B2. The Logical Framework

(a) Basic Form of Rules
(b) The non-dependent function type and product.
(c) The dependent function type and product.
(d) Structural rules.
Four kinds of rules:

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

Additionally, there are equality versions of the formation, introduction, and elimination rules for each type construction we have usually 4 kinds of rules.
The formation rules introduce new types. Each type construction has one such rule. The conclusion of such a rule will have the form:

\[ C(a_1; \ldots ; a_n) : \text{Type} \]

- \( a_1, \ldots , a_n \) are its arguments.
- Where \( C \) is a type constructor.
Example 1: The Type of Lists

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

\[
\frac{A : \text{Type} \quad \text{List}(A) \text{ : Type}}{A : \text{Type}}
\]
Example 2: The Type of Natural Numbers

Formation rule for the type of natural numbers:

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

Later we will see that we can replace in this example \( \text{Type} \) by \( \text{Set} \).
Example 3: The Non-Dependent Product

Formation rule for the non-dependent product:

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

- The type-constructor is \((\times)(\forall B.)(A, B)\).
- \(A \times B\) stands for \((A)(\forall B.)(A, B)\).
Formation of a type is usually done by introducing a constant of a certain type.

Example 1:

```
\ldots =

\text{Type} :: \text{Type}

\text{List} :: (\forall \text{Type}) \text{Type}
```

The formation of a type is usually done by introducing a constant of a certain type in Agda.
Agda syntax for introducing the non-dependent product:

Example 2: \((\times)\)
Traditionally onewayrite in

- type-constructors in

uncurried form.

• We have to write \texttt{List \langle A \rangle},

- \texttt{List} alone does not make sense as a term.

• Traditionally one writes in \textit{type theory} type-constructors in uncurried form.

\begin{itemize}
  \item \texttt{List :: Type \rightarrow Type}

\end{itemize}

(\text{more precisely this is kind and not a type})

- So \texttt{List} alone is a term and has type

\begin{itemize}
  \item function type (are always curried).
  \item \texttt{in Agda} type constructors (except those predefined for \texttt{dep}, \texttt{product} and

\end{itemize}

\begin{itemize}
  \item \texttt{List \ A} means the application of the function \texttt{List} to \texttt{A}.
  \item We write therefore \texttt{List \ A} and not \texttt{List \langle A \rangle}.

\end{itemize}
Currying in Agda (cont.)

Agda allows to write $A \times B$ for $(\times) A B$.

The latter is the operation which takes a $B$ and returns $N \times B$.

\[
\begin{align*}
\text{Type} & \leftarrow \text{Type} :: N (\times) \\
\text{Type} & \leftarrow \text{Type} :: (\times)
\end{align*}
\]

- $N$ are terms and have types (more precisely kinds)
The introduction rule introduces elements of a type.

\[ C(\ldots) \]

Where

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

The conclusion of such a rule will have the form

\[ C(\ldots) \]

The introduction rule introduces elements of a type.
Lists of type $A$ have two introduction rules:

- In case of the rule for $\text{nil}$, this premise is implicit in the premise $a : A$. 

\[
\frac{a : A}{\text{List}(a) : (\forall A. \text{Type}(A))}
\]

- Guarantee that we can form the type $\text{List}(A)$.

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


\[
\frac{a : A \quad \text{List}(a) : (\forall A. \text{Type}(A))}{\text{cons}(a, l) : \text{List}(A)}
\]

- In case of the rule for $\text{nil}$, we needed the premise $A : \text{Type}$ in order to form the type $	ext{List}(A)$.

\[
\frac{\text{List}(a) : (\forall A. \text{Type}(A))}{a : A}
\]
Example 2: Natural Numbers.

The natural numbers \( \mathbb{N} \) can be considered as being formed from two operations:

\[
\begin{align*}
\mathbb{N} : & (u) S \\
\mathbb{N} : & u \\
\mathbb{N} : & 0
\end{align*}
\]

Using these two operations we can form 0, \( S(0) = 1 \), \( S(1) = 2 \), \( S(2) = 3 \), \( S(3) = 4 \), \( S(4) = 5 \), \( S(5) = 6 \), \( S(6) = 7 \), \( S(7) = 8 \), \( S(8) = 9 \), \( S(9) = 10 \), \( \cdots \) and

The introduction rules of \( \mathbb{N} \) are:

- The constructors of \( \mathbb{N} \) are 0 and \( S \).
- Therefore all natural numbers.

The introduction rules of \( \mathbb{N} \) are:

\[
\begin{align*}
& 0 : \mathbb{N} \\
& n : \mathbb{N} \Rightarrow S(n) : \mathbb{N}
\end{align*}
\]

- Where \( S \) where \( S(0) = 1 \), \( S(1) = 2 \), \( S(2) = 3 \), \( S(3) = 4 \), \( S(4) = 5 \), \( S(5) = 6 \), \( S(6) = 7 \), \( S(7) = 8 \), \( S(8) = 9 \), \( S(9) = 10 \), \( \cdots \)
Canonical elements of a type are those introduced by an introduction rule.

- Constructor + 1, cons(cons(0, nil), nil) in case of List(N).
- nil, cons(1, cons(cons(0, nil), nil)) in case of List(N).
- ((0))(0) and 3 for S(S(S(0))) in case of N.

* Here 2 stands for S(S(0)) and 3 for S(S(S(S(0)))) in case of N.

Examples:

- Canonical elements therefore always start with a constructor.
- Canonical elements of a type are those introduced by an introduction rule.
Terms can usually be reduced further. 

Example: Using the above reduction we obtain:

Further, reductions can be applied to subterms.

- 
  - \( \text{concat}(\text{cons}(2, \text{nil}), \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: Terms can usually be reduced further.

- \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)

- Example: \( \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \rightarrow \text{concat}(\text{cons}(2, \text{nil}), \text{cons}(3, \text{nil}))) \)
The reduction rules for addition on $\mathbb{N}$ are:

- $n + 0 \rightarrow n$,
- $n + S(m) \rightarrow S(n + m)$.

Example:

$0 + S(0) \rightarrow S(0 + 0) \rightarrow S(0)$ is one reduction built from two one-step-reductions.

Reductions can be formed by a sequence of one-step-reductions.

These (and reductions using them in subterms) are the one-step reductions.

The reduction rules for addition on $\mathbb{N}$ are:

- $m + u \rightarrow S(m) + u$,
- $u \rightarrow 0 + u$.
Canonical Elements and Reductions

Canonicalelementscanonlybereducedfurtherbyreducingtheconstructor.

{Theconstructorwillalwaysremaininplace.

Forinstance

\[ (\text{S} (\text{S} (\text{S} (\text{S} (\text{S} 0)))))) \]

\[ \text{reducesto} \]

\[ (\text{S} (\text{S} (\text{S} (\text{S} (\text{S} 0))))) \]

or

\[ \text{S} (\text{S} (\text{S} (\text{S} (\text{S} 0)))) \].

Thisreductioncanbeformedfromonestepreductionsasfollows:

– Forinstance \( \text{S} (\text{S} (\text{S} (\text{S} (\text{S} 0)))) \)

– Theconstructorwillalwaysremaininplace.

– Canonical elements can only be reduced further by reducing the arguments.
Independent of each other.

- Further, the arguments of the constructor reduce individually and

  \[ \text{cons}(x, \text{cons}(y, \text{cons}(z, \text{nil}))) \rightsquigarrow \text{cons}(x) \]

  \[ \text{cons}(1, \text{cons}(2, \text{cons}(3, \text{cons}(4, \text{cons}(5, \text{cons}(6, \text{nil})))))) \]

Similarly

- It cannot change later to 0.
  - This information will remain as it is when further reducing it.
  - Once we have determined that we have the “the successor of something”

  \[ S \]

  The outermost S always remains in place.
Any element of a type has to reduce to a canonical element of it.

- Again the outermost operator changes to cons.
  - constructor (concat (cons (2, nil), nil)) → cons (2, nil).

- When reducing \( 2 + 3 \), the outermost operator changes to S.
  - Note that \( 2 + 3 \) is only a more readable way of writing \( (+) \) \((2,3)\):
  - \( 2 + 3 = SSSSSSSSSSS = SS = 5 \)
  - \( \text{concat} (\text{cons} (2, \text{nil}), \text{nil}) \) is a non-canonical element, and concat is not a constructor.

- \( 2 + 3 \) is a non-canonical element, and \((+)\) is not a constructor.

Constructors and Canonical Elements (Cont.)
In Agda, constructors are curried:

\[
\begin{align*}
\text{Nil} & \leftarrow \text{List} N \\
\text{cons}(N :: u) & \leftarrow (\text{List} N) :: \text{List} N
\end{align*}
\]

We have:

- **As type-constructors**, in Agda, constructors are curried:

- If \( A \) can be inferred automatically, we can replace the above by \( C(A) \).

- In Agda, the constructor \( C \) of type \( A \) is written as \( C(A) \).
Sincenotationslike nil (ListN) isusuallytocumbersome,itisbettertoIntroduceabbreviations:

\[
\begin{align*}
\text{nil} &:: \text{ListN} = \text{nil} @ \\
\text{cons} &:: \text{ListN} :: \text{ListN} = \text{cons} @ \nline
\end{align*}
\]

Notethattheaboveintroduces\ nil,\ \text{cons}for\ \text{ListN},\ and\ \textbf{not}\ \textbf{for}\ \textbf{the}\ \textbf{general}\ \textbf{case}\ \text{List} A \text{for}\ \textbf{any}\ \text{type} \ A.\ \text{(That}\ \text{would}\ \text{require}\ \text{an}\ \text{extra}\ \text{argument}}\ A :: \text{Type}.

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

\[\text{general\ case\ \text{List} A \text{for}\ \textbf{any}\ \text{type} \ A.}\]

\[\text{Note\ that\ the\ above\ introduces\ \text{nil,} \ \text{cons}\ \text{for\ \text{ListN},\ and\ \textbf{not\ for\ the}}\ \textbf{general}\ \textbf{case}\ \text{List} A \text{for}\ \textbf{any}\ \text{type} \ A.\ \text{(That}\ \text{would}\ \text{require}\ \text{an}\ \text{extra}\ \text{argument}}\ A :: \text{Type}.\]

\[\text{Introduce\ abbreviations:}\]

\[\begin{align*}
\text{nil} &:: \text{ListN} = \text{nil} @ \\
\text{cons} &:: \text{ListN} :: \text{ListN} = \text{cons} @ \nline
\end{align*}\]

\[\text{Since\ notations\ like\ \text{nil} (\text{ListN})\ \text{is\ usually}\ \text{to\ cumbersome,\ it\ is\ better\ to}}\]

\[\text{Constructors in Agda (Cont.)}\]
Elimination rules allow to take an element of a type and compute from it another type.

Equality rules will express:

\[ q = (\langle q', a \rangle)_{0,1} = (\langle q', a \rangle)_{1,0} \]

- First and second projection of a product:

\[
\frac{B \colon (c)_{1,0}}{B \times A \colon c}
\]

\[
\frac{A \colon (c)_{0,1}}{B \times A \colon c}
\]

Example 1: First and second projection of a product:

Elimination rules will express

\[ q = (\langle q', a \rangle)_{0,1} = (\langle q', a \rangle)_{1,0} \]
Example 2: Addition in \( \mathbb{N} \)

- Recursivites on.
- Introduce all functions we expect to be definable, including all primitive.

Equality rules will express:

\[
\begin{align*}
\mathbb{N} : w + 0 &= w \\
\mathbb{N} : w + S(m) &= S(w + m) \\
\end{align*}
\]

Proceeding like this would require one elimination rule for each function.

Instead we will introduce one general elimination rule which allows to

from \( \mathbb{N} \) we want to define.

- Elimination rule expresses.

\[
\begin{align*}
\mathbb{N} : w + u & = (w)(S + u) \\
u & = 0 + u \\
\end{align*}
\]
Elimination rules invert the introduction rules.

- Reduce $a$ (which is of type $A$) to its canonical form.
- This element must be of the form $\langle a', q \rangle$.
- Reduce $c$ to a canonical element.

Therefore, if $c : A \times B$, the canonical form of $\langle 0(c) \rangle$ can be computed as follows:

\[ \begin{align*}
\text{Reduce } c \text{ to a canonical element.} \\
\text{This element must be of the form } \langle a', q \rangle. \\
\text{Reduce } a \text{ (which is of type } A) \text{ to its canonical form.}
\end{align*} \]

- A non-canonical element of type $A \times B$ must reduce to a canonical element.
- In case of $A \times B$, the canonical elements are of the form $\langle a', q \rangle$ for $a : A'$.
Elimination in Agda

Elimination for built-in types has special notation.

Example: Definition of addition in \( \mathbb{N} \):

\[
\begin{align*}
\text{\{\(\text{\(\text{\(m + n\)}}\)}\}} \rightarrow (\text{\(\text{\(m\)}}\} \\
\text{\(n\}} \rightarrow (\text{\(\text{\(Z\)}}) \\
\text{\} \text{\{case m of \} =} \\
\text{\(N::} \\
\text{\(\text{\(N::} m, n\)}\) \rightarrow (+) \\
\end{align*}
\]

For user-defined types, elimination is realized by case distinction.

For built-in types, elimination has special notation.
For instance in case of \( A \times B \), (Red) are the reductions:

- Reduce it to canonical form.

  Type:
  - The result will be a canonical or non-canonical element of the target.
  - Then make one reduction step (Red).
  - Reduce the element to be eliminated to canonical form.

 can be always computed as follows:

The canonical element for an element, which is the result of an elimination,

\[
q \leftarrow (q', a) \\
q \leftarrow (q', a)
\]
The result is already in canonical form.

\[
((0 + (0)s) + 0)s \leftarrow (0 + (0)s)s + 0 \leftarrow ((0)s + (0)s) + 0 = (1 + 1) + 0
\]

So the computation of \((1 + 1) \cdot 0\) is as follows:

- 
  
  "Eliminating" 

  Note that the second argument is the argument which we are 

  \[
  (w + u)s \leftarrow (w)s + u \leftarrow u \leftarrow 0 + u 
  \]

  In case of \((+)(\text{Red})\) are the reductions

Equality Rules (Cont.)
Equality rules express (Red) type theoretically. Equality rules express (Red) type theoretically. 

They describe what happens if one first introduces an element and then immediately eliminates it.
The equality rule explains how to reduce that element (namely to $a : A$).

So it is derived by first introducing and then eliminating it immediately.

\[
\begin{align*}
\forall : (\langle q, \nu \rangle)^0 \\
B \times \forall : \langle q, \nu \rangle \\
\forall \vdash q \quad \forall : a
\end{align*}
\]

In the first judgment we can derive $\forall : (\langle q, \nu \rangle)^0 \nu$ as follows:

\[
\begin{align*}
\forall : \nu = (\langle q, \nu \rangle)^0 \\
B \vdash q \quad \forall : a
\end{align*}
\]

Equality rules for $B \times A$.  

**Example (Equality Rule)**
The second equality rule for \( x \) is similar:

\[
\frac{B : q = (\langle q, a \rangle)_{\langle \perp \rangle}}{B : q} \quad A : a
\]

Example (Equality Rule, Cont)
The equality rule explains how to reduce $n + 0$.

\[
\begin{align*}
\frac{N : 0 + u}{N : 0 \quad N : u}
\end{align*}
\]

The right side is an axiom, the left side has to be concluded using some derivation. + can be derived by first introducing it and then by eliminating it.

\[
\begin{align*}
\frac{N : u = 0 + u}{N : u}
\end{align*}
\]
These second equality rule for + is as follows:

\[
\frac{N : (m)S + u}{N : (m)S} \quad \frac{N : u}{N : u}
\]

\[N : u + (m)S + u \quad \pm \]

The second equality rule for + is as follows:

Example 3 (Equality Rule)
Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.

Equality Rules in Agda are implicit.
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 \( N \) (degenerated):

\[
\begin{align*}
\text{Type} : N &= N \\
\frac{(B) \text{ List} = (\forall) \text{ List}}{	ext{Type} : B = A}
\end{align*}
\]
Example: Equality version of the elimination rule for (+), \( \mathbb{N} \):

\[
\begin{align*}
\forall (\forall \text{List}) : & (n', a) \text{cons} = (l', a) \text{cons} \\
\forall (\forall \text{List}) : & n = n' \\
\forall (\forall \text{Type}) : & a = a
\end{align*}
\]

\[
\forall \text{Type} : \text{nil} = \text{nil}
\]

degenerated.

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

\[
\begin{align*}
\forall (\forall \text{List}) : & (\forall \text{Type})
\end{align*}
\]
Equality versions of the rules in questions can be formed in a straightforward way, once one knows the non-equality version. In Agda they are implicit (part of the reduction machinery).

- We will often not mention them.

- The equality versions of the rules in questions can be formed in a straightforward way.
The Non-Dependent Product Rules

Introduction Rule

Elimination Rule

Equality Rules

Formation Rule

Rules of the Non-Dependent Product

\( B : q = (\langle q', q \rangle)^\nu \)
\( B : q \quad A : q \)
\( B : q \quad A : q \)

\( A : q = (\langle q', q \rangle)^0 \)
\( B : q \quad A : q \)
\( B : q \quad A : q \)

\( B : (\langle c \rangle)^\nu \)
\( B : c \quad A : c \)
\( B : c \quad A : c \)

\( B : (\langle c \rangle)^0 \)
\( B : c \quad A : c \)
\( B : c \quad A : c \)

\( B : (\langle c \rangle)^\nu \)
\( B \times A : \langle c \rangle \)
\( B \times A : \langle c \rangle \)

\( B \times A : \langle c \rangle \)
\( B : q \quad A : q \)
\( B : q \quad A : q \)

\( B : \text{Type} \quad A : \text{Type} \quad B : \text{Type} \quad A : \text{Type} \)

\( B : \text{Type} \quad A : \text{Type} \quad B : \text{Type} \quad A : \text{Type} \)
This rule does not fit into the above schema: The \( \eta \)-Rule
The \( n \)-rule expresses that any element of \( B \times A \) is of the form \( B \times A \).
makings use of the \( \iota \)-rule.

\[
\langle (x)(\iota^1, x)(\iota^0) \rangle = x
\]

(\( so \( x \) is just a variable, \( we \) cannot derive that \( x \) without)

\[
B \times A : x \iff B \times A : x
\]

\( x \) is just a variable, we cannot derive that \( x \) without making use of the \( \iota \)-rule.

However, if we assume an element of type \( A \times B \), \( x \) is just a variable, we cannot derive that \( x \) without making use of the \( \iota \)-rule. 

For elements of \( A \times B \) introduced by an introduction rule, we don’t need the \( \iota \)-rule.

\( \text{The } \iota \text{-Rule (Cont.)} \)
Equality Versions of the Rules

Equality Version of the Formation Rule

\[ A \equiv A_0 : \text{Type} \]

\[ B \equiv B_0 : \text{Type} \]

\[ A \equiv B_0 : \text{Type} \]

Equality Version of the Introduction Rule

\[ a \equiv a_0 : A \]

\[ b \equiv b_0 : B \]

\[ h \equiv h_0 ; b_0 : A \]

Equality Versions of the Elimination Rules

\[ B \times A : \langle q, r \rangle = \langle q', r \rangle \]

\[ B : q = q \]

\[ A : r = r \]

Equality Version of the Terminator Rule

\[ B \times A = B \times A \]

Equality Version of the Formation Rule

\[ B \equiv B_0 : \text{Type} \]

\[ A \equiv A_0 : \text{Type} \]

Equality Versions of the Rules

(C) Anton Setzer 2003 (except for pictures)
the term \( a \) (after some renaming of bounded variables).

Here \( [\nu =: x]q \) is the result of substituting in \( q \) every occurrence of variable \( x \) by

\[
\frac{B : [\nu =: x]q = \nu \left(q \left( \forall x : \chi \right) \chi \right)}{\forall : \nu \quad B : q \Leftarrow \forall : x}
\]

Equality Rule

\[
\frac{B : \nu \, f}{\forall : \nu \quad B \Leftarrow \forall : f}
\]

Elimination Rule

\[
\frac{B \Leftarrow \forall : q \left( \forall x : \chi \right) \chi}{B : q \Leftarrow \forall : x}
\]

Introduction Rule

\[
\frac{\exists \nu \, B \Leftarrow \forall : \nu \exists \nu \, B}{\exists \nu : B \Leftarrow \forall : \nu \exists \nu : B}
\]

Formation Rule
The reduction corresponding to the equality rule is often called $\beta$-reduction.

\[ \left[ \varphi =: x \right] \varphi \leftarrow \varphi \left( \forall : x \right) \]

- As a reduction, it reads:
- Greek letter spelled "beta".
- $\beta$ - $\beta$-reduction.
Again this rule does not fit into the above schema:

\[
\frac{\forall x : \mathcal{A} \cdot \forall x : \mathcal{A} \chi = f}{\forall x : \mathcal{A}} \quad \frac{\forall x : \mathcal{A}}{\forall x : \mathcal{A}}
\]
The $\nu$-rule expresses that any element of $A!B$ is of the form $(x:A)\nu x \gamma$. So the conclusion of the $\nu$-rule can be derived without using the $\nu$-rule.

\[
f = \nu x \gamma \quad = \quad [x := x] \nu x \gamma \quad = \quad (x(\nu x \gamma)) \cdot x \gamma = x f. x \gamma
\]

On the other hand, if we have $f$ of this form, e.g. $x f = \nu x \gamma$ then we get

- \text{some\,thing:} \quad x f = \nu x \gamma

- If $B \leftarrow A : f$ and we have the $\nu$-rule, then this follows with

$$\gamma(x : A) \nu x \gamma \quad \text{some\,thing:} \quad \text{The $\nu$-rule expresses that any element of $B \leftarrow A$ is of the form}$$
The $\forall$-Rule (Cont.)

For elements of $A \vdash B$ introduced by an introduction rule, we don't need making use of the $\forall$-rule.

However, if we assume an element of type $A \vdash B$, e.g. state $f$ is just a variable, we cannot derive that $f = \chi$.

$$B \leftarrow \forall : f \Rightarrow B \leftarrow \forall : f$$

(C) Anton Setzer 2003 (except for pictures)
Equality Version of the Formation Rule

\[
\begin{align*}
\mathcal{B} & \vdash \varphi, f = a \ f \\
\forall & \vdash \varphi = a & \mathcal{B} & \vdash \forall : \varphi = f \\
\end{align*}
\]

Equality Version of the Elimination Rule

\[
\begin{align*}
\mathcal{B} & \vdash \forall : \varphi \ (\forall : x) \chi = q \ (\forall : x) \chi \\
\mathcal{B} & \vdash \varphi = q & \vdash \forall : \chi \\
\end{align*}
\]

Equality Version of the Introduction Rule

\[
\begin{align*}
\text{def} & \vdash \mathcal{B} & \vdash \forall = \mathcal{B} & \vdash \forall \\
\text{def} & \vdash \mathcal{B} = \mathcal{B} & \vdash \forall = \forall \\
\end{align*}
\]
The introduction rule requires an extra premise \( \forall x : A \Rightarrow B \) which is not implied by the other premises.

**Introduction Rule**

\[
\frac{B \times (\forall x : A) : \langle q, a \rangle \quad [a =: x]B : q 
\quad \forall : a 
\quad \text{Type} : B \iff \forall : x}{\forall \text{Type} : B \iff \forall : x}
\]

**Formation Rule**

\[
\frac{\text{Type} : B \times (\forall x) \quad \text{Type} : B \iff \forall : x}{\forall \text{Type} : B \iff \forall : x}
\]

**Rules of the Dependent Product**

(c) Antón Setzer 2003 (except for pictures)
In the last introduction rule, an extra premise was required. This is necessary in order to guarantee that we can form the type $\forall x : A \rightarrow B$. In case of the non-dependent product, this premise was not necessary:

$\forall A : B \rightarrow C, A \rightarrow B \rightarrow C \rightarrow D$.

From which it follows $\forall A : B \rightarrow C : (A \rightarrow B) \rightarrow C \rightarrow D$.
Rules of the Dependent Product (Cont.)

**Elimination Rules**

\[
\begin{align*}
\forall x : A \vdash B & \iff \forall x : x : A \vdash B \\
\forall x : A \vdash B & \iff \forall x : x : A \vdash B \\
\forall x : A \vdash B & \iff \forall x : x : A \vdash B \\
\end{align*}
\]

*Critical note:* The last two rules require the extra premise \( x : A \) \( \vdash B \) (which is not implied by the premises).

---

(C) Anton Setzer 2003 (except for pictures)
We have the following \( \eta \)-rule:

\[
\begin{array}{c}
\forall x : A \rightarrow (\forall x : A) \times B \times (\forall x : A) = c \\
\hline
\end{array}
\]

Rules of the Dependent Product (Cont.)
Equality Version of the Formation Rule

\[ \{(c)0,\nu = x\}B : (\rho)1,\nu = (c)1,\nu \quad \Rightarrow \quad B \times (V : x) : \rho = c \]

Equality Version of the Elimination Rules

\[ B \times (V : x) : \langle q, p \rangle = \langle q, p \rangle \quad \Rightarrow \quad \begin{array}{l}
[ p = x]B : q = q \\
x : p = a \\
L\text{ype} : B \Leftarrow V : x
\end{array} \]

Equality Version of the Introduction Rule

\[ L\text{ype} : B \times (V : x) = B \times (V : x) \quad \Rightarrow \quad \begin{array}{l}
L\text{ype} : B = B \\
L\text{ype} : V = V
\end{array} \]
In Agda, we have the dependent record type.

\{ B :: q :: A; a :: A \} = D :: Type

Then we can introduce

Assume we have introduced already A :: Type, a :: Type, B :: Type

– It is essentially a "labelled product."

In Agda, we have the dependent record type.

The Dependent Product in Agda
\{ f \cdot q :: A ; q :: B \mid c :: C \}\leq f

One can introduce longer records as well. E.g.

\[ D :: \{ q = q ; p = q \} \quad \text{strict} = \]

\[ c :: D \]

If we have \( d :: A \), \( q :: B \) then we can introduce

The Dependent Product in Agda (Cont.)
Wecannowprojectanyelement\(d::D\)asabovetodown\(A\)and\(B\):
\[\begin{align*}
c\cdot a & \text{isequal to} \quad q \\
c\cdot a & \text{isequal to} \quad p \\
\{q = q; p = p\} & \text{thenwehave:} \\
[v \cdot p = v] & A \\ & \vdash q \cdot p \\
A & \vdash v \cdot p
\end{align*}\]

If \(c\) is \textit{strict} = \{\}

\(V\) can now project any element \(A\) as above down to \(A\) and\(B\):

\textbf{The Dependent Product in Agda (Cont.)}
Unfortunately, the dependent product does not behave very well.

\{(q :: a :: A \vdash q = c :: a :: A) \mid q = c \}

In most cases one can avoid this by using the inductively defined \(\aleph\)-type,

\{(p :: B) \mid A \vdash q \}

which will be treated later.

– In this setting \(\aleph\)-equality asserts that if

– This is due to the fact that Agda doesn’t support the \(\aleph\)-rule.

The Dependent Product in Agda (cont.)
Rules of the Dependent Function Type

**Equality Rule**

\[
\frac{[a =: x] B : [a =: x] q = a \cdot (\forall x : x)Y}{\forall : a \quad B : q \Leftarrow \forall : \tilde{Y}}
\]

**Elimination Rule**

\[
\frac{[a =: x] B : a f}{\forall : a \quad B \Leftarrow (\forall x : x) : f}
\]

**Introduction Rule**

\[
\frac{B \Leftarrow (\forall x : x) : q \cdot (\forall x : x)Y}{B : q \Leftarrow \forall : x}
\]

**Formation Rule**

\[
\frac{\alpha \perp \beta \vdash B \Leftarrow (\forall x)\alpha \perp \beta \vdash \forall : x \quad \alpha \perp \beta \vdash \forall : x}{\alpha \perp \beta \vdash B \Leftarrow \forall : x}
\]
The $u$-Rule has a special status:

$$\forall x : (\forall x : \text{something}).$$

Again the $u$-rule cannot be derived if the element in question is a variable.

Again the $u$-rule expresses that every element of $B$ is of the form $\forall x : \text{something}$. $\forall x : \text{something}$.

Finally the $u$-rule expresses that every element of $B$ is of the form $\forall x : f$.

The $u$-Rule
Further terms which differ in the choice of bounded variables are identified:

- Called α-equivalence ($\alpha = \text{Greek letter spelled alpha}$).
Equality Version of the Formation Rule

\[ A = A_0 : Type \]
\[ B = B_0 : Type \]
\[ ! : B = (x : A_0) \]

Equality Version of the Introduction Rule

\[ x : A \]
\[ b = b_0 : B \]
\[ ! : b = b_0 : (x : A) \]

Equality Version of the Elimination Rule

\[ f = f_0 : (x : A) \]
\[ ! : f = f_0 : A \]
\[ ! : f(a) = f_0(a) : B \]

Equality Versions of the above Rules

\[ \forall : p = q \]
\[ \forall : p = q \]
\[ \forall : p = q \]

Equality Version of the Introduction Rule

\[ p = q \]
\[ ! : p = q \leq x : A \]

Equality Version of the Elimination Rule

\[ p = q \]
\[ ! : p = q \leq x : A \]

Equality Version of the Formation Rule

\[ B \]
\[ (x : A) \]
\[ B = B_0 \]
\[ \forall : B = B_0 \]

Equality Versions of the above Rules
The non-dependent function type \( A \rightarrow B \) is a special case of the dependent function type \((x: A) \rightarrow B\)

where \( B \) does not depend on \( x \).

The non-dependent function type

\[ A \rightarrow B \]
The Dependent Function Typein Agda

In Agda one writes (x::A) -> C for the dependent function type.

The Dependent Function Type in Agda
Alternatively, one can use the $\forall$-notation:

\[ \forall x : A \quad \Rightarrow \quad \forall x : A \quad \Rightarrow \quad C \]

The above can be rewritten as

- The above can be rewritten as
- In our slides we will use $\forall$.
- Remember that $\\%$ is used instead of $\forall$ in Agda.

\[ f :: \forall x :: A \Rightarrow C \]

\[ = \forall x :: A \Rightarrow C \]

The Dependent Function Type in Agda (Cont.)
The example is better introduced using the first notation.

\[
\text{The result would be:}
\]

\[
(\text{cons two (cons three nil))}
\]

\[
\rightarrow\begin{array}{l}
S \ x \\
\mathrm{map} \ (\lambda x : \mathbb{N} \to \mathbb{N})
\end{array}
\]

A typical example from functional programming is the \textit{map} function, which applies a function to each element of a list: functions without giving them names:

\textit{Anonymous functions}, i.e.

\[
\text{functions without giving them names:}
\]

\textit{Anonymous functions}, i.e.

\[
\text{Anonymous functions}
\]

\textit{Anonymous functions}.

However, \(\lambda\)-notation allows to introduce anonymous functions, i.e.

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

\textit{Anonymous functions}.
\[ \cdots \leftarrow (\mathbb{N} : w) \forall \leftarrow (\mathbb{N} : u) \forall \]

instead of

\[ \cdots \leftarrow (\mathbb{N} : w) \forall \]

Similarly we can write •

\[ (w, u) \forall \leftarrow (\mathbb{N} : w) \leftarrow (\mathbb{N} : u) \]

instead of

\[ (w, u) \forall \leftarrow (\mathbb{N} : w, u) \]

We can write •

Abbreviations

(C) Anton Setzer 2003 (except for pictures)
Similarly, we can write

\[ \ldots = N :: \neg (N :: w, u) \]

instead of

\[ \ldots = N :: \neg (N :: w) \]

Abbreviations (cont.)
The Dependent Function Type in Agda (Cont.)

Application has the same syntax as in the rules above:

\[ \text{Application} \]

And we have that:

\[ \exists \ a \ b \ (x :: A) \ (a \rightarrow (x :: A) \ y) \]

\[ \exists \ a \ b \ (x :: A) \ (x :: B) \]

\[ a :: A \]

\[ x :: A \]

\[ b :: B \]

\[ a \rightarrow (x :: A) \]

\[ x :: A \]

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

\[ \exists \ a \ b \ (x :: A) \]
In Agda syntax, the \( \mathcal{U} \)-rule would state that if

\[
\forall \, x : A \quad (x : A) \rightarrow B
\]

then

\[
\mathcal{F} = \forall \, x : A \quad (x : A) \rightarrow B
\]

- \( \mathcal{U} \)-rule is computationally expensive and therefore not implemented.
- The lack of the \( \mathcal{U} \)-rule causes sometimes problems.
Let $G$ be the set of genders, $G = \{\text{male}, \text{female}\}$.

For each gender $g$, $\text{Names}(g)$ is the collection of names of that gender, e.g.,

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

Example of the Use of Dependent Products

Let $\mathcal{G}$ be the set of genders, $\mathcal{G} = \{\text{male}, \text{female}\}$.

Let $\mathcal{G}$ be the set of genders.
Example of the Use of Dependent Products (Cont.)

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 \( (g : G) \times \text{Names}(g) \).
Although we haven’t introduced yet a notation for algebraic data types, the following is readable for those familiar with Haskell:
(x)B(x) = (male)B(male).

It wouldn't make sense to say select male : B.

select male will be an element of B(x) = B[male].

It wouldn't make sense to say select male : B(x).

select selects for every gender a name.

select(female) = Jill
select(male) = Tom
(Names ← (G : b)) : Select

Define Example of the Dependent Function Type
As before, here is the code for the select example, which should be readable for those familiar with Haskell:

```
{ (female) Jim ← (male) Tom; }

\begin{array}{l}
\{ (female) Jill ← (male) John; \\
\}
\end{array}
```

```
\begin{array}{l}
\{ (female) Jill ← (male) John; \\
\}
\end{array}
```

```
select \:: (g \in G) \Rightarrow Names \in G \Rightarrow Names = \begin{array}{l}
(g \in G) \Rightarrow \begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
```

The "select"-Name Example in Agda

The convention is that all rules can as well be weakened by a common context.
Example
Consider the sample derivation (assuming $\forall$ : Type):

\[
\frac{\forall \vdash \forall : \tilde{h} \vdash (\forall : \tilde{h})\forall \vdash (\forall : x)\forall}{\forall \vdash \forall : \tilde{h} \vdash (\forall : \tilde{h})\forall \vdash \forall : x}
\]

The second rule used is the rule for $\forall$-introduction without any weakening.

The first rule used is the rule for $\forall$-introduction weakened by the context $x : \forall$.

The second rule used is the rule for $\forall$-introduction weakened by the context $x : \forall$. 

\textbf{Example (Cont.)}
If we have an axiom, we need to be sure that the context, we weakened with,

Weakening of Axioms

- Will be discussed later.

This requires the context judgment $u : x_1 : A_1, \ldots, x_n : A_n \xleftarrow{u} \text{Context}$ is well-formed.
Weakening of Axioms

For the moment we mention how the formation rule for \( \mathbb{N} \):

\[
\text{Type} \quad \mathbb{N} \quad \text{Context} \quad \mu \forall x : x, \ldots, \forall x : x_1 \quad \mu \forall x : x, \ldots, \forall x_1 : x_1
\]

can be weakened:

\[
\text{Type} \quad \mathbb{N}
\]

More about this later.
Let expressions in Agda allow to introduce temporary variables, using "let-expressions". Let-expressions have the form

\[
\begin{align*}
\text{let } & a_1 :: A_1 = s_1 \\
& a_2 :: A_2 = s_2 \\
& \ldots \\
& a_n :: A_n = s_n
\end{align*}
\]
This means that we introduce new constants \( \lambda_1, \ldots, \lambda_n \) of types \( A_1, \ldots, A_n \) respectively, and can then use them. as \( s_1, \ldots, s_n \) respectively, and can then use them.

\( s_i \) can refer to \( \lambda_i \) (might be result in non-termination; termination will be discussed below).
Let expressions in Agda (Cont.)

If we are in a goal, we can use the command

```
let expression in Agda (cont.)
```

Agda will construct a template of the form:

- We have to write down the variables, separated by a blank.
- Make let expression.)

\[
\begin{align*}
\{ i \} & \text{ in } \\
\{ i \} &= \\
\{ i \} &:: \ u &\\
\ldots \ \\
\{ i \} &= \\
\{ i \} &:: \ v &\\
\{ i \} &= \\
\{ i \} &:: \ 1 &
\end{align*}
\]
Example of let expressions

Here follows a simple concrete example, which computes \((n + n) \times (n + n)\) for natural numbers \(n, m : \mathbb{N}\).
In this subsection we look at the relationship between Agda code and the corresponding derivations.

First we will go through the development of the Agda code.

Then we will look at how the corresponding derivations are developed.

We consider various examples.

Derivations and the Corresponding Agda Code

Inserted Section (c.i)
WewanttoderiveinAgda

postulateA::Type

postulate(i.e.assume)onetypeA:

–SincewewanttohavethedefinitionforanarbitrarytypeA,we

–WeneedtointroducethetypeAfirst.

Step1:

•SeeexamplefileexampleIdentity.agda

\forall (a:A)(a::A)A\rightarrow A

•WewanttoderiveinAgda

Example1
Step 2: We state our goal.

Example 1 (Cont.)
Step 3: 

The precise Agda code uses \( \text{instead of } \land \text{ and } \rightarrow \text{ instead of } \rightarrow \).

\[
\{ i \mid i \}\leftarrow \{(i \mid i) :: y\}\forall = \forall \leftarrow \forall :: f
\]

- After executing \texttt{agda-intro} we get:
  - Has to be executed while the cursor is inside one goal.
  - Agda has a command \texttt{agda-intro (Intro)} which does this step automatically.
- If introduced as a \( \forall \)-term, the term in question will be of the form
  - Elements of the function type \( \forall \) are introduced by using \( \forall \)-terms.
  - We want to derive an element of function type \( \forall \).

Example 1 (Cont.)
Step 4:

- The first goal, the type of the variable $h$ can be solved automatically.

- We obtain:
  
  $\{ i \mid i \} \leftarrow (\forall \cdot h) \forall = \forall \leftarrow \forall :: f$

  Use `agda-solve (solve)`
We obtain:

- Otherwise the changes will not be known by Agda.

```
agda-load-buffer (load Buffer)
```

and can edit everythjng.

Then one is in a mode where the goals are converted to ordinary symbols

```
agda-restart (Re)start Agda
```

If one wants to edit parts involving goals, one first has to execute:

* We can always edit the current code.

This can be done by simple editing.

- It is a good idea to rename the variable to something, for instance to `a`.

Step 4 (Cont.)
Example 1 (cont.)

Step 5:

\[ \{ i \} \leftarrow \{ i \} \leftarrow \{ a \} \]

In order for \( \forall a : A \leftarrow A \)
can use a. Since we are defining an element of type \( A \) depending on \( a :: A \), we have:

\[
A :: a \quad * 
\]

See next slide:

\[
A \leftarrow A :: f \quad * 
\]

- The context contains everything we can use when solving our goal.
- We can inspect the context.
- Step 5 (Cont.)

**Example 1 (Cont.)**
On the last slide we had \( f : \mathcal{A} \leftarrow \mathcal{A} \) in the context.

This appears, since the type checker allows to define functions recursively.

For the type checker a definition \( q :: \mathcal{A} = q \) would be legal, although evaluating \( q \) doesn't terminate (black hole recursion).

Termination Check
Termination Check (Cont.)

Agda has a command `agda-term-check-buer (Check Termination)` which checks whether recursive definitions are done properly. If the termination check succeeds, all programs checked will terminate. If the termination check fails, it might still be the case that all programs checkable will terminate.

One should use this command at the end of a session to avoid black holism.

Terminate.

One cannot write a universal termination checker, since the Turing halting problem is undecidable.
Now everything with result type $A$ (i.e. which has at the right side of the arrow $A$) can be used in order to solve the goal.

* $f$ would result in black-hole recursion.

* So we take $\alpha$.

We type in $\alpha$ into the goal and then use the command

\[
gda\text{-refine} (\text{Refine})
\]

We obtain:

\[
f :: A \rightarrow A
= \lambda (\alpha :: A) \rightarrow \alpha
\]

and are done.
Example 1: Using Rules

In Agda step 1 we postulated \( A : \text{Type} \).

In Agda step 2 we stated our goal:

\[
V \leftarrow V : 0p
\]

We write this down without any derivation.

This corresponds in the rule system that we can assume \( A : \text{Type} \), i.e., can write this down without any derivation.

In terms of rules this means that we want to derive something of type \( \forall A \).

\[
\{i \mid i\} = \forall V \leftarrow V :: f
\]

\( \forall A \leftarrow \forall : \text{Type} \)

Example I, Using Rules (Cont.)

In terms of rules, this means that we replace $p^0$ by $\forall \ v \ (\forall : v)$ which is derived by an introduction rule.

\[
\begin{align*}
V & \leftarrow V : \forall p. (V : p) \forall \\
V : \forall p & \leftarrow V : p
\end{align*}
\]

In Agda step 3 and 4 we replaced $\forall$ by $\forall$:

\[
\begin{align*}
\{i \ i\} & \leftarrow (V :: v) \forall \\
V & \leftarrow V :: f
\end{align*}
\]

\[
\begin{align*}
\{i \ i\} & \leftarrow (V :: v) \forall \\
\{i \ i\} & \leftarrow (V :: v)
\end{align*}
\]
In *Agda* step 5 were placed:

\[(a :: A) \rightarrow^1 \left( a :: A \right) \]

Interms of rules this means that we replace \( a \) by \( a :: A \).

\[ V \leftarrow V :: a \]

Example I, Using Rules (cont.)
We consider a derivation of

\[ \{ i \ i \} = \]

\[ \forall \leftrightarrow (\forall \leftrightarrow (\forall \leftrightarrow \forall)) :: f \]

Step 1:

We postulate \( A \) :: Type

We state our goal:

postulate \( A :: Type \)

See also Example 2

\[ \forall \leftrightarrow (\forall \leftrightarrow (\forall \leftrightarrow A)) :: (\forall \leftrightarrow (\forall :: A \forall :: (x :: A) \forall x \leftrightarrow (\forall :: A \leftrightarrow A)) :: x :: A \]

We consider a derivation of the example script...
Step 2:

We obtain:

\[ \{ i \cdot i \} \leftrightarrow (V \leftrightarrow (V \leftrightarrow V) :: \psi) \cdot V = V \leftrightarrow (V \leftrightarrow (V \leftrightarrow V)) :: f \]

Using \texttt{agda-solve (Solve)} we obtain:

\[ \{ i \cdot i \} \leftrightarrow (\{ i \cdot i \} :: \psi) \cdot V = V \leftrightarrow (V \leftrightarrow (V \leftrightarrow V)) :: f \]

The type of the goal is a function type, and we can use \texttt{agda-intro (Intro)}:

**Example 2 (Cont.)**
Step 2 (cont.):

- We rename the variable \( y \) to \( x \) and use \texttt{agda-load-buffer (load-buffer)}

\[
\{ i \mid i \} \leftarrow (\forall \leftarrow (\forall \leftarrow \forall) :: x) \forall = \\
\forall \leftarrow (\forall \leftarrow (\forall \leftarrow \forall)) :: f
\]

so that \texttt{Agda} realizes this change.
Example 2 (Cont.)

Step 3:

\begin{align*}
\{i \ i\} x \leftarrow (\forall \leftarrow (\forall \leftarrow \forall) :: x) \forall = \\
\forall \leftarrow (\forall \leftarrow (\forall \leftarrow \forall)) :: f
\end{align*}

We obtain *

In our case it is one (of type \(\forall \leftarrow \forall\)).

\(\forall\) itself will then apply \(x\) to as many goals as needed in order to obtain an element of the desired type.

Therefore we type \(x\) into the goal and use \texttt{agda-refine (Refine)}.

Agda will then apply \(x\) to as many goals as needed in order to obtain an element of the desired type.

We obtain

\begin{align*}
&\text{Agda will then apply } x \text{ to as many goals as needed in order to obtain an element of the desired type.} \\
&\text{Therefore we type } x \text{ into the goal and use } \texttt{agda-refine (Refine)}. \\
&\text{Agda will then apply } x \text{ to as many goals as needed in order to obtain an element of the desired type.} \\
&\text{Therefore we type } x \text{ into the goal and use } \texttt{agda-refine (Refine)}. \\
&\text{Agda will then apply } x \text{ to as many goals as needed in order to obtain an element of the desired type.} \\
&\text{Therefore we type } x \text{ into the goal and use } \texttt{agda-refine (Refine)}.
\end{align*}
Step 4:  

The type of the new goal is $\mathcal{A}$.

Example 2 (Cont.)
Using \texttt{agda-solve (solve)} we obtain:

\begin{align*}
\{i \mid i\} &\leftarrow (\forall \vdash y)(\forall x \vdash (\forall x \vdash x)(\forall x \vdash f))
\end{align*}

\textbf{Step 4 (Cont.)}
Example 2 (Cont.)

Step 4 (Cont.)

\[
\begin{align*}
\{i \ i\} & \leftarrow (\forall :: v)x \leftarrow (\forall \leftarrow (\forall \leftarrow \forall) :: x) \gamma = \\
\forall & \leftarrow (\forall \leftarrow (\forall \leftarrow \forall)) :: f
\end{align*}
\]

We rename \( h \) by \( a \), reload the buffer, and obtain:
Step 5

The new goal has type `A`.

\[(\mathbb{V} \vdash \mathbb{V} \vdash \mathbb{V}) \vdash \mathbb{V} \vdash (\mathbb{V} \vdash \mathbb{V} \vdash \mathbb{V}) : f\]

The following and are done:

- We try \(a :: A\). After inserting it and using \texttt{agda-refine} (\texttt{Refine}) we obtain
- The context contains \(A :: \text{Type}\), \(f\), \(x\) and \(a\).

This means that we sometimes have to backtrack and try a different solution. There is usually more than one solution for proceeding in Agda.

- The context contains \(A :: \text{Type}\), \(f\), \(x\) and \(a\).
- The complete expression \(\mathbb{V} :: a\) should have type \(A\).
- We can use both \(x\) and \(a\) here.

- The context contains \(A :: \text{Type}\), \(f\), \(x\) and \(a\).

We try \(a :: A\). After inserting it and using \texttt{agda-refine} (\texttt{Refine}) we obtain

\[
\begin{align*}
  (a & \vdash (\mathbb{V} :: a) \forall x \vdash (\mathbb{V} \vdash (\mathbb{V} \vdash \mathbb{V}) :: x) \forall = \\
  \mathbb{V} & \vdash (\mathbb{V} \vdash (\mathbb{V} \vdash \mathbb{V})) : f
\end{align*}
\]

Example 2 (Cont.)
Example 2, Using Rules

Postulating \( \forall \rightarrow (\forall \rightarrow (\forall \rightarrow \forall)) : \text{Type} \)

Stating the goal means that we have as last line of the derivation:

\[
\forall \rightarrow (\forall \rightarrow (\forall \rightarrow \forall)) : \text{Type}
\]

- Deriving it:

Postulating \( \forall : \text{Type} \) corresponds to assuming \( \forall : \text{Type} \) in the rules without

\[
\forall \rightarrow (\forall \rightarrow (\forall \rightarrow \forall)) : \text{Type}
\]
Example 2, Using Rules

The next step in the Agda-derivation was to replace the goal by

\[ \{ i \mid i \} \leftarrow (\forall \leftarrow (\forall \leftarrow \forall) :: x) \]
Example 2, Using Rules

Then next step in the Agda-derivation used refine.

This corresponds to replacing $\exists x$ by $\forall x$, and using one Elimination rule in order to derive it:

The left top judgement can be derived by an assumption rule (more about this later).

\[
\frac{\forall \leftarrow (\forall \leftarrow (\forall \leftarrow \forall)) : \exists \ p \ x : (\forall \leftarrow (\forall \leftarrow \forall) : x) x}{\forall : \exists \ p \ x \leftarrow \forall \leftarrow (\forall \leftarrow \forall) : x}
\]

This was replaced by $\{i, i\}$.

The next step in the Agda-derivation used refine.
\[
\begin{align*}
\text{Example 2, Using Rules} & \\
\text{We then used intro on the goal which was then replaced by } & \text{by } \exists p \cdot (A : \forall) \forall \text{ which can be introduced by } \\
\{ i, i \} & \text{ by } \forall (A :: x) \forall
\end{align*}
\]
Finally we used reification which replaced the goal by \( a \).

**Example 2. Using Rules**

\[
V \leftarrow (V \leftarrow (V \leftarrow V)) \cdot (\nu \cdot (V : V) V) \cdot \chi \cdot (V \leftarrow (V \leftarrow V) : x) V
\]

\[
V : (\nu \cdot (V : V) V) \cdot x \leftarrow V \leftarrow (V \leftarrow V) : x
\]

\[
V \leftarrow V : \nu \cdot (V : V) \chi \leftarrow V \leftarrow (V \leftarrow V) : x
\]

\[
V \leftarrow (V \leftarrow V) : x \leftarrow V \leftarrow (V \leftarrow V) : x
\]

\[
V : \nu \leftarrow V : \nu \cdot V \leftarrow (V \leftarrow V) : x
\]

This corresponds to replacing \( d \) by \( a \).
Example 3

\[ A \rightarrow B \rightarrow AB \]

We derive an element of type $AB$ which is the product of $A$ and $B$.

(See exampleProductIntro.agda)
Step 1: We postulate types $A$, $B$:

```
\{ B :: q \\
 \forall A :: \text{sig} \}
AB :: \text{Type}
```

This will be a record with element $a :: A$, $q :: B$.

- We introduce the product of $A$, $B$:

```
\text{postulate } B :: \text{Type} \\
\text{postulate } A :: \text{Type}
```

- We postulate types $A$, $B$:

Exhibit 3 (Cont.)
Step 2:

Our goal is:

\[ \{ i \ i \} = \]

\[ B \leftarrow B \leftarrow A \ V \ A B \leftarrow f \]

Example 3 (Cont.)
Step 3:

We use intro.

Example 3 (Cont.)
\{i \ i\} \leftarrow (B :: A) \chi \leftarrow (V :: \varphi) \chi =
\{i \ i\} \leftarrow (\{i \ i\} :: \psi) \chi \leftarrow (\{i \ i\} :: \psi) \chi =

\begin{align*}
AB & \leftarrow B \leftarrow V :: f
\end{align*}

After applying A\&d-solve and renaming of variables we get:

\begin{align*}
\{i \ i\} & \leftarrow (\{i \ i\} :: \psi) \chi \leftarrow (\{i \ i\} :: \psi) \chi =
AB & \leftarrow B \leftarrow V :: f
\end{align*}

After applying into we get:

Step 3 (Cont.)
Step 4:

- Thenewgoalistotype $AB$ whichisarecordtype. Anelementofitcanbeintroducedbyanintroductionrule.

- Anelementoffirstype $AB$ whichisarecordtype.

- Elements of type $AB$ introducedbytheintroductionprinciplewillhave

$$\{i, i\} = q$$

$$\{i, i\} = a \rightarrow \{i, i\} = a$$

$$AB \leftarrow B \leftarrow A : f$$

- Whenusingintrowecget

$$\{i, i\} = q$$

$$\{i, i\} = a$$

$$\{i, i\} = a$$

...
Step 5:

- The first goal has as context:

Example 3 (cont.)
We insert a, use refine and solve the first goal:

\[
\begin{align*}
\{ \{ i \quad i \} = q \\
\quad p = q \}\quad \text{struct} &\quad \left( B :: \quad \_ \right) \chi \quad \left( \forall :: \quad \_ \right) \chi = \\
\chi &\quad A B \quad \leftarrow B \quad \leftarrow A :: \quad \_ \\
\end{align*}
\]

Step 5 (cont.)
Similarly we can solve the second one:

Step 6: 

Example 3 (Cont.)
The definition of $AB$ means that $AB$ abbreviates $A \times B$.

\[ \begin{array}{c}
\text{Type} : B \\
\text{Type} : A \\
\hline
\text{Type} : A \times B \\
\end{array} \]

which can be derived as follows.

\[ \begin{array}{c}
\text{Type} : B \\
\text{Type} : A \\
\hline
\text{Type} : A \times B \\
\end{array} \]

Using assumption rules only, 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
Stating the goal corresponds to having as last line of the derivation:

\[
(\mathcal{B} \times \mathcal{A}) \rightarrow \mathcal{B} \rightarrow \mathcal{A}:^0 \mathcal{p}
\]

Using into means that we replace \(^0\mathcal{p}\) by \(\mathcal{p}\) which is introduced by two introduction rules:

\[
\frac{(\mathcal{B} \times \mathcal{A}) \rightarrow \mathcal{B} \rightarrow \mathcal{A}:^1 \mathcal{p} \cdot (\mathcal{B} : ^0 \mathcal{q}) \cdot (\mathcal{A} : ^0 \mathcal{p}) \cdot (\mathcal{B} \times \mathcal{A}) \rightarrow \mathcal{B} \rightarrow \mathcal{A}:^1 \mathcal{p}}{(\mathcal{B} \times \mathcal{A}) \rightarrow \mathcal{B} \rightarrow \mathcal{A}:^0 \mathcal{p}}
\]

Example 3, Using Rules (Cont.)
Example 3, Using Rules (Cont.)

Introduction by an introduction rule:

\[
\begin{align*}
(B \times A) & \leftarrow B \leftarrow A : \langle \exists p, \forall p \rangle (B : \alpha \forall (A : \rho) \chi) \\
(B \times A) & \leftarrow B : \langle \exists p, \forall p \rangle (B : \alpha) \chi \leftarrow A : \rho \\
B \times A & \leftarrow B : \langle \exists p, \forall p \rangle \leftarrow B : \alpha, A : \rho \\
B & \leftarrow B : \alpha, A : \rho, A : \rho \\
\end{align*}
\]

which can be

\[
\begin{align*}
B \times A & \leftarrow B \leftarrow A : \langle \exists p, \forall p \rangle (B : \alpha \forall (A : \rho) \chi) \\
B \times A & \leftarrow B : \langle \exists p, \forall p \rangle (B : \alpha) \chi \leftarrow A : \rho \\
B \times A & \leftarrow B : \langle \exists p, \forall p \rangle \leftarrow B : \alpha, A : \rho \\
B & \leftarrow B : \alpha, A : \rho, A : \rho \\
\end{align*}
\]

Using intro again means that we replace \( p \) by \( \langle p, p \rangle \), which can be...
Example 3, Using Rules (Cont.)

The premises require an assumption rule (which will use the derivation of $A \times B$), see later for details.

\[
\begin{align*}
(B \times V) & \leftarrow B \leftarrow V : \langle q, p \rangle \vdash (B : q) \land (V : p) \\
(B \times V) & \leftarrow B : \langle q, p \rangle \vdash (B : q) \land (V : p) \leftarrow V : p \\
B \times V & \vdash \langle q, p \rangle \leftarrow B : q, V : p \\
B : q & \leftarrow B : q, V : p \\
V : p & \leftarrow B : q, V : p
\end{align*}
\]

\[d_3 \text{ by } c: \]

Solving the goals by refining them with $q$, $q$ means that we replace $d_2$ by $q$. 

(C) Anton Setzer 2003 (except for pictures)
Example 4

We derive an element of type \((A \leftarrow B \leftarrow A \leftarrow BC)\)

where \(BC\) is the product of \(B\) and \(C\). (See Example Product Elim.agda.)
Step 1:

- We postulate types $A$, $B$, $C$:

\[
\begin{aligned}
A &:: \text{Type} \\
B &:: \text{Type} \\
C &:: \text{Type} \\
\end{aligned}
\]

- We introduce the product of $B$, $C$:

\[
\begin{aligned}
\{ &:: C \\
B &:: q} = \\
BC &:: \text{Type} \\
\end{aligned}
\]
Step 2:

Our goal is:

\[
\{ i \ i \} = B \leftarrow \forall \forall (BC \leftarrow \forall) :: f
\]
Step 3:

We use intro and get (after using agda-solve and renaming of variables):

\[
\begin{align*}
\{ i \ i \} \leftarrow (A \!::\! a)\!\chi \leftarrow (A \!::\! x)\!\chi = \\
B \leftarrow A \!\vdash (B \!::\! x) \leftarrow f
\end{align*}
\]
Step 4:

Example 4 (Cont.)

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:

Using agda-let (Make let expression) we obtain:

\[
\{i \ i\} \ \text{in} \\
\{i \ i\} = \\
\{i \ i\} :: \ \text{let } \left( A :: a \right) \chi \leftarrow (A :: x) \chi = \\
B \leftarrow A \leftarrow (A :: x) \chi = \\
f
\]

(c) Anton Setzer 2003 (except for pictures)
Step 5: We insert a type of variable \( bc \) of type \( BC \) (using refine) and obtain:

\[
\{ i \mid i \} \quad \text{in} \\
\{ i \mid i \} = \\
\{ i \mid i \} = \\
\begin{array}{ll}
\text{BC} : & \text{let } \chi \leftarrow \forall : a \forall : B C \forall : A \forall : x \forall : f \\
& \chi \leftarrow A : B C \leftarrow A : B C \leftarrow A : B C \leftarrow A \end{array}
\]
Step 6:

For solving the first goal (definition of $bc$) we can refine $x$, which has as result type $BC$.

Example 4 (Cont.)
Step 7: The new goal can be solved by refining it with variable $a$. 

Example 4 (cont.)
Step 8:

Example 4 (Cont.)

Currently, Agda doesn't have any direct support for refining \( q \) to an element of type \( B \). Hence, we have to do this by hand. Insert \( bc \cdot q \), choose refining and obtain:
In our rule calculus we don’t introduce a let construction (we could add this).

\[ q(x) \leftarrow (\forall :: v) \chi \leftarrow (\forall :: x) \chi = \]
\[ \beta \leftarrow \forall \leftarrow (\forall \beta \leftarrow \forall) :: f \]

We get

- In order to get close to the derivations, we omit in the Aęda derivation the let expression, and replace in the body of it with its definition \((x).\)

Example 4 (cont.)
Example 4, Using Rules

Using rules, we start with our goal

\[ B \leftarrow \forall \left( (\forall C \times B) \leftarrow \forall \right) : 0 \mu \]

Critical Systems, CS-411, Lenterm 2003, Sec. B2
Example 4, Using Rules (Cont.)

The intro step amounts to replacing $d_0$ by

\[ \Gamma \vdash \varphi (\alpha, (\mathcal{C} \times B) \leftarrow \forall : x) \\]

Introduced by two applications of an introduction rule:

\[ \Gamma \vdash \varphi (\alpha, (\mathcal{C} \times B) \leftarrow \forall : x) \\]
The two initial judgements can be introduced by assumption rules:

\[
\begin{align*}
B & \leftarrow V \leftarrow ((C \times B) \leftarrow V) : (\forall x)^0 \forall (V : a) (C \times B) \leftarrow V : x) \\
B & \leftarrow V : (\forall x)^0 \forall (V : a) (C \times B) \leftarrow V : x \\
B : (\forall x)^0 \forall & (V : a) (C \times B) \leftarrow V : x \\
C \times B & \leftarrow V : a (C \times B) \leftarrow V : x \\
V : a & \leftarrow V : a (C \times B) \leftarrow V : x \\
(C \times B) & \leftarrow V : x \\
V : x & \leftarrow V : x \\
\end{align*}
\]

This can be introduced by two applications of elimination rules:

\[
\begin{align*}
\end{align*}
\]

In our rule calculus, this reads $\forall x^0 (a) (x)$.

In Agda, we then replace the goal corresponding to $p$ by $\forall (a)(x)$.

---

Example 4, Using Rules (Cont.)
depend on $\mathcal{I}$. In fact $\mathcal{I}$ is usually not really a context, but a „context piece“ – it might denote contexts.

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]

\[ \mathcal{C} : z : h, \quad \mathcal{A} : x = \mathcal{I}, \quad \mathcal{C} : z = \mathcal{I}, \quad \mathcal{A} : x = \mathcal{I} \]
Similarly, for $\forall x : B', \forall y : B : x = y$
The expression $\forall x : B', \forall y : B : x = y$ stands for the context $\Gamma$.

Abbreviation for $\theta \iff \emptyset$ (no variables are bound in it).

A non-dependent judgement (e.g. $\forall : A : \text{Type}$) can be regarded as an empty context.

Critical Systems, CS-411, Lenterm 2003, Sec. B2

Notations for Contexts (Cont.)
Sometimes we need a new context. Expressing "I is a valid context".

\[ \text{Context} \leftarrow I \]

- We form a new judgement
  \[ \begin{align*}
  \forall : \mathcal{C} & \leftarrow \mathcal{B} : \forall \forall : x \ * \\
  \mathcal{B} & \leftarrow \forall : x \ * \\
  \forall : x & \leftarrow \forall : x \ *
  \end{align*} \]

- If I is a valid context, "I is a valid context".

Sometimes we need assumptions on axioms of an assertion.
Context

Extending a context

The empty context

Context Rules

(where in the last rule \( x \) must not occur in \( \Gamma \)).

\[
\frac{\Gamma \vdash A \leftarrow \bigcirc}{\Gamma \vdash A : T} \quad \frac{\Gamma \vdash \bigcirc \leftarrow}{\Gamma \vdash \bigcirc}
\]
We assume the following formation rule for the type of natural numbers:

\[
\frac{N \in \text{Type}}{\text{Context} \implies N \in \text{Type}}
\]

Introduced the rules

- With this rule, following the convention on slide B2-63, we have as well

\[
N \in \text{Type}
\]

Example Derivation (Context Rules)
Example Derivation (Context Rules)

\[
\text{Context} \leftarrow N : z, N : \tilde{h}, N : x \quad \text{Type} \leftarrow N \leftarrow N : \tilde{h}, N : x \\
\text{Context} \leftarrow N : \tilde{h}, N : x \\
\text{Type} \leftarrow N : \tilde{h}, N : x \\
\text{Context} \leftarrow N \leftarrow N : \tilde{h}, N : x
\]

(Note that \text{Type} \leftarrow N : \tilde{h}, N : \text{Type} is the same as \text{Type} \leftarrow \emptyset)

The following derives \( N : \tilde{h}, N : z, N : \tilde{h} \quad \text{Context} \leftarrow N \leftarrow N : \tilde{h}, N : x \).
Assumption Rule
Example Derivation (Assumption Rule)

\[
\begin{align*}
N : f_i & \leftarrow N : z', N : f_i', N : x \\
\text{Context} & \leftarrow N : z', N : f_i', N : x \\
\text{Type} : N & \leftarrow N : f_i', N : x \\
\text{Context} & \leftarrow N : f_i', N : x \\
\text{Type} : N & \leftarrow N : x \\
\text{Context} & \leftarrow N : x \\
\end{align*}
\]

The following derives $x : N$, $N : f_i$, $N : z'$.
When we define a function:

\[ \text{Assumption Rule in Agda} \]
In order to solve this goal:

\[
\begin{align*}
B & \leftarrow \forall : \alpha (\forall : \alpha) \chi \\
\overline{B : \alpha \iff \forall : \alpha}
\end{align*}
\]

If \( B \) is equal to \( \forall \) we can use the assumption rule directly.

\[
\begin{align*}
B & \leftarrow \forall : \{i \ i\} (\forall : \alpha) \chi \\
\overline{B : \{i \ i\} \iff \forall : \alpha}
\end{align*}
\]

The above corresponds to a derivation.

**Assumption Rule in Agda (Cont.):**
More generally we might in the derivation of \( a : A \) make anywhere use of \( a : A \), as long as this is in the context.

**Assumption Rule in Agda (cont.)**
Similarly, when solving the goal

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

in order to derive

\[ B :: \{i \ i\} \leftarrow V :: a \]

so we have to solve

\[ B \leftarrow V :: \{i \ i\} \leftarrow (V :: a) \]

In fact, when solving the above, we implicitly use the rule — in fact when solving the above, we implicitly use the rule

\[ \{i \ i\} \leftarrow (V :: a) \]

In order to derive

\[ f :: A \rightarrow B \]

Similarly, when solving the goal
Weakening Rule

\[ \theta \vdash_{\text{Context}} \text{is weakened by } \forall. \]

- The judgement $I, I'$ is weakened by $\forall$.

a judgement

\[ \text{This rule allows to add an additional context piece } (\forall) \text{ to the context of the judgement.} \]

\[ \theta \vdash_{\text{Context}} I, I' \]

\[ \theta \vdash_{\text{Context}} I, \forall I', I' \]

\[ \theta \vdash I, I' \]

\[ \theta \vdash I, I' \]

\[ \theta \vdash I, I' \]

Weakening Rule

\[ \theta \vdash I, I' \]
Remark: One can in fact show that the Thinning rule can be weakly derived.

An exception is when we additionally assume some judgments.

- Then $\Gamma \vdash A : \text{Type}$ doesn't follow without the weakening rule.

- $A : \text{Type}$ (corresponding to "postulate" in Agda).

- However, this can't be derived from the premise the conclusion directly.

- Whenever the assumptions of the rule can be derived in the complete set of rules we provide, then as well the conclusion.

- Weakly derived means: Whenever the assumptions of the rule can be weakly derived.
The following derives the first premise in Example 3 (slide B2-106) from assumptions $A : \text{Type}, \quad B : \text{Type}$.

---

**Example Derivation (Weakening Rule)**

1. $\forall \alpha : p \Leftarrow B : \forall \alpha'B : p$
2. $\forall \alpha' B : \forall \alpha : p$
3. $\forall \alpha : p$
4. $\forall \alpha : \text{Type}$
5. $\forall \alpha : \forall \alpha' : p$
6. $\forall \alpha : \forall \alpha' : p$
7. $\forall \alpha : \text{Type}$
The following derives the first premise in Example 4 (slide B2-119) from assumptions
$\text{Type}: A$, $\text{Type}: B$, $\text{Type}: C$.

Example Derivation 2 (Weakening Rule)
General Equality Rules

Reflexivity

\[
\begin{align*}
\forall \ : \ a = a \\
\forall \ : \ q = q
\end{align*}
\]

Symmetry

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

(Reflexivity can be weakly derived, except for additional assumptions.)

\[
\begin{align*}
\forall \ : \ a = a \\
\forall \ : \ q = q
\end{align*}
\]

Reflexivity
General Equality Rules (Cont.)

Transitivity

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

\[
\frac{\forall : c = a}{\text{Type} : \forall = c} \quad \frac{\forall : q = a}{\text{Type} : \forall = q}
\]

\[
\frac{\text{Type} : \forall = c}{\text{Type} : \forall = B} \quad \frac{\text{Type} : \forall = \forall}{\text{Type} : \forall = \forall}
\]
Example Derivation (General Equality Rules)
Substitution Rules

The following rules can be weakly derived:

\[ [v = : x]q = [v = : x]q \iff [v = : x] \psi \]
\[ \forall : [v = : x] q \iff \exists \forall : x', \psi \]

Substitution 1

\[ [v = : x] A \]
\[ 0 ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]
\[ ]

Substitution 2

\[ [v = : x] B = [v = : x] B \iff [v = : x] \psi \]
\[ \forall : [v = : x] B \iff \exists \forall : x', \psi \]

Substitution 3

\[ [v = : x] \theta \iff [v = : x] \theta \]
\[ \forall : [v = : x] \theta \iff \exists \forall : x', \psi \]
Example Derivation (Substitution)
Example Derivation (Substitution)

\[ \begin{align*}
N : (\hat{h} + (z)S) \cdot (N : \hat{h}) & \cdot (N : z)Y = \hat{h} + (0 + (z)S) \cdot (N : \hat{h}) \cdot (N : z)Y \\
N : (\hat{h} + (z)S) \cdot (N : \hat{h}) & \cdot (N : z)Y = \hat{h} + (0 + (z)S) \cdot (N : \hat{h}) \cdot (N : z)Y & \iff N : z \\
N : \hat{h} + (z)S & = \hat{h} + (0 + (z)S) & \iff N : \hat{h}, N : z \\
N : (z)S & = 0 + (z)S & \iff N : z \\
N : \hat{h} + x & \iff N : \hat{h}, N : x, N : z \\
N : (z)S & \iff N : z \\
N : z & \iff N : z \\
\text{Context} & \iff N : z \\
\text{Type} & : N
\end{align*} \]
In order to derive \( x : \text{Type} \), we need to show:

\[
\forall \ : B \iff \forall x : A
\]

So the judgement implicitly contains the judgements

\[
\forall \ : C \iff B \iff \forall x : A
\]

In order to derive \( x : \text{Type} \), we need to show:

\[
A \ :	ext{Type} \quad x : A \quad B : \text{Type}
\]
The next slide shows the presuppositions of judgments.

\[ Type \ C \leftarrow B : y, \forall x : x \]

Pre-suppositions of the judgment

Pre-suppositions (Cont.)
<table>
<thead>
<tr>
<th>Presuppositions</th>
<th>Presuppositions</th>
<th>Presuppositions</th>
<th>Presuppositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma, x : A \vdash \text{Context} )</td>
<td>( \Gamma \vdash A : \text{Type} )</td>
<td>( \Gamma \vdash \text{Context} )</td>
<td>( \Gamma \vdash A : \text{Type} )</td>
</tr>
<tr>
<td>( \Gamma \vdash A = B : \text{Type} )</td>
<td>( \Gamma \vdash A = B : \text{Type} )</td>
<td>( \Gamma \vdash a : A )</td>
<td>( \Gamma \vdash a : A )</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
\text{Judgement} & \quad \text{Presuppositions} \\
\text{Judgement} \quad \text{Presuppositions} & \\
\end{align*}
\]

\[
\begin{align*}
\mathsf{J} : \mathcal{B} & \iff \forall x : \mathbf{A} \rightarrow \mathbf{B} \\
\mathsf{J} : \mathsf{J} & \iff \forall (x) : \mathbf{A} \rightarrow \mathbf{B} \\
\mathsf{J} : \mathsf{J} & \iff \forall (x) : \mathbf{A} \rightarrow \mathbf{B} \\
\forall q : \mathbf{A} & \iff \mathsf{J} \\
\forall q : \mathbf{A} & \iff \mathsf{J} \\
\forall q : \mathbf{A} & \iff \mathsf{J} \\
\forall q = a & \iff \mathsf{J} \\
\end{align*}
\]
Furthermore, presuppositions of presuppositions of presuppositions are as well presuppositions of presuppositions of presuppositions.
Example of Presuppositions:

\[ D \times (c : z) : q \equiv B : h', \forall : x \]
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

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

and our rules allow this only if we had

\[
\text{Type} : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}
\]
The corresponding paradox is called Girard's paradox.

- Using this rule we can prove everything, especially false formulas.

As a rule results however in an inconsistent theory:

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

\[ \text{Set} \]
Per Martin-Löf

The Founder of Martin-Löf Type Theory.
The main theoretician behind Agda (which was implemented by his wife, of whom I have no picture).
Instead we introduce a new type:

- A set is a small type.
- \( \text{Set} \) is the type of sets.

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

$\text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$

and we can assign to $\text{prod}$ above the type $\text{prod : Set : Set}$.

$\text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$

Since $\text{Set : Type}$ we get

$\exists \text{Set : Type}$ we get

We add rules asserting that if $A : \text{Set}$ then $A : \text{Type}$.

$\exists \text{Set : Type}$
Set $\rightarrow$ Set

$\mathbb{N} \rightarrow \mathbb{N}$

Type

$\mathbb{N} \rightarrow \times \rightarrow \mathbb{N}$

$\times \rightarrow \times \rightarrow \mathbb{N}$

Set

Type
However, we cannot use prod in order to form the product of two sets, i.e., we cannot introduce sets. Since Set : Set does not hold, that would result in the same inconsistency as Type : Type.

* That would result in the same inconsistency as Type : Type.

However, we cannot use prod in order to form the product of two sets (cont.).
Formulation Rule for Set

\[
\begin{align*}
\text{Set} & : \text{B} \leftarrow (\forall : x) \\
\text{Set} & \leftarrow \forall : x \\
\text{Set} & : \text{B} \times (\forall : x) \\
\text{Set} & \leftarrow \forall : x \\
\text{Set} & : \text{A} \\
\end{align*}
\]

Every Set is a Type

Formation Rule for Set

Set : Type

Rules for Set
Equality Versions of the Above Rules

Formation Rule for Set

Set : Type

Every Set is a Type

Set = Set : Type


Closure of Set under the dependent function type

\[
\frac{\text{Set} : \mathcal{B} \leftarrow (\forall x : \mathcal{A}) = \mathcal{B} \leftarrow (\forall x : \mathcal{A})}{\text{Set} : \mathcal{A} \times (\forall x : \mathcal{A} = \mathcal{A} \times (\forall x : \mathcal{A})}
\]

Closure of Set under the dependent product

\[
\frac{\text{Set} : \mathcal{B} = \mathcal{B} \leftarrow (\forall x : \mathcal{A}) = \mathcal{B} \leftarrow (\forall x : \mathcal{A})}{\text{Set} : \mathcal{A} \times (\forall x : \mathcal{A} = \mathcal{A} \times (\forall x : \mathcal{A})}
\]

(© Anton Setzer 2003 (except for pictures)
We can now introduce \texttt{prod}:

\[
\text{prod} : \text{Set} \times \text{Set} \rightarrow \text{Set}
\]
\[ \lambda \times (X : x) \lambda X' = \text{prod} \]

So define

\[
\begin{align*}
\text{Set} & \leftarrow \text{Set} \\
\text{Set} & \leftarrow \lambda \times (X : x) \lambda X' \\
\text{Set} & \leftarrow \lambda \times (X : x) \lambda X' \\
\text{Set} & \leftarrow \lambda \times (X : x)
\end{align*}
\]

Now we can derive our desired judgment:

Example: prod (Cont.)
Hierarchies of Types

If one wants to form prod:Type

Then Type ← Type ← Type : Kind.

If one wants to form prod:Type

Then Type ← Type ← Type : Kind.

Further level Kind, s.t. Type : Kind.

...
Plus equality versions of the above rules.

\[
\begin{align*}
\text{Kind : } B & \iff (\forall : x) \\
\text{Type : } A & \iff \forall : x \\
\text{Kind : } A & \iff (\forall : x) \\
\text{Type : } B & \iff (\forall : x) \\
\end{align*}
\]

Closure of Kind under the dependent function type

\[
\begin{align*}
\text{Kind : } B \times (\forall : x) \Rightarrow \forall : A \\
\text{Kind : } A \Rightarrow (\forall : x) \\
\end{align*}
\]

Closure of Kind under the dependent product

\[
\begin{align*}
\text{Type is a Kind} \\
\text{Type is a Kind} \\
\end{align*}
\]
Hierarchies of Types (Cont.)

This can be iterated further, forming

\[ \text{Type} = \text{Type}_1 \text{Kind} = \text{Type}_2, \text{Type}_3, \text{Type}_4, \ldots \]

So we have:

- \#1
- \#2
- \#3
- \#4

(in the rule calculus called Kind) \#3 etc.

\#2 says has a hierarchy of types built in, written as \#1 (which is \#ype), \#2

- etc.


B2-160