg \neg A$ is false, therefore $\neg \neg A \rightarrow A$ holds.

We show intuitionistically

$(\neg \exists x : A.B) \leftrightarrow (\forall x : A.\neg B) :$

Assume $\neg \exists x : A.B$, $x : A$ and show $\neg B$.
If we had $B$, then we had $\exists x : A.B$, contradicting $\neg \exists x : A.B$. Therefore $\neg B$.

Assume $\forall x : A.\neg B$. Show $\neg \exists x : A.B$:
Assume $\exists x : A.B$. Assume $x$ s.t. $B$ holds.
By $\forall x : A.\neg B$ we get $\neg B$, therefore a contradiction.

Now it follows (classically):

$(\exists x : A.B) \leftrightarrow (\neg \exists x : A.B) \leftrightarrow (\neg \forall x : A.\neg B)$

Constructive Logic (Cont.)

Proof of

$A \lor B \leftrightarrow \neg (\neg A \land \neg B) :$

We show intuitionistically

$(\neg A \lor B) \leftrightarrow (\neg A \land \neg B) :$

Assume $(\neg A \lor B)$. If $A$ then $A \lor B$, a contradiction, therefore $\neg A$.
Similarly we get $\neg B$, therefore $\neg A \land \neg B$.
Assume $\neg A \land \neg B$, show $(A \lor B)$.
Assume $A \lor B$. If $A$ then a contradiction with $\neg A$, similarly with $B$.

Now it follows (classically):

$(A \lor B) \leftrightarrow \neg (\neg A \lor B) \leftrightarrow \neg (\neg A \land \neg B)$
Constructive Logic (Cont.)

- **Weak disjunction and existential quantification** is expressed by the formulae \( \neg(\neg A \land \neg B) \) and \( \neg\forall x : A. \neg B \).
- When using only weak disjunction, existential quantification and decidable atomic formulae, we obtain classical logic.
- **Strong disjunction and existential quantification** is expressed by the original type theoretic formulae.

The Logical Connectives in Agda

Implication is represented by \( \rightarrow \) in Agda.

We can introduce the formula (or set) expressing \( A \rightarrow (A \rightarrow B) \rightarrow B \) as follows:

\[
\text{Lemma1 :: Set} = A \rightarrow (A \rightarrow B) \rightarrow B
\]

A proof of Lemma1 is an element lemma1 of it.

So in order to prove Lemma1 we make the following goal:

\[
\text{lemma1 :: Lemma1} = \{ ! ! \}
\]

One obtains

\[
\text{lemma1 :: Lemma1} = \lambda(a :: A) \rightarrow \lambda(ab :: A \rightarrow B) \rightarrow \{ ! ! \}
\]

Lemma1 was \( A \rightarrow (A \rightarrow B) \rightarrow B \),

we have abstracted from \( A \) and \( A \rightarrow B \),

so the type of the goal is the conclusion of the implication, namely \( B \).

The Logical Connectives in Agda in Agda
The Logical Connectives in Agda

\[ \text{lemma1} :: \text{Lemma1} \]
\[ = \lambda(a :: A) \to \lambda(ab :: A \to B) \to \{! \!\! \! \!\} \]

Type of goal is \( B \)
- At the position of the goal we have context \( a :: A \) and \( ab :: A \to B \), because we have \( \lambda \)-abstracted those variables.
- Can be checked by using goal-menu context.
- We can take \( ab :: A \to B \) and apply it to \( a :: A \) in order to obtain \( ab \ a :: B \), which solves the goal.

Implication in Agda
- Note that in the previous example
  - \( ab \) is an element of the function type \( A \to B \).
  - \( a \) is an element of \( A \)
  - therefore \( ab \ a \) is an element of \( B \),
  - therefore the typing is correct.

Interactive Theorem Proving, CS 336, Lentterm 2004, Sec. 3(f)

The Logical Connectives in Agda
- We obtain the following proof:
  \[ \text{lemma1} :: \text{Lemma1} \]
  \[ = \lambda(a :: A) \to \lambda(ab :: A \to B) \to ab \ a \]
- This is exactly the same as introducing a \( \lambda \)-term of type \( A \to (A \to B) \to B \).
- See exampleproofprologic1.agda

Implication in Agda
- As for \( \lambda \)-terms, the following is equivalent to the definition of lemma1 above, and therefore an equivalent proof of \( A \to (A \to B) \to B \):
  \[ \text{lemma1} (a :: A) \]
  \[ (ab :: A \to B) \]
  \[ :: B \]
  \[ = ab \ a \]

See exampleproofprologic2.agda
Conjunction in Agda

- Conjunction is represented as a product.
- There are two products in Agda, therefore as well two ways of representing conjunction:
  - One using the logical framework product:

\[
\text{AND1} \ (A, B :: \text{Set})
\]
\[
:: \text{Set}
\]
\[
= \ \text{sig}
\]
\[
\text{and1} :: A
\]
\[
\text{and2} :: B
\]

Interactive Theorem Proving, CS_336, Lentterm 2004, Sec. 3(f)

Conjunction in Agda

- One can write as well $\land$ for one of the versions of conjunction, and use it infix.
- We write on slides $\land$ for it, and get therefore:

\[
(A, B :: \text{Set})
\]
\[
:: \text{Set}
\]
\[
= \ \text{data and}(a :: A)(b :: B)
\]
\[
\text{AB} :: \text{Set}
\]
\[
= A \land B
\]

Interactive Theorem Proving, CS_336, Lentterm 2004, Sec. 3(f)

Example (Conjunction)

- We prove $(A \land B) \to (B \land A)$ using both versions of the conjunction (see exampleproofproplogic6.agda):

\[
\text{lemma2a} \ (ab :: \text{AND1} \ A \ B)
\]
\[
:: \text{AND1} \ B \ A
\]
\[
= \ \text{struct}
\]
\[
\text{and1} = ab.\text{and2}
\]
\[
\text{and2} = ab.\text{and1}
\]

\[
\text{lemma2b} \ (ab :: \text{AND2} \ A \ B)
\]
\[
:: \text{AND2} \ B \ A
\]
\[
= \ \text{case ab of}
\]
\[
\text{(and a b) } \to \ \text{and a b}
\]

Interactive Theorem Proving, CS_336, Lentterm 2004, Sec. 3(f)
Conjunction with more Args.

If one has a conjunction with more than one conjunctions, e.g. $A \land B \land C$, one can always express it using the binary $\land$:

As $(A \land B) \land C$ or $A \land (B \land C)$.

But it is often more convenient to use a trinary version of conjunction in one of the two versions.

Similarly one can introduce conjunctions of 4 or more conjuncts.

Disjunction in Agda

Or is represented as disjoint union in type theory.

In Agda we can write $\lor$ for it (on slides we write $\lor$) and define it as follows:

$$\lor (A, B :: \text{Set}) :: \text{Set} = \text{data or1}(a :: A) | \text{or2}(b :: B)$$

See `exampleproofprologic7.agda`.

On the blackboard $A \to A \lor B$ and $A \lor A \to A$ will now be shown in Agda.

Example (Disjunction)

The following derives $(A \lor B) \to (B \lor A)$:

$$\text{lemma3 } (ab :: A \lor B) :: B \lor A = \text{case } ab \text{ of}$$

$$\begin{align*}
\text{or1 } a & \to \text{or2@ } a \\
\text{or2 } b & \to \text{or1@ } b
\end{align*}$$

See `exampleproofprologic9.agda`.

Conjunction with more Args.

AND3a  $(A,B,C :: \text{Set})$

:: Set

= sig

and1 :: A

and2 :: B

and3 :: C

AND3b  $(A,B,C :: \text{Set})$

:: Set

= data and3 (a :: A) (b :: B) (c :: C)

See `exampleproofprologic5.agda`
Disjunction with more Args.

As for the conjunction, it is useful to introduce special ternary versions of the disjunction (and versions with higher arities):

\[
\text{OR3 } (A, B, C :: \text{Set})
:: \text{Set}
= \text{data or1 } (a :: A) \mid \text{or2 } (b :: B) \mid \text{or3 } (c :: C)
\]

See exampleproofprologic8.agda.

Example (∨, Cont.)

- \(ff < tt\) should be true, therefore as a formula equivalent to True, this is obtained by defining it as True.
- \(tt < ff\) should be false, therefore as a formula equivalent to False, this is obtained by defining it as False.
- Similarly \(tt < tt\) and \(ff < ff\) will be defined as False.

∀ in Agda

∀\(x : A.B\) is represented by \((x : A) \to B\) in Agda.

As an example,

- we define a ≺-operation on Bool using \(ff < tt\)
- and show ∀\(x : \text{Bool} \cdot \neg(x < x)\).

See exampleproofprologic10.agda.

Example (∨, Cont.)

First \(\prec\) is defined as follows:

\[
\prec(\,(a, b :: \text{Bool})
:: \text{Set}
= \text{case a of}
\quad (tt) \to \text{False}
\quad (ff) \to \text{case b of}
\quad (tt) \to \text{True}
\quad (ff) \to \text{False}
\]

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 3(f)
We introduce Not $A$:

\[
\text{Not} \quad (A :: \text{Set}) \\
:: \text{Set} \\
= \quad A \rightarrow \text{False}
\]

The statement that $<$ is antireflexive is

\[
\forall a : \text{Bool}. \neg(a < a)
\]

which is represented in Agda as follows:

\[
\text{Lemma4} :: \text{Set} \\
= (a :: \text{Bool}) \rightarrow \text{Not}(a < a)
\]

We want to prove Lemma4.

A proof of Lemma4 will be an element

\[
\text{lemma4} :: \text{Lemma4}
\]

So we have to solve the following goal:

\[
\text{lemma4} :: \text{Lemma4} \\
= \{! !\}
\]

The type of the goal is

\[
\text{Lemma4} = (a :: \text{Bool}) \rightarrow (a < a) \rightarrow \text{False}
\]

Since $\text{Not} \ (a < a) = (a < a) \rightarrow \text{False}$, we have

\[
\text{Lemma4} = (a :: \text{Bool}) \rightarrow \text{Not} (a < a) \\
= (a :: \text{Bool}) \rightarrow (a < a) \rightarrow \text{False}
\]

An element of $(a :: \text{Bool}) \rightarrow (a < a) \rightarrow \text{False}$ can be introduced by $\lambda$-abstracting $\lambda(a :: \text{Bool})$ and $\lambda(aa :: (a < a))$:

\[
\text{lemma4} :: \text{Lemma4} \\
= \lambda(a :: \text{Bool}) \rightarrow \\
\lambda(aa :: a < a) \rightarrow \\
\{! !\}
\]

The type of goal is now the conclusion of

$(a :: \text{Bool}) \rightarrow (a < a) \rightarrow \text{False}$, namely False.
Example (\(\forall\), Cont.)

\[
\text{lemma4 :: Lemma4} = \lambda(a :: \text{Bool}) \rightarrow \\
\lambda(aa :: a < a) \rightarrow \\
\{! \}
\]

Type of goal is \(\text{False}\).

- We need to make use of our assumptions, namely 
  \(a :: \text{Bool}\) and \(aa :: a < a\).
- \(a < b\) is defined by case disjinction on \(a\) and \(b\).  
- Unless we know that \(a = \text{tt}\) or \(a = \text{ff}\), we don’t know much about \(a < a\).
- So it seems to be a good step to make case disjinction on \(a\).

Example (\(\forall\), Cont.)

\[
\text{lemma4 :: Lemma4} = \lambda(a :: \text{Bool}) \rightarrow \\
\lambda(aa :: a < a) \rightarrow \\
\text{case } a \text{ of} \\
\text{(tt) } \rightarrow \{! \} \\
\text{(ff) } \rightarrow \{! \}
\]

However, we know now more about the assumptions \(aa :: a < a\).

- In case of \(a = \text{tt}\), we have \(a < a = (\text{tt} < \text{tt}) = \text{False}\)
- In case of \(a = \text{ff}\), we have \(a < a = (\text{ff} < \text{ff}) = \text{False}\)

The type of both goals is the same as before, namely \(\text{False}\), since it wasn’t dependent on \(a\).

Example (\(\forall\), Cont.)

\[
\text{lemma4 :: Lemma4} = \lambda(a :: \text{Bool}) \rightarrow \\
\lambda(aa :: a < a) \rightarrow \\
\text{case } a \text{ of} \\
\text{(tt) } \rightarrow \{! \} \\
\text{(ff) } \rightarrow \{! \}
\]

Since in both goals we have \(a :: (a < a) = \text{False}\), we can make case disjinction on \(aa\), which is the empty case disjinction.
Example (∀, Cont.)

We finish our proof as follows:

\[
\text{lemma4} :: \text{Lemma4} = \lambda (a :: \text{Bool}) \to \\
\lambda (aa :: a < a) \to \\
\text{case } a \text{ of} \\
\quad (\text{tt}) \to \text{case } aa \text{ of } \{ \}
\quad (\text{ff}) \to \text{case } aa \text{ of } \{ \}
\]

Example (∃, Cont.)

∃ in Agda

∃x : A.B is represented by one of the two dependent products in Agda.

Using meaningful names, we can define ∃x : A.B as follows:

\[
\text{Version1} :: \text{Set} = \text{sig} \\
\quad a :: A \\
\quad b :: B[x := a]
\]

\[
\text{Version2} :: \text{Set} = \text{data exists } (a :: A)(b :: B[x := a])
\]

Example (∀, Cont.)

In the previous example,
- the type of goal was False,
- and aa : False.

So we could instead of using the empty case distinction directly inserted aa in those goals:

\[
\text{lemma4a} :: \text{Lemma4} = \lambda (a :: \text{Bool}) \to \\
\lambda (aa :: a < a) \to \\
\text{case } a \text{ of} \\
\quad (\text{tt}) \to aa \\
\quad (\text{ff}) \to aa
\]

∃ in Agda

Above B[x := a] is the result of substituting in B for x the variable a.
Example ( )))

- As an example,
  - we define negation \( \neg \) on \( \text{Bool} \),
  - define an equality (\( \text{==} \)) on \( \text{Bool} \),
  - and show \( \forall a : \text{Bool}. \exists b : \text{Bool}. a \text{==} \neg b \).
- See exampleproofpropllogic11.agda.

Example ( )))

First (\( \text{==} \)) is defined as follows:

\[
(\text{==}) \quad (a, b :: \text{Bool}) \\
:: \text{Set} \\
= \text{case } a \text{ of} \\
(tt) \rightarrow \text{case } b \text{ of} \\
\quad \quad (tt) \rightarrow \text{True} \\
\quad \quad (ff) \rightarrow \text{False} \\
(ff) \rightarrow \text{case } b \text{ of} \\
\quad \quad (tt) \rightarrow \text{False} \\
\quad \quad (ff) \rightarrow \text{True}
\]

Example ( )))

\( \neg \) is defined as follows:

\[
\neg \quad (a :: \text{Bool}) \\
:: \text{Bool} \\
= \text{case } a \text{ of} \\
(tt) \rightarrow \text{ff} \\
(ff) \rightarrow \text{tt}
\]

Example ( )))

In order to introduce the statement, we introduce first the formula \( \exists b : \text{Bool}. a \text{==} \neg b \) depending on \( a : \text{Bool} \):

\[
\text{Lemma5aux} \quad (a :: \text{Bool}) \\
:: \text{Set} \\
= \text{sig} \\
\quad \quad b :: \text{Bool} \\
\quad \quad ab :: a \text{==} \neg b
\]

The statement \( \forall a : \text{Bool}. \exists b : \text{Bool}. a \text{==} \neg b \) is now as follows:

\[
\text{Lemma5} \quad :: \text{Set} \\
= (a :: \text{Bool}) \rightarrow \text{Lemma5aux } a
\]
Example (Ⅲ, Cont.)

- A proof of Lemma5 is an element
  \[
  \text{lemma5} :: \text{Lemma5}
  \]
  and we get the goal
  \[
  \text{lemma5} :: \text{Lemma5} = \{! !\}
  \]

- The type of goal is
  \[
  \text{Lemma5} = (a :: \text{Bool}) \rightarrow \text{Lemma5aux} \ a
  \]

- Any goal of function type is usually best solved by using \(\lambda\)-abstracting.

---

Example (Ⅲ, Cont.)

\[
\text{lemma5} :: \text{Lemma5} = \lambda(a :: \text{Bool}) \rightarrow \{! !\}
\]

Type of goal is
\[
\text{sig} \\
  b :: \text{Bool} \\
  ab :: a == \text{neg} \ b
\]

- We cannot show this goal universally for all \(a\) directly.
  - We have to provide a different \(b\) depending on whether \(a = \text{tt}\) or \(a = \text{ff}\).
  - So we need to make case distinction on \(a\).

---

Example (Ⅲ, Cont.)

We get

\[
\text{lemma5} :: \text{Lemma5} = \lambda(a :: \text{Bool}) \rightarrow \text{case} \ a \ 	ext{of} \\
  (\text{tt}) \rightarrow \{! !\} \\
  (\text{ff}) \rightarrow \{! !\}
\]
Example (9, Cont.)

\[\text{lemma5 :: Lemma5} = \lambda(a :: \text{Bool}) \to \text{case } a \text{ of } \{(\text{tt}) \to \{! !\} \} \]

- In case of \(a = \text{tt}\), the type of goal is

\[\text{Lemma5aux tt} = \text{sig}\{b :: \text{Bool}; ab :: tt == \text{neg b}\}\]

- So we can use goal menu intro and obtain:

Example (9, Cont.)

\[\text{lemma5 :: Lemma5} = \lambda(a :: \text{Bool}) \to \text{case } a \text{ of } \{(\text{tt}) \to \text{struct } b = \{! !\}, ab = \{! !\} \}
\]

- The first goal can be solved by setting \(b := \text{ff}\).

- Then the type of the second goal is

\[(\text{tt} == \text{neg b}) = (\text{tt} == \text{neg ff})
= (\text{tt} == \text{tt})
= \text{True}\]

which can be solved by setting \(ab := \text{true}\).

Example (9, Cont.)

So we get:

\[\text{lemma5 :: Lemma5} = \lambda(a :: \text{Bool}) \to \text{case } a \text{ of } \{(\text{tt}) \to \text{struct } b = \text{ff}, ab = \text{true} \}
\]

- In case \(a = \text{ff}\), we use again intro and obtain:

Example (9, Cont.)

\[\text{lemma5 :: Lemma5} = \lambda(a :: \text{Bool}) \to \text{case } a \text{ of } \{(\text{tt}) \to \text{struct } b = \{! !\}, ab = \{! !\} \}
\]

- The case \(a = \text{ff}\) can be solved in a similar way by setting \(b = \text{tt}, ab = \text{true}\).
Example (3, Cont.)

The resulting proof is as follows:

\[
\text{lemma5} :: \text{Lemma5} = \lambda(a :: \text{Bool}) \to \text{case } a \text{ of} \\
\quad (tt) \to \text{struct} \\
\quad \quad b = \text{ff} \\
\quad \quad ab = \text{true} \\
\quad (ff) \to \text{struct} \\
\quad \quad b = \text{tt} \\
\quad \quad ab = \text{true}
\]

Complex Example

We assume \( A, B : \text{Set} \) and equality relations on \( A, B \):

\[
\begin{align*}
\text{postulate } A & :: \text{Set} \\
\text{postulate } \text{EqA} & :: A \to A \to \text{Set} \\
\text{postulate } B & :: \text{Set} \\
\text{postulate } \text{EqB} & :: B \to B \to \text{Set}
\end{align*}
\]

We will introduce

\[ \begin{align*}
\text{the disjoint union } AB \text{ of } A \text{ and } B \\
\text{an equality } \text{EqAB} \text{ on } AB \\
\text{and show that if } \text{EqA} \text{ and } \text{EqB} \text{ are symmetric, so is } \text{EqAB.}
\end{align*} \]

See \text{exampleDisjointUnionEqual.agda}.

Equality Sets

- \( \text{EqA} \) (and \( \text{EqB} \)) could be decidable equalities,
  - i.e. \( \text{EqA} = \lambda(a, b :: A) \to \text{atom (eqboolA } a \ b) \),
    where \( \text{eqboolA} :: A \to A \to \text{Bool} \),
  - Or an undecidable equality.
    - E.g. the equality on \( \mathbb{N} \to \mathbb{N} \) is in standard logic
      \[
      f = g : \Leftrightarrow \forall n : \mathbb{N}. f(n) = g(n)
      \]
      which reads in Agda as follows:
      \[
      \text{EqN_N } (f, g :: \mathbb{N} \to \mathbb{N}) \\
      :: \text{Set} \\
      = (n :: \mathbb{N}) \to f \ n \ == g \ n
      \]
      where \( == \) is the equality on \( \mathbb{N} \).

Undecidable Equalities

- The last equality is undecidable, since in order to check whether \( \text{EqN_N } f \ g \) holds we have to check \text{for all } n : \mathbb{N} \text{ whether } f \ n = g \ n \) holds
The formation of $A + B$ is straightforward:

\[(+) \ (A, B :: \text{Set})\]
\[:: \text{Set} \]
\[= \ \text{data inl}(a :: A) \mid \text{inr}(b :: B)\]

We define the equality $EqAB$ on $A + B$ as follows:

- Assume $ab, ab' : A + B$.
- If one is of the form $\text{inl} \ a$ and the other of the form $\text{inr} \ b$, then $EqAB \ ab \ ab'$ should be false, so we define

  \[EqAB \ (\text{inl} \ a) \ (\text{inr} \ b) = EqAB \ (\text{inr} \ a) \ (\text{inl} \ a) = \text{False} .\]

- If they are of the form $\text{inl} \ a$ and $\text{inl} \ a'$, respectively, then $EqAB \ ab \ ab'$ should be true if $EqA \ a \ a'$ holds.

  This can be achieved by defining

  \[EqAB \ (\text{inl} \ a) \ (\text{inl} \ a') = EqA \ a \ a' .\]

If they are of the form $\text{inr} \ b$ and $\text{inr} \ b'$, respectively, then $EqAB \ ab \ ab'$ should be true if $EqB \ b \ b'$ holds.

This can be achieved by defining in this case

\[EqAB \ (\text{inr} \ b) \ (\text{inr} \ b') = EqB \ b \ b' .\]

The above equations will be definitional equalities, i.e. for instance $EqAB \ (\text{inl} \ a) \ (\text{inl} \ a')$ will rewrite to $EqA \ a \ a'$.

The definition of $EqAB$ is as follows:

\[EqAB :: (A + B) \to (A + B) \to \text{Set}\]
\[= \ \lambda(ab, ab' :: A + B) \to \]
\[\text{case} \ ab \ of\]
\[\left(\begin{array}{l}
\text{inl} \ a \ \to \ \text{case} \ ab' \ of \\
\quad \begin{array}{l}
\text{inl} \ a' \ \to \ EqA \ a \ a' \\
\text{inr} \ b' \ \to \ False
\end{array} \\
\text{inr} \ b \ \to \ \text{case} \ ab' \ of \\
\quad \begin{array}{l}
\text{inl} \ a' \ \to \ False \\
\text{inr} \ b' \ \to \ EqB \ b \ b'
\end{array}
\end{array}\right) .\]
The following code is equivalent to the previous code:

```
EqAB (ab, ab' :: A + B)
:: Set
= case ab of
  (inl a)  -> case ab' of
    (inl a')  -> EqA a a'
    (inr b')  -> False
  (inr b)  -> case ab' of
    (inl a')  -> False
    (inr b')  -> EqB b b'
```

We introduce the formulae expressing that EqA, EqB, EqAB are symmetric:

**Symmetry of EqA** is the formula

\[ \forall a, a' : A.\ EqA a a' \rightarrow EqA a' a \]

which translates as follows:

```
SymA :: Set
= (a, a' :: A) -> EqA a a' -> EqA a' a
```

The others are similar:

```
SymB :: Set
= (b, b' :: B) -> EqB b b' -> EqB b' b
SymAB :: Set
= (ab, ab' :: A + B) -> EqAB ab ab' -> EqAB ab' ab
```

Note that **SymA** is the statement expressing that EqA is symmetric.

- It is not a proof that EqA is symmetric.
- We can define SymA independently of whether EqA is symmetric or not.
- A proof that EqA is symmetric is an element of SymA, i.e. a term symA s.t.

\[ \text{symA :: SymA} \]

Note that we don’t have to show that SymA holds.

- We have to show that if SymA and SymB hold, then SymAB holds.
Complex Example

What we want to show is that $\text{SymA}$ and $\text{SymB}$ implies $\text{SymAB}$.

So we need to solve

$$\text{symAB} :: \text{SymA} \rightarrow \text{SymB} \rightarrow \text{SymAB}$$

$$= \{! !\}$$

As pointed out before, it is equivalent and more convenient to define $\text{symAB}$ as follows:

$$\text{symAB} (\text{symA} :: \text{SymA})$$
$$\hspace{2em} (\text{symB} :: \text{SymB})$$
$$:: \text{SymAB}$$
$$= \{! !\}$$

The type of the goal is $\text{symAB}$ which is

$$(ab, ab' :: A + B) \rightarrow \text{EqAB} ab ab' \rightarrow \text{EqAB} ab' ab.$$ 

An element of this type can be introduced by a $\lambda$-term, and using agda-goal-menu “intro” results in the code on the next slide.

---

Complex Example

$$\text{symAB} (\text{symA} :: \text{SymA})$$
$$\hspace{2em} (\text{symB} :: \text{SymB})$$
$$:: \text{SymAB}$$
$$= \lambda(ab, ab' :: A + B) \rightarrow \lambda(abab' :: \text{EqAB} ab ab') \rightarrow \{! !\}$$

The type of the goal is now $\text{EqAB} ab' ab$.

We make case distinction on $ab$ and $ab'$ and obtain the following:
Complex Example (Cont.)

In case $ab = \text{inl } a$ and $ab' = \text{inl } a'$ we

- have to show
  \[
  \text{EqAB } ab' \text{ ab , which is equal to EqA } a' a
  \]
- and have as assumption
  \[
  abab' :: \text{EqAB } ab \text{ ab' , which is equal to EqA a a'.}
  \]

So we have to derive from $abab' : \text{EqA a a'}$ an element of $\text{EqA a a'}$.

We have

\[
\text{symA} : (a, a' :: A) \rightarrow \text{EqA a a'} \rightarrow \text{EqA a a'}.
\]

Therefore we can apply $\text{symA}$ to $a, a'$ and $abab'$.

Complex Example (Cont.)

(Case $ab = \text{inr } b, ab' = \text{inr } b'$)

- We obtain

\[
\text{symA} a a' abab' : \text{EqA a a'}
\]

Since $\text{EqA a a'} = \text{EqAB ab' ab}$ we get as well

\[
\text{symA} a a' abab' : \text{EqAB ab' ab}
\]

and can use this to solve our first goal.

Complex Example (Cont.)

\[
\begin{align*}
\text{symAB} & (\text{symA :: SymA}) \\
& (\text{symB :: SymB}) \\
& :: \text{SymAB} \\
& = \lambda (ab, ab' :: A + B) \rightarrow \\
& \lambda (abab' :: \text{EqAB ab ab'}) \rightarrow \\
& \text{case } ab \text{ of} \\
& \hspace{1em} (\text{inl } a) \rightarrow \text{case } ab' \text{ of} \\
& \hspace{2em} (\text{inl } a') \rightarrow \text{symA a a' abab'} \\
& \hspace{2em} (\text{inr } b') \rightarrow \{! \} \\
& \hspace{1em} (\text{inr } b) \rightarrow \text{case } ab' \text{ of} \\
& \hspace{2em} (\text{inl } a') \rightarrow \{! \} \\
& \hspace{2em} (\text{inr } b') \rightarrow \{! \}
\end{align*}
\]

Complex Example (Cont.)

In case $ab = \text{inr } b$ and $ab' = \text{inr } b'$ we can similarly use

\[
\text{symB } b b' \text{ abab'}
\]

in order to solve our goal.

In case $ab = \text{inl } a$, and $ab' = \text{inr } b$

- we have $\text{EqAB ab ab'} = \text{False}$,

therefore

\[
abab' : \text{False}
\]

therefore empty case distinction on $abab'$ solves the goal.
Complex Example (Cont.)

- Similarly, in case \( ab = \text{inl} \ a \), and \( ab' = \text{inr} \ b \) we have that
  \[
  abab' : \text{False}
  \]
  and again empty case distinction on \( abab' \) solves the goal.
- The complete solution is on the next slide.

When we made the empty case distinctions, our goal was of type \( \text{False} \).
- Since in those cases \( abab' : \text{False} \), we could have solved the goal as well by directly inserting \( abab' \) in those cases.
- On the next slide is this alternative solution.
Remark on Case Distinction

- Case distinction over complex expressions causes problems in Agda.

Example (exampleCaseDistinctionComplexExpression.agda):

Assume we have defined $\text{ProdBool}$ as the product of two Boolean values:

$$\text{ProdBool} :: \text{Set} = \text{sig}$$

$$\text{first} :: \text{Bool}$$

$$\text{snd} :: \text{Bool}$$

Interactive Theorem Proving, CS_336, Lentterm 2004, Sec. 3(f)

Remark on Case Distinction

- The reason is that Agda will use in its reduction mechanism
  - only reductions from variables to other expressions,
  - but no reductions of complex expressions to other expressions.

- It would be very expensive to check reductions for complex expressions:
  - This would mean to check whether any subexpression of an expression matches the left side of any of those reductions.
  - Checking whether a variable which reduces occurs in a expression is instead a cheap operation.

Workaround

- One can work around this problem by defining an auxiliary function, which depends on a variable representing the complex expression.

- Then make case distinction on this single variable.

- In the example above define:

$$\text{h} :: \text{Bool}$$

$$\text{atom}(\text{and } a \ b)$$

$$:: \text{atom } a$$

$$= \text{case } a \text{ of}$$

$$\text{tt} \rightarrow \text{true}$$

$$\text{ff} \rightarrow \text{case } p \text{ of} \{ \}$$

Interactive Theorem Proving, CS_336, Lentterm 2004, Sec. 3(f)
Workaround

Now one can define the function in question in terms of the auxiliary function:

\[
\begin{align*}
f \ (\text{pair} :: \text{ProdBool}) \\
\quad \ (p :: \text{atom}(\text{and} \ p\text{.first} \ \text{pair}.\text{snd})) \\
\quad \quad : \ \text{atom} \ p\text{.first} \\
\quad = \ h \ \text{pair}.\text{first} \ \text{pair}.\text{second}
\end{align*}
\]

In the example, \( h \) had only as arguments the subexpressions of the complex expression in question.

In general, it might depend on other variables which form the context of the complex expression in question.

\[ (x :: A) \rightarrow B \ vs. \ \lambda(x :: A) \rightarrow s \]

There seems to be a confusion about the two expressions

\[ (x :: A) \rightarrow B \quad \text{vs.} \quad \lambda(x :: A) \rightarrow s \]

\((x :: A) \rightarrow B \) is the dependent function set.

- It is a set (or a type or a kind or a higher kind).
- Because it is a set, it makes sense to talk about \( r :: ((x :: A) \rightarrow B) \).
- \( r :: C \) makes only sense if \( C \) is a set or a type or a kind or a higher kind.

\( \lambda(x :: A) \rightarrow s \) is a function, which applied to \( x :: A \) returns \( s \).

- \( a :: (\lambda(x :: A) \rightarrow s) \) never makes sense, since \( \lambda(x :: A) \rightarrow s \) is not a set, type or (higher) kind.

\[ (x :: A) \rightarrow B \ vs. \ \lambda(x :: A) \rightarrow s \]

Especially, \( \lambda(x :: A) \rightarrow \text{Set} \) is a function which returns for \( x :: A \) the type \( \text{Set} \).

Note that

\[
\begin{align*}
A & \quad (b :: B) \\
\quad :: \ \text{Set} \\
\quad = \ d
\end{align*}
\]

is an abbreviation for

\[
\begin{align*}
A & \quad (b :: B) \rightarrow \text{Set} \\
\quad = \ \lambda(b :: B) \rightarrow d
\end{align*}
\]

So \( A \) defined as such is a function, not a set.

- It does not make sense to talk about \( c :: A \).
- Would be the same as \( c :: (\lambda(b :: B) \rightarrow d) \).

\[ (x :: A) \rightarrow B \ vs. \ \lambda(x :: A) \rightarrow s \]

\[ A \quad (b :: B) \]

\[
\begin{align*}
\quad :: \ \text{Set} \\
\quad = \ d
\end{align*}
\]

It does make sense to talk about

\[ c :: (b :: B) \rightarrow A b \]

Since \( (b :: B) \rightarrow A b \) is a set.
(g) The Set of Natural Numbers

The set \( \mathbb{N} \) is the type theoretic representation of the set
\[ \mathbb{N} := \{0, 1, 2, \ldots\} \, . \]

\( \mathbb{N} \) can be generated by
- starting with the empty set,
- adding 0 to it, and
- adding, whenever we have \( x \) in it \( x + 1 \) to it.

The Set of Natural Numbers (Cont.)

Let \( S \) be a type theoretic notation for the operation
\[ x \mapsto x + 1 \, . \]

Then the type theoretic rules are

\[
\begin{align*}
N & : \text{Set} \\
0 & : N \\
\vdots \\
n & : N \\
S n & : N
\end{align*}
\]

Primitive Recursion

Primitive Recursion expresses:
Assume we have
- \( a : \mathbb{N} \, . \)
- and, if \( n : \mathbb{N}, x : \mathbb{N} \) then \( g n x : \mathbb{N} \).

Then we can define \( f : \mathbb{N} \rightarrow \mathbb{N} \, , \) s.t.
- \( f 0 = a \, , \)
- \( f (S n) = g n (f n) \, . \)

Primitive Recursion (Cont.)

The computation of \( f n \) proceeds now as follows:
- Compute \( n \, . \)
- If \( n = 0 \) then the result is \( a \, . \)
- Otherwise \( n = S n' \, . \)
  - We assume that we have determined already how to compute \( f n' \, . \)
  - Now \( f n \) reduces to \( g n' (f n') \, . \)
  - \( g n' (f n') \) can be computed, since we know how to compute
    \[
    \begin{align*}
    & \cdot g \\
    & \cdot f n'.
    \end{align*}
    \]
Example

- The function $f : \mathbb{N} \rightarrow \mathbb{N}$ with $f \ n = 2 \cdot n$ can be defined primitive recursively by:
  - $f \ 0 = 0$.
  - $f \ (S \ n) = S \ (S \ (f \ n))$.
- Therefore take in the definition above:
  - $a = 0$,
  - $g \ n \ x = S \ (S \ x)$.

Generalized Primitive Recursion

- We can generalize primitive recursion as follows:
  - First we can replace the range of $f$ by an arbitrary set $C$
  - i.e. we allow for any set $C$
  - $f : \mathbb{N} \rightarrow C$
- Further, $C$ can now depend on $\mathbb{N}$.
- We obtain the following set of rules:

Rules for the Natural Numbers

Formation Rule

- $N : \text{Set}$

Introduction Rules

- $0 : \mathbb{N}$
- $n : \mathbb{N}$
- $S \ n : \mathbb{N}$

Elimination Rule

- $C : \mathbb{N} \rightarrow \text{Set}$
- $a : C \ 0$
- $f : (x : \mathbb{N}) \rightarrow C \ x \rightarrow C \ (S \ x)$
- $n : \mathbb{N}$
- $P \ C \ a \ f \ n : C \ n$

Equality Rules

- $P \ C \ a \ f \ 0 = a$
- $P \ C \ a \ f \ (S \ n) = f \ n \ (P \ C \ a \ f \ n)$
Rules for the Natural Numbers

Note that if we define in the elimination rule \( g := P \, C \, f \) then

The conclusion of the elimination rule reads:

\( g \, n : C \, n \)

which means that

\[ \lambda(n : N).g \, n : (n : N) \to C \, n \, . \]

The equality rules read:

\[ g \, 0 = a \]
\[ g \,(S \, n) = f \, n \,(g \, n) \]

Logical Framework Rules for N

The more compact notation is:

- \( N : \text{Set}, \)
- \( 0 : N, \)
- \( S : N \to N, \)
- \( P : (C : N \to \text{Set}) \to C \, 0 \)
- \( \to ((x : N) \to C \, x \to C \,(S \, x)) \)
- \( \to (n : N) \)
- \( \to C \, n \, . \)

Natural Numbers in Agda

N is defined using \textbf{data}:

\[
\text{data } N = Z \mid S(n :: N) \\
(\text{Unfortunately, } 0 \text{ is not an acceptable name in Agda}).
\]

Therefore we have

\[
Z :: N \\
S :: N \to N
\]

Elimination Rules for N in Agda

Elimination is represented in Agda as before via case distinction.

Assume we want to define

\[
f \,(n :: N) :: A = \{! !\}
\]

A possibly depending on \( n, \)

Then we can type into the goal \( n \) and use the menu agda-case.
Elimination Rules for N in Agda

- We get

\[
\begin{align*}
  f \ (n :: N) & :: A \\
  &= \text{case } n \text{ of} \\
  (Z) & \rightarrow \{! !\} \\
  (S \ n') & \rightarrow \{! !\}
\end{align*}
\]

Elimination Rules for N in Agda

- For solving the goals, we can now make use of \( f \). That will be accepted by the type checker.

- However, if we use of full \( f \), and then use menu item “check-termination”, we might obtain an error-message.

- If we do not make use of \( f \) in the case \( n=Z \) and only use of \( f \ n' \) in case \( n = S \ n' \), then check-termination succeeds.

Elimination Rules for N in Agda

- If check-termination succeeds, the definition should be correct.

- (The lecturer hasn’t checked the algorithm).

- However, if check-termination fails, the definition might still be correct.

Power of Termination Check

- The following definition of the Fibonacci numbers can’t be defined this way directly using the rules of type theory, but it can be defined in Agda as follows and check-termination accepts it:

\[
\begin{align*}
  \text{fib} \ (n :: N) & :: N \\
  &= \text{case } n \text{ of} \\
  (Z) & \rightarrow \text{one} \\
  (S \ n') & \rightarrow \text{case } n' \text{ of} \\
  (Z) & \rightarrow \text{one} \\
  (S \ n'') & \rightarrow \text{fib} \ n' + \text{fib} \ n''
\end{align*}
\]
Limitations of Termination Checker

Assume we define the **predecessor function**

\[
pred \ (n :: N) :: N = \begin{cases} 
  Z & \text{if } n = 0 \\
  n & \text{otherwise}
\end{cases}
\]

i.e.

\[
pred(n) = \begin{cases} 
  0 & \text{if } n = 0 \\
  n - 1 & \text{otherwise}
\end{cases}
\]

Because of the **undecidability of the Turing halting problem**
- it is undecidable whether a recursively defined function terminates or not
- there is no extension of check-termination, which accepts exactly all in Agda definable functions, which terminate for all inputs.

Unfortunately, Agda does currently not deal with **simultaneous recursion**
- i.e. the situation, where we decrease in one case w.r.t. one variable, in another case w.r.t. another variable.

In order to deal with this situation, one has to **rearrange proofs**.

On next slide there is an example of a proof which results in non-termination, although each recursive call descends.
- Refers to a definition of \( (<) :: (n, m :: N) \rightarrow \text{Set} \) which will be introduced below.
Example

\[
\text{mono} \ (n, k, m :: N)(p :: n < k) :: (n + m) < (k + m)
\]

\[=
\begin{cases}
\text{case } n \text{ of} \\
(Z) & \rightarrow \text{case } k \text{ of} \\
(Z) & \rightarrow \text{case } p \text{ of } \{ \}
\end{cases}
\]

\[
\begin{cases}
(S \ k') & \rightarrow \text{true} \\
(S \ m') & \rightarrow \text{mono } n \ k \ m' \ p
\end{cases}
\]

\[
\begin{cases}
(S \ n') & \rightarrow \text{case } k \text{ of} \\
(Z) & \rightarrow \text{case } p \text{ of } \{ \}
\end{cases}
\]

\[
\begin{cases}
(S \ k') & \rightarrow \text{case } m \text{ of} \\
\{ (Z) & \rightarrow \text{mono } n' \ k' \ m \ p \\
(S \ m') & \rightarrow \text{mono } n \ k \ m' \ p
\end{cases}
\]

Version accepted by Agda

The following version will be accepted by the termination checker:

- (this version corresponds exactly to induction on \(m\))

\[
\text{mono} \ (n, k, m :: N)
\]

\[
(p :: n < k) :: n + m < k + m
\]

\[=
\begin{cases}
\text{case } m \text{ of} \\
(Z) & \rightarrow p \\
(S \ m') & \rightarrow \text{mono } n \ k \ m' \ p
\end{cases}
\]

Amendment of Non-Termin. Version

- If one cannot reduce a non-terminating version directly in one with only one descend, one can use auxiliary lemmata instead.

- For instance in the previous non-terminating version, if one doesn’t observe the previous much better solution, one can

  - replace the first reference to mono by a reference to a lemma.

  - (this change is not really necessary, since only the second reference is responsible for rejection by the termination checker)

  - and observe that the second reference can be replaced by \(p\).

\[
\text{lemma} \ (m, k :: N)
\]

\[
:: \ Z + m < S \ k + m
\]

\[=
\begin{cases}
\text{case } m \text{ of} \\
(Z) & \rightarrow \text{true} \\
(S \ m') & \rightarrow \text{lemma } m' \ k
\end{cases}
\]
Example: Addition

Definition of + in Agda:

\[(+ \ (n, m :: N)) :: N\]

\[= \ \text{case } m \ \text{of}\]

\[\{ \ (Z) \ \rightarrow \ n \]
\[\ (S \ m') \ \rightarrow \ S \ (n + m') \]

The definition expresses:

\[n + 0 = n\]
\[n + (m' + 1) = (n + m') + 1\]

Note that (+) is used **infix**, i.e. we write \(n + m\) for \((+ \ n \ m)\).

If \(m = Sm'\), the definition of \((+ \ n \ m)\) refers to \((+ \ n \ m')\),

\((+ \ n \ m')\) is **defined before** \((+ \ n \ m)\) since \(m'\) is introduced before \(m\).

Example: Multiplication

Definition

\[(\ast \ (n, m :: N)) :: N\]

\[= \ \text{case } m \ \text{of}\]

\[\{ \ (Z) \ \rightarrow \ Z \]
\[\ (S \ m') \ \rightarrow \ n \ast m' + n \]

The definition expresses:

\[n \cdot 0 = 0\]
\[n \cdot (m' + 1) = (n \ast m') + n \]
Example: Multiplication (Cont.)

- Again * is **treated infix**.
- Agda has built in that * **binds more than** +.
- n * m' + n is treated as (n * m') + n.
- Note that the definition of * requires, that + **is already defined**.

Equality on N

- The equality \((n == m) :: \text{Set}\) for \(n, m :: N\) can be defined using the equations:
  - \((Z == Z) = \text{True}\).
  - \((Z == S \, n) = (S \, n == Z) = \text{False}\).
  - \((S \, n == S \, m) = (n == m)\).

Equality on N (Cont.)

- From this one can now derive a definition in Agda:
  \[
  (==) \quad (n, m :: N)
  \quad :: \quad \text{Set}
  \quad = \quad \text{case } n \text{ of}
  \quad \quad (Z) \quad \to \quad \text{case } m \text{ of}
  \quad \quad \quad (Z) \quad \to \quad \text{True}
  \quad \quad \quad (S \, m') \quad \to \quad \text{False}
  \quad \quad (S \, n') \quad \to \quad \text{case } m \text{ of}
  \quad \quad \quad (Z) \quad \to \quad \text{False}
  \quad \quad \quad (S \, m') \quad \to \quad (n' == m')
  \]

Reflexivity of ==

- **Reflexivity** of == is the formula:
  \[
  \forall n : N. \, n == n
  \]

- **Type theoretically** this means that we have to define a function refl:
  \[
  \text{refl} \quad (n : N)
  \quad :: \quad n == n
  \quad = \quad \{! !\}
Reflexivity of == (Cont.)

- This can now be shown using case distinction:

\[
\begin{align*}
\text{refl} \ (n : N) \\
:: & \ n == n \\
= & \ \text{case } n \ \text{of} \\
\quad (Z) & \ \to \ \{! \} \\
\quad (S \ n') & \ \to \ \{! \}
\end{align*}
\]

Symmetry of ==

- Symmetry of == is the formula:

\[
\forall n, m : N. n == m \to m == n
\]

- Type theoretically this means that we have to define a function \(\text{sym}\):

\[
\begin{align*}
\text{sym} \ (n, m : N) \\
(p :: n == m) \\
:: & \ m == n \\
= & \ \{! \}
\end{align*}
\]

Reflexivity of == (Cont.)

- Case \(n = Z\) is trivial.

- Case \(n = S \ n'\) can be solved using \(\text{refl } n'\) (which is defined before \(\text{refl } n'\)).

Symmetry of == (Cont.)

- This can now be shown using case distinction:

\[
\begin{align*}
\text{sym} \ (n, m : N) \\
(p :: n == m) \\
:: & \ m == n \\
= & \ \text{case } n \ \text{of} \\
\quad (Z) & \ \to \ \text{case } m \ \text{of} \\
\quad \quad \{ (Z) & \ \to \ \{! \} \\
\quad \quad (S \ m') & \ \to \ \{! \}
\quad (S \ n') & \ \to \ \text{case } m \ \text{of} \\
\quad \quad \{ (Z) & \ \to \ \{! \} \\
\quad \quad (S \ m') & \ \to \ \{! \}
\end{align*}
\]
Symmetry of == (Cont.)

- The first goal can be solved by using true (since \( Z == Z \) = True).
- For the second goal we know \( p \) is an element of \( Z == S \ m' \) which is False.
  - Therefore if we make case distinction on \( p \) we get
    
    \[
    \text{case } p \text{ of } \{ \}
    \]
    
    and have solved the second goal.
- Similarly the third goal can be solved.

Example: < on N

- The following introduces < on N:
  
  \[
  (\langle \rangle \to Set) \quad :: \quad \text{Set} \\
  = \quad \text{case } m \text{ of} \\
  \quad (Z) \quad \to \quad \text{False} \\
  \quad (S \ m') \quad \to \quad \text{case } n \text{ of} \\
  \quad \quad (Z) \quad \to \quad \text{True} \\
  \quad \quad (S \ n') \quad \to \quad \text{n'} < m'
  \]

Symmetry of == (Cont.)

- In the fourth goal, we have as type of goal \( S \ m' == S \ n' \) which is identical to \( m' == n' \).
  - The type of \( p \) is \( S \ n' == S \ m' \) which is identical to \( n' == m' \).
  - The goal can be solved by using sym \( n' \ m' \ p \).
  - Note that we can use here \( p \) since it is of type \( n' == m' \).
  - It is correct to use it since \( n' \) is introduced before \( n \).
    - Therefore
      
      \[
      \text{sym } n' \text{ can be defined before sym } n.
      \]
  - This definition will be accepted by check-termination.

Example: Tuples of Length n

- We define tuples (or vectors) of length \( n \) in Agda.
  - Define first
    
    \[
    \begin{align*}
    \text{data Nil} & = \text{nil} \\
    \text{Cons } (A, B :: \text{Set}) & :: \text{Set} \\
    \text{data } \text{cons}(a :: A)(b :: B) & = \text{data cons}(a :: A)(b :: B)
    \end{align*}
    \]

Interactive Theorem Proving, CS 336, Lentterm 2004, Sec. 3(g)
Tuples of Length $n$

Now we can define

$$\text{Tuple} \ (A :: \text{Set}) \ 
(n :: N) \ :: \ \text{Set} \ 
= \ \text{case} \ n \ \text{of} \ 
\ (Z) \ \rightarrow \ \text{Nil} \ 
(S \ m') \ \rightarrow \ \text{Cons} \ A \ (\text{Tuple} \ A \ m')$$

Therefore (with the obvious definition of two),

$$\text{Tuple} \ A \ n = \text{Cons} \ A \ (\text{Cons} \ A \ \cdots \ (\text{Cons} \ A \ \text{Nil}) \ \cdots) \ \text{for } n \ \text{times}$$

The elements of Tuple $A \ n$ are

$$\text{cons} \ a_1 \ (\text{cons} \ a_2 \ \cdots \ (\text{cons} \ a_n \ \text{nil}) \ \cdots)$$

for elements $a_1, \ldots, a_n$ of $A$.

In ordinary mathematical notation, we would write

$$\langle a_1, \ldots, a_n \rangle$$

for such an element.

Remarks on Tuples of Length $n$

In ordinary mathematics, we would define

$$\text{Tuple}(A,0) \ := \ \{\langle \rangle \} \ ,$$

$$\text{Tuple}(A,n+1) \ := \ \{\langle a_1, \ldots, a_{n+1} \rangle \mid a_1, \ldots, a_{n+1} \in A\} \ .$$

If we define

$$\text{nil} \ := \ \langle \rangle \ ,$$

$$\text{cons}(a_1, \langle a_2, \ldots, a_{n+1} \rangle) \ := \ \langle a_1, \ldots, a_{n+1} \rangle \ ,$$

then this reads:

$$\text{Tuple}(A,0) \ := \ \{\text{nil} \} \ ,$$

$$\text{Tuple}(A,n+1) \ := \ \{\text{cons}(a,b) \mid a \in A \wedge b \in \text{Tuple}(A,n)\} \ .$$
Remarks on Tuples of Length n

In the type theoretic definition we have **constructors**
- \( \text{nil} :: \text{Tuple} A Z \)
- \( \text{cons}@(\text{Tuple} A (S n)) :: A \rightarrow \text{Tuple} A n \rightarrow \text{Tuple} A (S n) \).

This is the **type theoretic analogue** of the previous definitions.

---

Example: Sum of n-Tuples

Define

\[
\text{NTuple } (n :: N) :: \text{Set} \\
= \text{Tuple} N n
\]

**NTuple** \( n \) are tuples of natural numbers of length \( n \).

---

Componentwise Sum of n-Tuples

We define **component-wise sum of tuples of length** \( n \).
- Using mathematical notation, this sum for instance as follows:

\[
\langle 2, 3, 4 \rangle + \langle 5, 6, 7 \rangle = \langle 7, 9, 11 \rangle .
\]

---

Componentwise Sum of n-Tuples

Define

\[
\text{sumNTuple } (n :: N) (avec, bvec :: \text{NTuple} n) :: \text{NTuple} n = \text{case } n \text{ of} \\
(Z) \rightarrow \text{nil} \\
(S n') \rightarrow \text{case } avec \text{ of} \\
(cons a avec') \rightarrow \text{case } bvec \text{ of} \\
(cons b bvec') \rightarrow \text{cons}@(a + b) \text{ (sumNTuple } n' \text{ } avec' \text{ bvec')}
\]
(h) Lists

We define the set of lists of elements of type \( A \) in Agda.

We have two constructors:
- nil, generating the empty list.
- \( \text{cons} \), adding an element of \( A \) in front of a list

So we define lists as:

\[
\text{list} \ (A :: \text{Set}) :: \text{Set} = \text{data} \ \text{nil} \\
\mid \ \text{cons}(a :: A) \ (l :: \text{list} A)
\]

Elimination Rule for Lists

Elimination rule uses list-recursion:
Assume
- \( A : \text{Set} \)
- \( C :: \text{Set} \), depending on \( l :: \text{list} A \).

Then we can define

\[
f \ (l :: \text{list} A) :: C = \text{case} \ l \ of \\
\ (\text{nil}) \rightarrow \{! \}
\]

and in the second goal we can make use of \( f \ l' \).

Example: Length of a List

\[
\text{length} \ (l :: \text{list} N) :: N = \text{case} \ l \ of \\
\ (\text{nil}) \rightarrow \text{Z} \\
\ (\text{cons} \ a \ l') \rightarrow \text{S} \ (\text{length} \ l')
\]

Example: \( \text{sumlist} \)

\( \text{sumlist} \ l \) will compute the sum of the elements of list \( l \).

\[
\text{sumlist} \ (l :: \text{list} N) :: N = \text{case} \ l \ of \\
\ (\text{nil}) \rightarrow \text{Z} \\
\ (\text{cons} \ n \ l') \rightarrow n + \text{sumlist} \ l'
\]
Interesting Exercise

Define

\[
\text{append} : (A : \text{Set}) \rightarrow (\text{list} \ A) \rightarrow (\text{list} \ A) \rightarrow \text{list} \ A
\]

s.t. \( \text{append} \ A \ l \ l' \) is the result of appending the list \( l' \) at the end of list \( l \).

E.g., if \( a, b, c, d \) are elements of \( A \), and if we define \( \text{cons} := \text{cons@}(\text{list} \ A) \), \( \text{nil} := \text{nil@}(\text{list} \ A) \), then:

\[
\text{append} \ A \ (\text{cons} \ a \ (\text{cons} \ b \ \text{nil})) \ (\text{cons} \ c \ (\text{cons} \ d \ \text{nil})) = \text{cons} \ a \ (\text{cons} \ b \ (\text{cons} \ c \ (\text{cons} \ d \ \text{nil})))
\]

(i) Universes

A universe \( U \) is a set, the elements of which are codes for sets.

So we have

- \( U : \text{Set} \),
- \( T : U \rightarrow \text{Set} \) (the decoding function).

We consider in the following a universe closed under

- \( \text{Fin}_0, \text{Fin}_1, \text{Bool}, \)
- \( \text{N}, \)
- \( +, \)
- \( \Sigma, \)
- the dependent function type.
**Rules for the Universe**

**Introduction and Equality Rules (Cont.)**

\[
\begin{align*}
  a : U & \quad b : U \\
  \Rightarrow a + b : U
\end{align*}
\]

\[
T(a + b) = T(a) + T(b) : \text{Set}
\]

\[
\begin{align*}
  a : U & \quad b : T(a) \rightarrow U \\
  \Rightarrow \widehat{\Sigma}(a, b) : U
\end{align*}
\]

\[
T(\widehat{\Sigma}(a, b)) = \Sigma \ T(a) \ (\lambda x. T \ (b \ x)) : \text{Set}
\]

**Applications of the Universe**

- Ordinary elimination rules don’t allow to eliminate into \( \text{Set} \).
- However often, one can verify, that all sets needed are “elements of a universe”, i.e. there are codes in the universe representing them.
- Then one can eliminate into the universe instead of \( \text{Set} \) and use \( T \) to obtain the required function.

---

**Elimination and Equality Rules**

- There exist as well elimination rules and corresponding equality rules for the universe.
- They are very long (one step for each of constructor of \( U \)) and are not very much used.
- They follow the principles present in previous rules.
Applications of the Universe

Example: Define

\[
\widehat{\text{atom}} : \text{Bool} \rightarrow U,
\]

\[
\widehat{\text{atom}} := \text{Case}_{\text{Bool}} (\lambda(x : \text{Bool}). U) \text{ Fin}_1 \text{ Fin}_0,
\]

atom : Bool \rightarrow Set,
atom : \lambda(x : \text{Bool}). T (\widehat{\text{atom}} x),

Then
atom tt = Fin_1,
atom ff = Fin_0.

Universes in Agda

U and T need to be defined simultaneously.
Usually Agda type checks definitions in sequence,
so no reference to later definitions possible.
Special construct mutual.
Everything in the scope of it is type checked
simultaneously.
Scope determined by indentation.

Universes in Agda (Cont.)

\[
\begin{align*}
\text{mutual} && \text{U} & : \text{Set} \\
&& = & \text{data Nhat} \\
&& & | \text{Finzerohat} \\
&& & | \text{Finonehat} \\
&& & | \text{Boolhat} \\
&& & | \text{Sigmahat} (a :: U)(b :: T a \rightarrow U) \\
&& & | \text{Pihat} (a :: U)(b :: T a \rightarrow U)
\end{align*}
\]

T in the following is to be intended the same as U:

\[
\begin{align*}
T \ (u :: U) && : & \text{Set} \\
&& = & \text{case} \ u \ \text{of} \\
& (\text{Nhat}) & \rightarrow & N \\
& (\text{Finzerohat}) & \rightarrow & \text{Finzero} \\
& (\text{Finonehat}) & \rightarrow & \text{Finone} \\
& (\text{Boolhat}) & \rightarrow & \text{Bool} \\
& (\text{Sigmahat} \ a \ b) & \rightarrow & \text{Sigma} (T \ a) \\
& & & (\lambda(x :: T \ a) \rightarrow T \ (b \ x)) \\
& (\text{Pihat} \ a \ b) & \rightarrow & (x :: T \ a) \rightarrow T \ (b \ x)
\end{align*}
\]
**Algebraic Types**

The construct “data” in Agda is much more powerful than what is covered by type theoretic rules. In general we can define now sets having arbitrarily many constructors with arbitrarily many arguments of arbitrary types.

\[
A :: Set = \text{data } C_1(a_{11} :: A_{11}) \cdots (a_{1n_1} :: A_{1n_1}) \\
\mid C_2(a_{21} :: A_{21}) \cdots (a_{2n_2} :: A_{2n_2}) \\
\ldots \\
\mid C_m(a_{m1} :: A_{m1}) \cdots (a_{mn_m} :: A_{mn_m})
\]

**Meaning of “data”**

The idea is that \( A \) as before is the least set \( A \) s.t. we have constructors:

\[
C_1 \circ A :: (a_{11} :: A_{11}) \\
\to \ldots \\
\to (a_{1n_1} :: A_{1n_1}) \\
\to A
\]

where a constructor always constructs new elements.

In other words the elements of \( A \) are exactly those constructed by those constructors.

**Strictly Positive Algebraic Types**

In the types \( A_{ij} \) we can make use of \( A \).

- However, it is difficult to understand \( A \), if we have **negative** occurrences of \( A \).
- Example:
  \[
  A :: Set \\
  = \text{data } C (f :: A \to A)
  \]
- What is the least set \( A \) having a constructor
  \[
  C \circ A :: (f :: A \to A) \\
  \to A ?
  \]

If we
- have constructed some part of \( A \) already,
- find a function \( f :: A \to A \), and
- add \( C \circ f \) to \( A \),
then \( f \) might no longer be a function \( A \to A \).
(\( f \) applied to the new element \( C \circ f \) might not be defined).

In fact, “agda-check-termination” issues a warning, if we define \( A \) as above.

We shouldn’t make use of such definitions.
Strictly Positive Algebraic Types

A “good” definition is the set of lists of natural numbers, defined as follows:

\[
\text{Nlist} :: \text{Set} = \text{data nil | cons (a :: N) (l :: Nlist)}
\]

The constructor \(\text{cons@}_\) of N-lists refers to Nlist, but in a positive way:

We have: if \(a :: \text{N} \) and \(l :: \text{Nlist}\), then we have \(\text{cons@}_a l :: \text{Nlist}\).

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 3(j)

In general:

\[
A :: \text{Set} = \text{data } C_1 (a_{11} :: A_{11}) \cdots (a_{1n} :: A_{1n}) \\
| C_2 (a_{21} :: A_{21}) \cdots (a_{2n} :: A_{2n}) \\
| \cdots \\
| C_m (a_{m1} :: A_{m1}) \cdots (a_{mn} :: A_{mn})
\]

is a strictly positive algebraic type, if all \(A_{ij}\) are

\(\text{either types which don’t make use of } A\)

\(\text{or are } A \text{ itself.}\)

And if \(A\) is a strictly positive algebraic type, then \(\Sigma A\) is acceptable.

The definitions of finite sets, \(\Sigma A B\), \(A + B\) and \(N\) were strictly positive algebraic types.

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 3(j)

One further Example

The set of binary trees can be defined as follows:

\[
\text{Bintree} :: \text{Set} = \text{data leaf | branch (left :: Bintree) (right :: Bintree)}
\]

This is a strictly positive algebraic type.

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 3(j)
Extensions of Strict.Pos. Alg. Types

- An often used extension is to define several sets simultaneously inductively.
- Example: the even and odd numbers:
  
  ```haskell
  mutual
  Even :: Set
  = data Z | S (n:: Odd)

  Odd :: Set
  = data S (n::Even)
  ```

  In such examples the constructors refer strictly positive to all sets which are to be defined simultaneously.

Elimination Rules for data

- Functions from strictly positive algebraic types can now be defined by case distinction as before.
- For termination we need only that in the definition of \( f \), when have to define \( f \ (C@_ a_1 \cdots a_n) \), we can refer only to \( f \) applied to elements used in \( C@_ a_1 \cdots a_n \).

Examples

- For instance, in the Bintree example, when defining
  
  ```haskell
  f :: Bintree -> A
  ```

  by case-distinction, then the definition of
  
  ```haskell
  f (branch@_ left right)
  ```

  can make use of \( f \) left and \( f \) right.

- The last definition is unproblematic, since, if we have \( f :: N -> O \) and construct \( \text{lim}@_ f \) out of it, adding this new element to \( O \) doesn’t destroy the reason for adding it to \( O \).

- So again \( O \) can be “constructed”.

Interactive Theorem Proving, CS336, Lentterm 2004, Sec. 3(i)
Examples

- In the example of \( e \), when defining

\[
g :: 0 -> A
\]

by case-distinction, then the definition of

\[
g (\lim @_f)
\]

can make use of \( g (f n) \) for all \( n : : N \).

Revision Lecture

1.
2.
3.

General

- According to my data, exam Wednesday 19 May, 2 pm, Arts Hall.
  - Please check.
- **Essentially everything needed should be contained in the notes.**
- I will come to the exam in the first half hour (approx.; if you have questions ask then).
- All three **courseworks** are preparation for the exam.

Structure of the Exam

3 Questions.

- **Question 1** will be mainly on the \( \lambda \)-calculus, term rewriting and reduction systems (mainly Sect. 1 (d)).
- **Question 2** will be mainly on the dependent function set and the dependent product in Agda (Sect. 2).
- **Question 3** will be on Sect. 3, and of the material there on
  - data types
    - finite sets, Bool, and atom,
    - product and disjoint union
  - and simple definitions and proofs in Agda.
Structure of the Exam

- Agda questions will be in such a way that they can easily be solved by hand.
- Derivations only for
  - the \( \lambda \)-calculus as in Cwk 1,
  - the dependent \( \lambda \)-calculus as in Cwk2, Qu. 6.

Introduction (Cont.)

- 4 kinds of judgements in dependent type theory:
  - \( A : \text{Set}, A = B : \text{Set}, a : A, a = b : A. \)
  - Only \( A : \text{Set}, a : A \) are visible in Agda.
- Examples of dependent types in programming.
  - Templates (e.g. in C++) ie. parametric types.
  - Matrix multiplication.
  - Predicates.
  - Dependent grammars in linguistics.

1 Introduction

- 4 Principal approaches for writing verified software.
- Concept of a type.
  - Advantages of typed, of untyped languages.
  - Why are types good for writing correct software?
- Examples of types in other languages
  - Scalar types (e.g. Booleans, Integers).
  - Simple compound types (records, arrays).
  - Function types, algebraic types (“data”)
  - Interfaces.

\( \lambda \)-Calculus

- The untyped \( \lambda \)-Calculus:
  - Bound and free variables,
  - \( \alpha \)-, \( \beta \)-conversion.
- The typed \( \lambda \)-calculus:
  - Types,
  - rules,
  - product types,
  - \( \eta \)-rule.
  - But nothing to be “learned by heart” about the (un)typed \( \lambda \)-calculus – only how to use it as in the Cwk.
Reduction Systems

- **Notions**
  - Reduction system,
  - strongly normalising, weakly normalising,
  - normal form, irreducible,
  - confluence.
- **Too complicated** for the exam:
  - definition of term rewriting system and how to obtain the reduction system,
  - definition of →∗, ↔∗.
  - (But intuitive understanding of these notions is required).

2 The Logical Framework.

- **4 kinds of rules** (formation, introduction, elimination, equality).
- **Constructors and canonical elements.**
- **Dependent function type, dependent product.**
  - Rules only up to the level of Cwk2, Qu. 6.
  - Mainly how to use them in Agda (Cwk 2, Qu 1 - 3).
  - Notion of Set vs. Type (how to use Set, Type).
- **Let expressions.**

Cwk 1

- Typical questions as in Qu 1, 3, 4, 5 and Qu 6 (a), 6 (b), and part (i) of 6 (c) - 6 (e).

3. Data Types

- **Basic data types** (Booleans, finite sets, disjoint union)
  - Only how to introduce and use them in Agda.
- Two versions of the nondependent-dependent product in Agda
  - using “sig”,
  - using “data”.
- **Atomic formulae** (atom).
- How to represent other formulae in type theory.
- How to carry out simple proofs in type theory.
- Questions like in **Cwk 3, all questions** (only simple ones feasible in the exam).
Proofs in Agda

- Mainly proofs related to equality and \(<\)-relation.
- Definition of these relations in \textit{Bool}, disjoint union, nondependent product as in the \textit{Cwk} or similar to that.
- Simple proofs about them (notions like reflexive, anti-reflexive, transitive, symmetric; proofs of such properties).

Restrictions

- Constructive logic, natural numbers, lists not treated in the lecture.
- No natural deduction.
- No formal derivations, rules in this section.