B1. Introduction

(a) **Principal Approaches to Writing Verified Software**

(b) **The Theorem Prover Agda**

(c) **Concept of a Type**

(d) **Rules and Judgements**

(e) **Dependent Judgements**

(f) **Examples of Dependent Types in Programming.**

(g) **Dependent function types and products.**

(h) **Structural Rules.**
Principal Approaches to 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.

(ii) Prove that a solution for the problem exists. Extract a program out of it.
   - Technology not yet far developed.
(iii) Write programs 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:
  ∗ 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.
(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.
  - Coq (INRIA, France), as well technically very advanced.
  - LEGO (Edinburgh), about to be replaced.
  - 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.

- We will use in this course Agda, but Alfa can be used to create Agda code.
• Half, Agda, Alf are all written in Haskell.

• Half and Agda have an Emacs mode, which makes it quite convenient to develop proofs in it.

• In most theorem prover, 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.

• The Alf-family has a different approach of successive refinement.
  – One starts writing the proof code as one writes a functional program, as far as one can do this without machine assistance.
  – What one cannot do 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

- It is planned that Agda will be installed in the Linux Lab during the first week of February.

- Agda is most easily installed under Linux or other versions of Unix.

- It has been reported that it can 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.
Typed vs. untyped languages:

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

Advantage of untyped languages:
Greater freedom in programming.

Advantage 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.

In order to obtain correct software, we make use of a much more refined type system.

- It allows to specify any property of a program, which can be defined as a formula, as a type.
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, inductive data types (= what can be defined using “data”).

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

- **Function types.**
  \( \text{Int} \rightarrow \text{Int} \) is the type of functions mapping integers to integers.

- **Products** (essentially records).
  \( \text{Int} \times \text{Int} \) is the type of pairs \( \langle r, s \rangle \), where \( r, s \) are integers.

- **Inductive data types.** More about this later.
• The type theory will allow us to both write programs and prove their correctness.

• Proving requires usually several steps. Similarly in dependent type theory, programs have to be derived:
  Essentially we derive, that a program \( f \) is of a type \( A \), written

\[
f : A
\]

– Example: for a sorting algorithm, we derive:

\[
\text{sort} : \text{List} \to \text{Sorted} \to \text{List}.
\]

• **Remark on syntax:**
  – In the lecture we will use single colons “:\=” for “has type”.
  – In Haskell “:\=” 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 don’t play an important rôle there).
Rule Based Approach

- In functional programming, one creates a file containing expressions together with their types. E.g. one has defined before \( f : A \rightarrow B, \ a : A \). Now one can introduce \( f a : B \).

- The simple type theory of e.g. Haskell makes it easy to do this “by hand”.

- The steps taken before correspond to having a rule:
  If \( f : A \rightarrow B, \ a : A \), then \( f a : B \).
  We write for this briefly

\[
\frac{f : A \rightarrow B \quad a : A}{f a : B}
\]

- Writing down rules makes it more clear how the systems works.
  It makes it easier as well to reason about the system.
Rule Based Approach (Cont.)

- In Agda, programs will be typed in, similarly as in Haskell.
  But some holes can be left open.

- The rule

\[
\frac{f : A \to B \quad a : A}{f\ a : B}
\]

will allow us to carry out the following steps:

- Assume \( f :: A \to B \) and \( a :: A \) have been introduced.
- Assume we have a judgement: \( ? :: B \).
  Here \( ? \) is a goal.
  (Precise syntax in Agda: \{!!\} :: B; a judgement is a typing statement in type theory).
- Now we insert into the goal “\( f \)” and choose from a menu “refine”.
- Then the judgement changes to \( f\ ? :: B \).
- The menu “infer type” tells us, that the new goal requires something of type \( A \).
- Again we can insert into the new goal “\( a \)” and choose “refine”.
- We obtain the judgement \( f\ a :: B \).
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.

Therefore we have not only the judgement as in functional programming

\[ a : A \]

but as well a typing judgement

\[ A : \text{type} \]

“A is a type”.

Before deriving \( a : A \) we first have to show \( A : \text{Type} \).
Equality Judgements

• On a machine level, terms which reduce to the same will be identified:
  E.g. \((\lambda x.x) \, r\) and \(r\) will be identified.

  – Here \((\lambda x.x)\) is a notation for Haskell \(\textbackslash x \rightarrow x\).

• If one needs at some place \(r\) and has \(s = (\lambda x.x) \, r\),
  one can insert \(s\) instead of \(r\).

• In the rule based system, it is better to add two more judgements:
  – \(a = b : A\) for \(a, b\) are equal elements of type \(A\).
  – \(A = B : \text{Type for } A \text{ and } B \text{ are equal types.}\)
    (The last one is something novel in the setting of dependent type theory).
We have the following 4 types of judgements:

- $A : \text{Type}$ "$A$ is a type".
- $a : A$ "$a$ is of type $A$".
- $A = B : \text{Type}$ "$A$ and $B$ are equal types".
- $a = b : A$ "$a$ and $b$ are equal elements of type $A$".
Rules of the Non-Dependent Product

Formation Rule

\[
\frac{A : \text{Type} \quad B : \text{Type}}{A \times B : \text{Type}}
\]

Introduction Rule

\[
\frac{a : A \quad b : B}{\langle a, b \rangle : A \times B}
\]

Elimination Rules

\[
\frac{c : A \times B}{\pi_0(c) : A} \quad \frac{c : A \times B}{\pi_1(c) : B}
\]

Equality Rules

\[
\frac{a : A \quad b : B}{\pi_0(\langle a, b \rangle) = a : A} \quad \frac{a : A \quad b : B}{\pi_1(\langle a, b \rangle) = b : B}
\]
Explanation of the Names for the Rules

- Formation rule: Allows us to form a new type.

- Introduction rule: Allows us to introduce a new element of a type.
  - Elements introduced by an introduction rules are called canonical elements of that type.

- Elimination rule: Allows to take an element from this type, and generate out of it an element of another type.
  - Elements introduced by an elimination rule are called non-canonical elements.

- Equality rules: If we introduce an element and then eliminate it again, we have done a detour via our current type. Equality rules tell us how to eliminate this detour.
  - So non-canonical elements can be reduced.
Consider the judgement in functional programming:

\[ \lambda x. x : A \rightarrow A \]

The most convenient rule for deriving \( \lambda \)-epression has as premise in the above case:
If \( x : A \) then \( x : A \).

This requires that we have judgements, which depend on assumptions about variables.

The above will be written as

\[ x : A \Rightarrow x : A \]

Here \( x : A \) is called context.

In general this context can consist of several variables. For instance, if \( A : \text{Type}, \ a, b : A \), then

\[ f : A \rightarrow \text{Set}, x : f \ a \Rightarrow x : f \ b \]

(“Set” will be explained later).
Rules of the Non-Dependent Function Type

Formation Rule

\[ A : \text{Type} \quad B : \text{Type} \]
\[ \frac{\quad}{A \rightarrow B : \text{Type}} \]

Introduction Rule

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

Elimination Rule

\[ c : A \rightarrow B \quad a : A \]
\[ \frac{c a : B}{\quad} \]

Equality Rule

\[ x : A \Rightarrow b : B \quad a : A \]
\[ \frac{(\lambda x.b) a = b[x := a] : B}{\quad} \]

Here \( b[x := a] \) is the result of substituting in \( b \) every occurrence of variable \( x \) by the term \( a \) (after some renaming of bounded variables).
• In Agda, one has to provide a type to lambda terms. So instead of writing

\[ x \rightarrow x :: A \rightarrow A \]

one writes

\[ f :: A \rightarrow A = (x :: A) \rightarrow x \]

• In Agda, we have no explicit contexts, since we don’t use rules. However, if we have the open judgement

\[ (x :: B) \rightarrow? :: B \rightarrow A \]

then we can make use of \( x :: B \) for refining the goal.
This context can be shown with a menu.
Dependent types are often needed in programming. Some examples:

- In Java, a relatively big library of “collection classes” is available. They provide implementations of lists, sets, hash tables etc. It would be nice to have “lists of type $A$”. However this is a dependent type, depending on a type $A$. This cannot be carried out in Java. Instead, in Java only lists of elements of type Object are available. Elements of other types have to be downcast to Object. Therefore type checking happens at run time rather than at compile time.

  - This is a weak form of dependent types, polymorphism.
  - In C++, this form of dependency is available (called templates).
Examples of Dependent Types in Programming (Cont.)

- **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 \( n, m, k \).

  - 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. This is not a very clean approach.
• Predicates will be dependent types. In the type of sorting algorithms

$$\text{List} \to \text{Sorted} \rightarrow \text{List}$$

Sorted \rightarrow \text{List} is the type of lists which are sorted. Here sorted is a type, which depends on the list it refers to.

We will soon learn how to express this type.

• Aarne Ranta has used dependent types in linguistics:

– In a sentence like “The man goes home”, the predicate (“goes”) depends on, whether the subject (“The man”) is singular or plural.
– He constructed grammars based on dependent types and used them for translating sentences between different languages.
(g) Dependent Function Types and Products

- In the presence of dependent types we have now a new dependent function type, where the type of the result depends on the argument of the function.

- Notation: \((x : A) \rightarrow B\), for the type of functions \(f\) which map an element \(a : A\) to an element of \(B[x := a]\).

- In set theoretic pseudo notation this is:

\[
\{ f \mid f \text{ function} \\
\quad \land \text{dom}(f) = A \\
\quad \land \forall a \in A. f(a) \in B[x := a] \}
\]
Examples of the Use of the Dependent Function Type

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

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

Then matrix multiplication has type

\[
(n : N) \\
\to (m : N) \\
\to (k : N) \\
\to \text{Mat}(n, m) \\
\to \text{Mat}(m, k) \\
\to \text{Mat}(n, k)
\]

or, using a shorter notation,

\[
(n, m, k : N) \\
\to \text{Mat}(n, m) \\
\to \text{Mat}(m, k) \\
\to \text{Mat}(n, k)
\]
Examples of the Use of the Function Types (Cont.)

• The ordinary function type $A \rightarrow B$ is a special case of $(x : A) \rightarrow B$, where $B$ doesn’t depend on $x$. 
Rules of the Dependent Function Type

Formation Rule

\[
\frac{A : \text{Type} \quad x : A \Rightarrow B : \text{Type}}{(x : A) \rightarrow B : \text{Type}}
\]

Introduction Rule

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

Elimination Rule

\[
\frac{f : (x : A) \rightarrow B \quad a : A}{f\ a : B[x := a]}
\]

Equality Rule

\[
\frac{x : A \Rightarrow b : B \quad a : A}{(\lambda x.b)\ a = b[x := a] : B[x := a]}
\]
The $\eta$-rule has a special status:

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

\[
f : (x : A) \rightarrow B \\
\frac{f = \lambda x. f \ x : (x : A) \rightarrow B}{f : (x : A) \rightarrow B}
\]
The Dependent Function Type in Agda

- In Agda one writes \((x:A) \rightarrow C\) for the dependent function type \(A \rightarrow C\) for the nondependent function type. We write when referring to Agda code always \(\rightarrow\) instead of \(-\rightarrow\).

- Similar to Haskell, Agda uses
  - \(\backslash\) as substitute for \(\lambda\),
  - \(\rightarrow\) in order to separate the variable and the term.

- In Haskell, the notation is \(\backslash x \rightarrow a\) for \(\lambda x.a\).

- We will use, when writing informally Agda code
  - \(\lambda\) instead of \(\backslash\),
  - \(\rightarrow\) instead of \(-\rightarrow\).
The Dependent Function Type in Agda (Cont.)

- Agda enforces to attach a type to lambda-terms, although this is not really necessary (in order to be compatible with Cayenne).

- So the full Agda notation for $\lambda x.x : A \to A$ is:

  $$\lambda(x :: A) \to x$$

- So, if $b :: B$ depending on $x :: A$, then we can introduce

  $$g = \lambda(x :: A) \to b$$
  $$:: (x :: A) \to B$$

- Usually the type $A$ in an expression $\lambda(x :: A)$, can be inferred automatically by the system.

  – Use the Agda command (agda-solve), C-c =.
As Haskell, Agda allows as well to introduce elements of a function type without using \( \lambda \):

Instead of

\[
f :: A \to A
= \lambda (a :: A) \to a
\]

one can write

\[
f(a :: A)
:: A
= a
\]

The latter notation, which coincides with the definition functions in imperative programming, is usually much more readable.

Further Agda allows to write

\[
\lambda (x, y :: A) \to s
\]

for

\[
\lambda (x :: A) \to \lambda (y :: A) \to s
\]

similarly for more variables.
The Dependent Function Type in Agda (Cont.)

- Application has the same syntax as in the rules above:
  If \( f :: (x :: A) \to B, a :: A \), then \( f a :: B[x := a] \).

- And we have that \((\lambda(x :: A) \to b) a\) and \(b[x := a]\) are identified.

- A problem is that the \(\eta\)-rule is not implemented in Agda:
  In Agda syntax, the \(\eta\)-rule states that if
  \[
  f :: (x :: A) \to B
  \]
  then
  \[
  f = \lambda(x :: A) \to f x .
  \]
  - It causes run time problems to implement it, since it depends on the type of \(f\).
  - However in some cases the lack of the \(\eta\)-rule causes problems.
The Dependent Product

• Example of a dependent product:

• Let $G$ be the set of genders,

$$G = \{\text{male, female}\}.$$ 

• Let for $g : G$ the type $\text{Names}(g)$ be the collection of names of that gender, eg.

- $\text{Names}(\text{male}) = \{\text{Tom, Jim}\}$,
- $\text{Names}(\text{female}) = \{\text{Jill, Sara}\}$.

• Now the set of names is the set of pairs $\langle g, n \rangle$ s.t. $g$ is a Gender and $n : \text{Names}(g)$.

• This type is written as

$$(g : G) \times \text{Names}(g)$$
Rules of the Dependent Product

**Formation Rule**

\[
A : \text{Type} \quad x : A \Rightarrow B : \text{Type} \\
(x : A) \times B : \text{Type}
\]

**Introduction Rule**

\[
x : A \Rightarrow B : \text{Type} \quad a : A \quad b : B[x := a] \\
\langle a, b \rangle : (x : A) \times B
\]

**Elimination Rules**

\[
c : (x : A) \times B \\
\pi_0(c) : A \\
\pi_1(c) : B[x := \pi_0(c)]
\]

**Equality Rules**

\[
x : A \Rightarrow B : \text{Type} \quad a : A \quad b : B[x := a] \\
\pi_0(\langle a, b \rangle) = a : A
\]

\[
x : A \Rightarrow B : \text{Type} \quad a : A \quad b : B[x := a] \\
\pi_1(\langle a, b \rangle) = b : B[x := a]
\]
Again we have an $\eta$-rule:

$\eta$-Rule

\[
\frac{c : (x : A) \times B}{c = \langle \pi_0(c), \pi_1(c) \rangle : (x : A) \times C}
\]
The Dependent Product in Agda

- In Agda, we have the dependent record type.
  - It is essentially a “labelled product”.

- Assume we have introduced already $A :: \text{Type}$, 
  $a :: A \Rightarrow B(a) :: \text{Type}$.
  Then we can introduce

$$D :: \text{Type}$$
$$= \text{sig}\{a :: A; b :: B(a)\}$$

- If we have $a' :: A$, $b' :: B(c)$, then we can introduce

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

- One can introduce longer records as well, eg.

$$\text{sig}\{a :: A; b :: B; c :: C; d :: D\}$$
• We can now project any element \( d :: D \) down to \( A \) and \( B \):

\[
\begin{align*}
  d.a & :: A \\
  d.b & :: B \ d.a
\end{align*}
\]

• If \( d = \text{struct} \ a = a'; b = b' \), then we have:

\[
\begin{align*}
  c.a & \text{ is equal to } a' \\
  c.b & \text{ is equal to } b'
\end{align*}
\]
• Unfortunately, the dependent product doesn’t behave very well.
  – This is due to the fact that Agda doesn’t support $\eta$-conversion.
  – In this setting $\eta$-equality asserts that if

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

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.
General Remarks

• In order to be in accordance with Agda notation, I will from now on write “Type” instead of “type”.

• The problem with Alfa is now solved.
  – You can use KDE. You will only have to enlarge the menus manually in order to get them operational. (If you use “windowmaker”, this is not necessary.)
  – When you have logged in, open a terminal and type in the following:
    bash
    export PATH=\${PATH}:\~csetzer/Alfa/bin
    alfa
  – Now load one of the example files. In the menu you can always choose one of the items having green buttons.
  – Have fun!!
In the introduction rule for the dependent product, we need an addition premise:

**Introduction Rule**

\[
x : A \Rightarrow B : \text{Type} \quad a : A \quad b : B[x := a] \\
\langle a, b \rangle : (x : A) \times B
\]

Further the equality rules should be written as follows:

**Equality Rules**

\[
x : A \Rightarrow B : \text{Type} \quad a : A \quad b : B[x := a] \\
\pi_0(\langle a, b \rangle) = a : A
\]

\[
x : A \Rightarrow B : \text{Type} \quad a : A \quad b : B[x := a] \\
\pi_1(\langle a, b \rangle) = b : B[x := a]
\]

Then the comment

“(The last two rules assume \( x : A \rightarrow B : \text{Type} \)).”

can be omitted.
- \( \Gamma, \Delta \) denote contexts.  
So \( \Gamma, \Delta \) stand for expressions like

\[
x : A, y : B, z : C
\]

- Notation for contexts: If we have two contexts \( \Gamma, \Gamma' \), then \( \Gamma, \Gamma' \) is the result of concatenating them:

- Eg. if \( \Gamma = x : A, y : B \) and \( \Gamma' = z : C \), then \( \Gamma, \Gamma' = x : A, y : B, z : C \).

  * In fact \( \Gamma' \) is usually not really a context, but a “context piece” — it might depend on \( \Gamma \).

- Similarly, for \( \Gamma \) as before \( \Gamma, z : D \) is the context \( x : A, y : B, z : D \).

- \( \emptyset \) is the empty context (no variables are bound in it).
A non-dependent judgement \( \theta \) (eg. \( A : \text{Set} \)) can be regarded as an abbreviation for \( \emptyset \Rightarrow \theta \).
Additional Judgement $\Gamma \Rightarrow \text{Context}$

- Sometimes we need as assumptions of an axiom the assertion “$\Gamma$ is a valid context”.
  - If $\Gamma$ is $x : A, y : B, z : C$ this would mean
    * $A : \text{Type}$.
    * $x : A \Rightarrow B : \text{Type}$.
    * $x : A, y : B \Rightarrow C : \text{Type}$.
  - We form a new judgement

    $\Gamma \Rightarrow \text{Context}$

    expressing “$\Gamma$ is a valid context”.
The empty context

\[ \emptyset \Rightarrow \text{Context} \]

Extending a context

\[
\frac{\Gamma \Rightarrow A : \text{Type}}{\Gamma, x : A \Rightarrow \text{Context}}
\]

(where in the last rule \( x \) does not occur in \( \Gamma \)).
• In order to derive \( x : A, y : B \Rightarrow C : \text{Type} \) we need to show:
  
  – \( A : \text{Type} \).
  – \( x : A \Rightarrow B : \text{Type} \)

• So the judgement

\[
  x : A, y : B \Rightarrow C : \text{Type}
\]

implicitly contains the judgements

\[
  A : \text{Type} ,
\]

\[
  x : A \Rightarrow B : \text{Type} .
\]

– \( A : \text{Type} \) and \( x : A \Rightarrow B : \text{Type} \) are presuppositions of the judgement

\[
  x : A, y : B \Rightarrow C : \text{Type} .
\]

• The next slide shows the presuppositions of judgements.
<table>
<thead>
<tr>
<th>Presuppositions</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Judgement</strong></td>
<td><strong>Presuppositions</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Gamma, x : A \Rightarrow \text{Context}$</td>
<td>$\Gamma \Rightarrow A : \text{Type}.$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Gamma \Rightarrow A : \text{Type}$</td>
<td>$\Gamma \Rightarrow \text{Context}$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| $\Gamma \Rightarrow A = B : \text{Type}$ | $\Gamma \Rightarrow A : \text{Type}$,  
$\Gamma \Rightarrow B : \text{Type}.$ |
| | |
| $\Gamma \Rightarrow a : A$ | $\Gamma \Rightarrow A : \text{Type}.$ |
| | |
| $\Gamma \Rightarrow a = b : A$ | $\Gamma \Rightarrow a : A$,  
$\Gamma \Rightarrow b : A.$ |
| | |
| $\Gamma \Rightarrow (x : A) \times B : \text{Type}$ | $\Gamma, x : A \Rightarrow B : \text{Type}.$ |
| | |
| $\Gamma \Rightarrow (x : A) \rightarrow B : \text{Type}$ | $\Gamma, x : A \Rightarrow B : \text{Type}.$ |
Presuppositions

- Further 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:
  - \( A : \text{Type} \),
  - \( x : A \Rightarrow B : \text{Type} \),
  - \( x : A, y : B \Rightarrow C : \text{Type} \),
  - \( x : A, y : B, z : C \Rightarrow D : \text{Type} \),
  - \( x : A, y : B \Rightarrow (z : C') \times D : \text{Type} \),
  - \( x : A, y : B \Rightarrow a : (z : C') \times D \),
  - \( x : A, y : B \Rightarrow b : (z : C') \times D \).
Assumption Rule

\[ \Gamma, x : A, \Gamma' \Rightarrow \text{Context} \]
\[ \Gamma, x : A, \Gamma' \Rightarrow x : A \]
• When we define a function:

\[
f \ (a::A) \\
\quad :: B \\
\quad = \ \{! \ \! \}
\]

we can make use of \( a :: A \) when solving the goal \( \{! \ ! \} \).

– This is an application of the assumption rule: When solving \( \{! \ ! \} \) we essentially define under the assumption \( a :: A \) an element \( \{! \! \} :: B \).
• 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

\[
\frac{a :: A \Rightarrow \{! !\} :: B}{\lambda (a :: A) \rightarrow \{! !\} :: A \rightarrow B}
\]

So we have to solve

\[
a :: A \Rightarrow \{! !\} :: B
\]

in order to derive

\[
\lambda (a :: A) \rightarrow \{! !\} :: A \rightarrow B
\]
Weakening Rule

\[
\Gamma \Rightarrow A : \text{Type} \quad \frac{\Gamma, \Gamma' \Rightarrow \theta}{\Gamma, x : A, \Gamma' \Rightarrow \theta}
\]

- \( x \) shouldn’t be bound in \( \Gamma, \Gamma' \).

- Exercise: find counter examples why
  - \( x \) shouldn't occur in \( \Gamma \),
  - and \( x \) shouldn’t occur in \( \Gamma' \).

- Remark: One can in fact show that the Thinning rule can be \textit{weakly derived}.
  - \textit{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 derive from the premise the conclusion directly.
In the following we always allow to add a common context to rules.

- Applies as well to previous rules except (because of the variable condition) of the context and weakening rules.

Example: when stating that we have the rule

\[ A : \text{Type} \quad x : A \Rightarrow B : \text{Type} \]
\[ (x : A) \rightarrow B : \text{Type} \]

we implicitly mean the more general rule

\[ \Gamma \Rightarrow A : \text{Type} \quad \Gamma, x : A \Rightarrow B : \text{Type} \]
\[ \Gamma \Rightarrow (x : A) \rightarrow B : \text{Type} \]
Common Contexts

- There is only one exception: axioms, i.e. rules without premisses.

- If we have an axiom

  \[ \theta \]

  the version weakened by \( \Gamma \) requires the additional assumption \( \Gamma \Rightarrow \text{Context} : \)

  \[ \begin{array}{c}
  \Gamma \Rightarrow \text{Context} \\
  \hline
  \Gamma \Rightarrow \theta
  \end{array} \]
General Equality Rules

**Reflexivity**

\[
\begin{align*}
A &: \text{Type} \\
A &= A &: \text{Type}
\end{align*}
\]

\[
\begin{align*}
a &: A \\
a &= a &: A
\end{align*}
\]

**Symmetry**

\[
\begin{align*}
A &= B &: \text{Type} \\
B &= A &: \text{Type}
\end{align*}
\]

\[
\begin{align*}
a &= b &: A \\
b &= a &: A
\end{align*}
\]

**Transitivity**

\[
\begin{align*}
A &= B &: \text{Type} & B &= C &: \text{Type} \\
A &= C &: \text{Type}
\end{align*}
\]

\[
\begin{align*}
a &= b &: A & b &= c &: A \\
a &= c &: A
\end{align*}
\]

(Reflexivity can be weakly derived).
General Equality Rules (Cont.)

Transfer

\[
\frac{a : A \quad A = B : \text{Type}}{a : B}
\]

\[
\frac{a = b : A \quad A = B : \text{Type}}{a = b : B}
\]
Equality Versions of Rules

For all formation/introduction/elimination rules in the following, we have as well equality versions of it. This means for instance in case of the dependent function type, that we have the following rules:

**Equality version of the formation rule**

\[
A = A' : \text{Type} \quad x : A \Rightarrow B = B' : \text{Type} \quad (x : A) \rightarrow B = (x : A') \rightarrow B' : \text{Type}
\]

**Equality version of the introduction rule**

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

**Equality version of the elimination rule**

\[
a = a' : A \quad f = f' : (x : A) \rightarrow B \quad f a = f' a' : B\[x := a]\]

As another example, the equality versions of the rules for the dependent product are:

**Equality version of the formation rule**

\[
\frac{A = A' : \text{Type} \quad x : A \Rightarrow B = B' : \text{Type}}{(x : A) \times B = (x : A') \times B' : \text{Type}}
\]

**Equality version of the introduction rule**

\[
\frac{x : A \Rightarrow B : \text{Type} \quad a = a : A \quad b = b' : B[x : a]}{(a, b) = (a', b') : (x : A) \times B}
\]

**Equality version of the elimination rules**

\[
\frac{c = c' : (x : A) \times B}{\pi_0(c) = \pi_0(c') : A}
\]

\[
\frac{c = c' : (x : A) \times B}{\pi_1(c) = \pi_1(c') : B[x := \pi_0(c)]}
\]
Substitution Rules

The following rules can be weakly derived:

**Substitution1**

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow \theta \quad \Gamma \Rightarrow a : A}{\Gamma, \Gamma'[x := a] \Rightarrow \theta[x := a]}
\]

(\(\Gamma'[x := a]\) is the result of substituting in \(\Gamma'\) all occurrences of \(x\) by \(a\)).

**Substitution2**

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow B : \text{Type} \quad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow B[x := a] = B[x := a'] : \text{Type}}
\]

**Substitution3**

\[
\frac{\Gamma, x : A, \Gamma' \Rightarrow b : B \quad \Gamma \Rightarrow a = a' : A}{\Gamma, \Gamma'[x := a] \Rightarrow b[x := a] = b[x := a'] : B[x := a]}\]
• We would like to add operations on types, such as

\[ \text{prod} : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} \]

which should take two types and form the product of it.

• The problem is that for this we need

\[ \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} : \text{Type} \]

and our rules allow this only if we have

\[ \text{Type} : \text{Type} \]

Adding this as a rule results however in an inconsistent theory, i.e. from this we can prove everything, especially false formulas. The corresponding paradox is called \textbf{Girard's paradox}. 
Set \rightarrow \text{Set} \quad \times \text{Set}

\text{N} \rightarrow \text{N} \quad \times \text{N}

\text{N} \rightarrow \text{Set}

\text{Type}

\text{Set}
Set (Cont.)

• Instead we introduce a new type

\[ \text{Set} : \text{Type} \]

Set is the type of sets.

– A set is a small type.

• We add rules that if \( A : \text{Set} \) then \( A : \text{Type} \).

– Since \( \text{Set} : \text{Type} \) we get

\[ \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} : \text{Type} \]

and we can assign to prod above a type

\[ \text{prod} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \]

(Of course \( \text{prod} \) needs to be defined – this will be done below.)
– However we cannot use \( \text{prod} \) in order to form the product of two sets, i.e. cannot introduce

\[
\text{prod Set Set : Set ,}
\]

since \( \text{Set : Set} \) does not hold.

\( \ast \) That would result in the same inconsistency as \( \text{Type : Type} \).
Rules for Set

Formation Rule for Set

Set : Type

Every Set is a Type

\[
\frac{A : \text{Set}}{A : \text{Type}}
\]

\[
\frac{A = B : \text{Set}}{A = B : \text{Type}}
\]

Closure of Set under the dependent product

\[
\frac{A : \text{Set}}{(x : A) \times B : \text{Set}}
\]

Closure of Set under the dependent function type

\[
\frac{A : \text{Set}}{(x : A) \rightarrow B : \text{Set}}
\]
Example: prod

We can now introduce \( \text{prod} : \text{Set} \to \text{Set} \to \text{Set} \):

First we derive \( X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set} \):

\[
\frac{\text{Set} : \text{Type}}{X : \text{Set} \Rightarrow \text{Context}}
\]

\[
\frac{X : \text{Set} \Rightarrow \text{Set} : \text{Type}}{X : \text{Set}, Y : \text{Set} \Rightarrow \text{Context}}
\]

\[
\frac{X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set}}{X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set}}
\]

Using this we can derive

\[
X : \text{Set}, Y : \text{Set}, x : X \Rightarrow Y : \text{Set}
\]

as follows:

\[
\frac{X : \text{Set}, Y : \text{Set} \Rightarrow X : \text{Set}}{X : \text{Set}, Y : \text{Set}, x : X \Rightarrow \text{Context}}
\]

\[
\frac{X : \text{Set}, Y : \text{Set}, x : X \Rightarrow \text{Context}}{X : \text{Set}, Y : \text{Set}, x : X \Rightarrow Y : \text{Set}}
\]
Now we can derive our desired judgement:

\[
\begin{align*}
X &: Set, Y &: Set \Rightarrow X &: Set \\
X &: Set, Y &: Set &\Rightarrow (x &: X) \times Y &: Set \\
X &: Set &\Rightarrow \lambda Y. (x &: X) \times Y &: Set \rightarrow Set \\
\lambda X, Y. (x &: X) \times Y &: Set &\rightarrow Set \rightarrow Set
\end{align*}
\]

So define

\[
\text{prod} := \lambda X, Y. (x &: X) \times Y
\]