### Basic Form of Rules

There are four kinds of rules:

1. **Formation Rules**
   - Formation rules introduce new types.
   - The conclusion of such a rule will have the form: $C(a_1; \ldots; a_n) : \text{Type}$.

2. **Introduction Rules**
   - Introduction rules have one such rule.
   - Each type constructor has one such rule.

3. **Elimination Rules**
   - Elimination rules have multiple rules.
   - Typically, each type constructor has multiple elimination rules.
   - For each type constructor, we have usually 4 kinds of rules:
     - Equality rules.
     - Equality version of the formation, introduction and elimination rules.

4. **Equality Rules**
   - Equality rules have a single rule.
   - The conclusion of such a rule will have the form: $a_1 = a_2$.

### Example 1: The Type of Lists

A type constructor is List.

- List($A$) is the type of lists of type $A$.

The type constructor is List.

- List($A$) : Type
  
- $A : \text{Type}$
Example 1: The Type of Natural Numbers

Formation rule for the type of natural numbers:

\[ N : \text{Type} \]

The type constructor is \( N \).

- \( N \) has 0 arguments, and we write \( N \) instead of \( N() \).

Later we will see that we can replace in this example \( \text{Type} \) by \( \text{Set} \).

Critical Systems, CS 411, Lent Term 2003, Sec. B2

Example 3: The Non-Dependent Product

Formation rule for the non-dependent product:

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

The type constructor is \( \times \).

- \( \times \) is asci symbol 215 (not the letter \( x \)).

Example 2: The Non-Dependent Product

Agda syntax for introducing the non-dependent product:

\[ (\times)(A : \text{Type})(B : \text{Type}) : \text{Type} = \]

\[ \text{isanAgdade}\text{fnitionofthistype(moreaboutthislater).} \]

Formulation for the type of natural numbers:

\[ N : \text{Type} \]

The formation of a type is usually done by introducing a constant of a certain

Critical Systems, CS 411, Lent Term 2003, Sec. B2
Currying in Agda

Traditionally, one writes in

\[ \text{type-constructors in } \text{uncurried form} \].

\[ \text{List alone does not make sense as a term. We have to write } \text{List}(A) \].

In Agda, type constructors (except those predefined for dep. product and

\[ \text{function type}s \text{ always come } \text{curried} \].

\[ \text{We have to write } \text{term} \text{ and not } \text{term} \text{.} \]

\[ \text{Traditional, one writes in } \text{type theory } \text{type-constructors in uncurried form.} \]


(2) Introduction Rules

\[ \text{C might have zero arguments, then we write } \text{C instead of } \text{C()} \].

\[ \text{C is a constructor of term-constructors.} \]

\[ \text{A is a type} \text{ introduced by the corresponding formation rule,} \]

where

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

\[ \text{The conclusion of such a rule will have the form} \]

\[ \text{The introduction rule introduces elements of a type.} \]

Introduction Rule, Example 1a

\[ \text{The type } \text{NatList of type N has two introduction rules:} \]

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

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

\[ \text{List} \rightarrow \text{Type} : \ N(\times) \]

\[ \text{So } \text{List alone is a term and has type} \text{ is kind and not a type} \].

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

In Agda, type constructors (except those predefined for dep. product and

\[ \text{function type}s \text{ are always curried.} \]

\[ \text{We have to write } \text{term} \text{ and not } \text{term} \text{.} \]

\[ \text{Thus alone does not make sense as a term.} \]

\[ \text{Traditionally one writes in } \text{type theory } \text{type-constructors in uncurried form.} \]

Currying in Agda (Cont.)
Introduction

Rule, Example 1b

Wegeneralizethepreviousexampletolistsofarbitrarytype.

Lists

of type $\mathrm{A}$ havetwointroductionrules:

- $\mathrm{A} \triangleright \mathrm{nil} : \mathrm{List}(\mathrm{A})$
- $\mathrm{A} \triangleright \mathrm{a}, \mathrm{A} \triangleright \mathrm{List}(\mathrm{A})$

Incaseoftherulefor $\mathrm{nil}$, weneededthe premise $\mathrm{A} \triangleright \mathrm{Type}$ inorder to guaranteethatwecanformthe type $\mathrm{List}(\mathrm{A})$.

Incaseoftherulefor $\mathrm{cons}$, thispremiseis implicit in the premise $\forall \pi. \mathrm{List}(\pi)$.

Example 2: NaturalNumbers.

The naturalnumbers $\mathbb{N}$ canbeconsideredasbeingformedfromtwo operations:

- $0$, $\mathrm{S}(\mathrm{n})$ where $\mathrm{S}(\mathrm{n}) = \mathrm{n} + 1$

Usingtheseoperationsweform $0, \mathrm{S}(0), \mathrm{S}(1)$, ... and thereforeallnaturalnumbers.

The constructors of $\mathbb{N}$ are $0$ and $\mathrm{S}$.

Examples: $\mathrm{S}(\mathrm{S}(\mathrm{S}(0)))$ standsfor $\mathrm{S} (\mathrm{S}(\mathrm{S}(0)))$.

Constructors and Canonical Elements

Canonical elements therefore always start with a constructor.

Canonical elements of a type are those introduced by an introduction rule.
Any element of a type has to reduce to a canonical element of it.

Reduction Rules for \( \mathbb{N} \)

- \( n + 0 \mapsto n \)
- \( n + S(m) \mapsto S(n + m) \)

These (and reductions using them in subterms) are the one-step reductions. Reductions can be formed by a sequence of one-step reductions.

Example:

\[
((((0)s)s)s)s \quad \mapsto \quad ((0 + ((0)s)s)s)s \quad \mapsto \quad (0)S + ((0)s)s)s \quad \mapsto \quad (0)S + S + u = S + 2 \quad \mapsto \quad (S + 2)S = S(S + 2)
\]

Critical Systems, CS 411, Lecture 2003, Section B2

Constructors and Canonical Elements (cont.)

Constructors are functions that take arguments and produce results.

- The outermost operator changes to \( S \).
- The operands reduce initially.
- The outermost operator changes to \( + \).
- Further, the arguments of the constructor reduce independently.

An element starting with \( 2 + 3 \) is not a constructor.

\[
\text{concat} (\text{cons}(2, \text{nil}), \text{nil}) \mapsto \text{cons}(2, \text{nil}) \quad \text{cons}(2, \text{nil})
\]

Again the outermost operator changes to \( \text{cons} \).

Similarly:

- If \( S \) cannot change later to \( S \), this information will remain as \( S \).
- When \( S \) changes, we have the information of the successor of something.

The outermost \( S \) always remains in place.

Constructors and Canonical Elements (cont.)

Constructors and Canoncial Elements (cont.)

The outermost operator \( (+) \) changes to \( S \).

\[
\text{concat} (\text{cons}(2, \text{nil}), \text{nil}) \mapsto \text{cons}(2, \text{nil}) \quad \text{cons}(2, \text{nil})
\]

Again the outermost operator changes to \( \text{cons} \).

Any element of a type has to reduce to a canonical element of it.

Reduction Rules for \( \mathbb{N} \)

- \( n + 0 \mapsto n \)
- \( n + S(m) \mapsto S(n + m) \)

These (and reductions using them in subterms) are the one-step reductions. Reductions can be formed by a sequence of one-step reductions.

Example:

\[
((((0)s)s)s)s \quad \mapsto \quad ((0 + ((0)s)s)s)s \quad \mapsto \quad (0)S + ((0)s)s)s \quad \mapsto \quad (0)S + S + u = S + 2 \quad \mapsto \quad (S + 2)S = S(S + 2)
\]

Critical Systems, CS 411, Lecture 2003, Section B2

Constructors and Canonical Elements (cont.)

Constructors are functions that take arguments and produce results.

- The outermost operator changes to \( S \).
- The operands reduce initially.
- The outermost operator changes to \( + \).
- Further, the arguments of the constructor reduce independently.

An element starting with \( 2 + 3 \) is not a constructor.

\[
\text{concat} (\text{cons}(2, \text{nil}), \text{nil}) \mapsto \text{cons}(2, \text{nil}) \quad \text{cons}(2, \text{nil})
\]

Again the outermost operator changes to \( \text{cons} \).

Similarly:

- If \( S \) cannot change later to \( S \), this information will remain as \( S \).
- When \( S \) changes, we have the information of the successor of something.

The outermost \( S \) always remains in place.

Constructors and Canonical Elements (cont.)

Constructors and Canoncial Elements (cont.)

The outermost operator \( (+) \) changes to \( S \).

\[
\text{concat} (\text{cons}(2, \text{nil}), \text{nil}) \mapsto \text{cons}(2, \text{nil}) \quad \text{cons}(2, \text{nil})
\]

Again the outermost operator changes to \( \text{cons} \).
Constructors in Agda

In Agda, the constructor `C` of type `A` is written as `C@(_ : A)`.

If `A` can be inferred automatically, we can replace the above by `C`.

In Agda, constructors are curried:
- If `A` can be inferred automatically, we can replace the above by `C@(_ : A)`.
- The constructors of type `A` are written as `C(@A)`.

Equality rules will express:

- `nil` for `List N` is usually cumbersome, it is better to introduce abbreviations:

```
nil :: List N = nil@
cons (n :: N) (l :: List N) :: List N = cons@
```

Example 2: Addition in `N`

Equality rules will express:

- `n + 0 = n`
- `n + S(m) = S(n + m)`

Instead we will introduce one general elimination rule which allows to
- introduce `nil` and `cons` for `List N`, and not for the
- general case. List `v` for any type `v`. That would require an extra argument
- not for the

Note that the above introduces `nil` and `cons` for `List N`, and not for the
- general case. List `v` for any type `v`. That would require an extra argument
- not for the

Example 1: First and second projection of a product:

Equality rules will express:

```
G : (v : V) e
G \times V : e
```

We have:

As type constructors, in Agda, constructors are curried:
- \( C@(_ : A) \) is written as `C(@A)`.
Elimination Rules

Inverting the Introduction Rules

Elimination rules invert the introduction rules.
The equality rule explain how to reduce $u + 0$:

\[
\frac{N : 0 + u}{N : u}
\]

The first equality rule for is as follows:

Example 2 (Equality Rule)

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

Example (Equality Rule, Cont.)

The equality rules explain how to reduce that element (namely to $u$).

\[
\frac{v : ((q \cdot v)) \circ \omega}{\frac{B \times Y : q}{B : v} \frac{Y : v}{B : u}}
\]

In the first judgment we can derive $v$ as follows:

\[
\frac{v : v = ((q \cdot v)) \circ \omega}{\frac{B : q}{B : v}}
\]

$\forall y R \times B$.

Equality rules for $\forall y R \times B$.

Example (Equality Rule, Cont.)

Immediately eliminates it.

They describe what happens, if one first introduces an element and then

Equalities rules express (Red) type theoretically.
Equality Rules

**Example 1 (Equality Rule)**

These equality rules for + are as follows:

\[
\begin{align*}
N : \mu + \mu & = \mu + u \\
N : \mu & = \mu
\end{align*}
\]

**Example 2 (Equality Rule)**

\[
\begin{align*}
\forall \lambda \exists \mu. \text{List} &= \mu \\
\exists \lambda \forall \mu. \text{List} &= \mu
\end{align*}
\]

**Example 3 (Equality Rule)**

The equality rules for + are as follows:

\[
\begin{align*}
N : (w + u)S & = (wS + u) \\
N : w & = w
\end{align*}
\]

**Equality Rules in Agda**

Equality versions of formation-, introduction- and elimination rules are defined by telling how elimination takes place. The notation for elimination however indicates already how the reductions take place. Equality rules in Agda are implicit.

Equality Versions of Rules (Cont.)

**Example:** Equality version of the introduction rule for List (rule for nil is degenerated):  
\[
\begin{align*}
\text{nil} & = \text{nil} \\
\text{a} & = \text{a} \\
\text{cons} & = \text{cons}
\end{align*}
\]

Equality Versions of Formation, Introduction- and Elimination Rules
The equality version 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, once one knows the non-equality version.

The $\eta$-rule expresses that any element of $A \times B$ is of the form $h_{something 0};something 1 i$:

- If $a : A$ and we have $A \times B$, then we get $something 0 = 0 (c)$,
- $something 1 = 1 (c)$.

On the other hand, if we have $c$ of this form, $c : A \times B$ and $A \times B$, then this follows with

If $b : A \times B$, then we get $h_{0} (c)$, $h_{1} (c) = c$ so the conclusion of the $\eta$-rule can be derived without using the $\eta$-rule.

\[
\begin{align*}
G : \rho & \vdash (\langle \rho, \eta \rangle)^0 \theta \\
V : \eta & \vdash (\langle \rho, \eta \rangle)^0 \theta
\end{align*}
\]

Equality Rules

Elimination Rules

Introduction Rule

Formulation Rule

(b) The Non-Dependent Function Type and Product

Rules of the Non-Dependent Product

The $\mu$-Rule (Cont.)

The $\eta$-Rule

This rule does not fit into the above schema:
The Rule (Cont.)

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

However, if we assume an element of type \( A \times B \), e.g., state \( x : A \times B \), then we cannot derive that \( x = h_0(x) \) without making use of the equality rule.

Equality Rules

Equality Version of the Formation Rule

\[ A = A_0 : \text{type} \]

Equality Version of the Introduction Rule

\[ a = a_0 : A \]

Equality Version of the Elimination Rule

\[ c = c_0 : A \times B \]

The \( \eta \)-Rule (Cont.)

Rules of the Non-Dependent Function Type

The reduction corresponding to the equality rule is often called \( \beta\)-reduction.
Again this rule does not fit into the above schema.

\[ f = f \]

The \( \lambda \)-Rule (Cont.)

Equality Version of the Introduction Rule

\[ \frac{A \vdash \lambda x : A \quad f \vdash x \vdash B}{\lambda x : A \vdash f \vdash B} \]

Equality Version of the Formation Rule

\[ \frac{\lambda \vdash A \quad \lambda \vdash B}{\lambda \vdash A \to B} \]

Equality Version of the Elimination Rule

\[ \frac{\lambda \vdash A \quad \lambda \vdash B}{\lambda \vdash A \equiv B \to \lambda \vdash A \equiv B} \]

Equality Versions of the Rules

\[ \begin{align*}
A & = A_0 : \text{Type} \\
B & = B_0 : \text{Type} \\
A \to B & = A_0 \to B_0 : \text{Type}
\end{align*} \]

\[ \begin{align*}
x : A & = b : B \\
x : A & = b_0 : B \\
x : A & = f : B \\
f & = f_0 : A \to B
\end{align*} \]

Equality Version of the Formation Rule

\[ \frac{A \vdash \lambda x : A \quad f \vdash x \vdash B}{\lambda x : A \vdash f \vdash B} \]

Equality Version of the Elimination Rule

\[ \frac{\lambda \vdash A \quad \lambda \vdash B}{\lambda \vdash A \equiv B \to \lambda \vdash A \equiv B} \]

Equality Versions of the Rules

\[ \begin{align*}
A & = A_0 : \text{Type} \\
B & = B_0 : \text{Type} \\
A \to B & = A_0 \to B_0 : \text{Type}
\end{align*} \]

\[ \begin{align*}
x : A & = b : B \\
x : A & = b_0 : B \\
x : A & = f : B \\
f & = f_0 : A \to B
\end{align*} \]
Rules of the Dependent Product (Cont.)

**Introduction Rule**

\[ x : A \rightarrow B \rightarrow \mathcal{P} \]

\[ a : A \rightarrow B \rightarrow \mathcal{P} \]

\[ b : B \rightarrow \mathcal{P} \left[ x := a \right] \]

\[ h_{a;b} : (x : A \rightarrow B \rightarrow \mathcal{P}) \]

The introduction rule requires an extra premise \( x : A \rightarrow B \rightarrow \mathcal{P} \), which is not implied by the other premises.

Critical Systems, CS 411, Lent Term 2003, Sec. B2

---

**Elimination Rules**

\[ x : A \rightarrow B \rightarrow \mathcal{P} \]

\[ a : A \rightarrow B \rightarrow \mathcal{P} \]

\[ b : B \rightarrow \mathcal{P} \left[ x := a \right] \]

\[ h_{a;b} : (x : A \rightarrow B \rightarrow \mathcal{P}) \]

Again, the \( \lambda \)-rule cannot be derived if the element in question is a variable.

Critical Systems, CS 411, Lent Term 2003, Sec. B2

---

**Equalities Rules**

\[ x : A \rightarrow B \rightarrow \mathcal{P} \]

\[ a : A \rightarrow B \rightarrow \mathcal{P} \]

\[ b : B \rightarrow \mathcal{P} \left[ x := a \right] \]

\[ h_{a;b} : (x : A \rightarrow B \rightarrow \mathcal{P}) \]

Again, the \( \lambda \)-rule expresses that every element of \( x : A \rightarrow B \rightarrow \mathcal{P} \) is of the form:

\[ C \times (y : x) : ((y)\{x \mapsto y\}) = y \]

\[ B \times (y : x) : y \]

We have the following \( \eta \)-rule:

---

**Equality Rules**

\[ x : A \rightarrow B \rightarrow \mathcal{P} \]

\[ a : A \rightarrow B \rightarrow \mathcal{P} \]

\[ b : B \rightarrow \mathcal{P} \left[ x := a \right] \]

\[ h_{a;b} : (x : A \rightarrow B \rightarrow \mathcal{P}) \]

Again, the \( \lambda \)-rule expresses that every element of \( x : A \rightarrow B \rightarrow \mathcal{P} \) is of the form:

\[ C \times (y : x) : ((y)\{x \mapsto y\}) = y \]

\[ B \times (y : x) : y \]

We have the following \( \eta \)-rule:

---

**Formulation Rule**

\[ x : A \rightarrow B \rightarrow \mathcal{P} \]

\[ a : A \rightarrow B \rightarrow \mathcal{P} \]

\[ b : B \rightarrow \mathcal{P} \left[ x := a \right] \]

\[ h_{a;b} : (x : A \rightarrow B \rightarrow \mathcal{P}) \]

Again, the \( \lambda \)-rule expresses that every element of \( x : A \rightarrow B \rightarrow \mathcal{P} \) is of the form:

\[ C \times (y : x) : ((y)\{x \mapsto y\}) = y \]

\[ B \times (y : x) : y \]

We have the following \( \eta \)-rule:
Equality versions of the above rules:

**Equality version of the formation rule**

\[ A = A_0 : \text{Type} \]
\[ B = B_0 : \text{Type} \]

**Equality version of the introduction rule**

\[ x : A \]
\[ B \]
\[ a = a_0 : A \]
\[ b = b_0 : B \]
\[ h_a ; b_i = h_{a_0} ; b_0_i : (x : A) B_0 \]

**Equality versions of the elimination rules**

\[ c = c_0 : (x : A) \]
\[ B_0 (c) = 0 (c_0) : A \]
\[ c = c_0 : (x : A) \]
\[ B_1 (c) = 1 (c_0) : B \]

The non-dependent product as an abbreviation:

Then the non-dependent product can now be seen as an abbreviation for:

\[ (x : A) B_0 \]

Taking \( B \) as an abbreviation, we can see later that the rule for the non-dependent product for some fresh variable \( x \):

\[ V : \text{Type} \]
\[ (V : x) \]

The non-dependent product is essentially a "labelled product".

More precisely we will see:

The non-dependent product as an abbreviation:

\[ \{(q) \} \]
\[ \{ \} \]
\[ \{ \} \]
If we have 
\[ a :: A, b :: B \] 
then we can introduce 
\[ c :: D = \text{struct } f a = a_0; b = b_0 \] 

One can introduce longer records as well, e.g.
\[ \text{sig } f a :: A; b :: B; c :: C; e :: E \] 

Unfortunately, the dependent product does not behave very well.
The rule has a special status:

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

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

Again the \( f \)-rule cannot be derived if the element in question is a variable.

Equality Version of the Elimination Rule

\[ G \leftarrow (V : x) : G \quad \{ \text{some term} \} \]

\[ G : \lambda x. G \leftarrow V : x \]

Equality Version of the Introduction Rule

\[ \text{free}(G) \land \forall x. G \leftarrow (V : x) \]

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

Equality Version of the Formation Rule

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

\[ G \leftarrow (V : x) : v = v \]

\[ G \leftarrow (V : x) : f = f \]

Relationship between Non-Dependent and Dependent Function Type

The non-dependent function type \( A \rightarrow B \) is a special case of the dependent function type

\[ \forall \alpha \in \text{elements} \quad \alpha \text{ is an element of } \alpha \]

Further terms which differ in the choice of bounded variables are identical:

\[ \forall \alpha \in \text{elements} \quad \alpha \text{ is an element of } \alpha \]

Equality

Equality Version of the Formation Rule

\[ C \leftarrow \lambda V. \alpha \]

\[ \text{free}(C) \land \forall x. C \leftarrow (V : x) \]

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

\[ C \leftarrow (V : x) : v = v \]

\[ C \leftarrow (V : x) : f = f \]

Equality Version of the Introduction Rule

\[ \text{free}(C) \land \forall x. C \leftarrow (V : x) \]

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

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

Equality Version of the Elimination Rule

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

\[ C \leftarrow (V : x) : v = v \]

\[ C \leftarrow (V : x) : f = f \]
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 on our slides

\( ! \)

instead of

\( \rightarrow \).

There are two ways of introducing an element of \((x::A)\rightarrow C\):

- \( x \) \( :: \) \( A \) \( \rightarrow \) \( C \)

  The above can be rewritten as

  - In our slides we will use \( \_ \).
  - Remember that \( \_ \) is used instead of \( x \) in Agda.

  Alternatively, one can use the \( \_ \)-notation:

  \( \_ \)-notation in Agda

  The example is better introduced using the first notation.

Abbreviations

We can write

\( (n,m::N) \rightarrow A(n,m) \)

instead of

\( (n::N) \rightarrow (m::N) \rightarrow A(n,m) \).

Similarly we can write

\( (n::N) \rightarrow (m::N) \rightarrow (n::N) \rightarrow (m::N) \)

instead of

\( (n::N) \rightarrow (m::N) \rightarrow (n::N) \rightarrow (m::N) \).

We can write

\( f :: (x::A) \rightarrow C \)

\( \equiv \)

\( \{ \)

\( x :: A \}

\( A \rightarrow C \)

\( \} \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)

\( \_ \)
Similarly we can write

\[ f(n, m :: N) :: N = \]

Instead of

\[ N :: \]

Similiarly we can write

Abbreviations (cont.)
Example of the Use of Dependent Products (Cont.)

Now the set of names is the set of pairs \((g, n)\) for a gender and a name.

Example of the Dependent Function Type

This type is written as \(\forall n : \text{Name} \times \text{Gender}(\text{Names}(g)) \to \text{Names}(g)\). The set of names is the set of pairs \((g, n)\) s.t. \(g\) is a gender and \(n\) is a name.

The "Names"-Example in Agda

Although we haven't introduced yet a notation for algebraic data types, the following is readable for those familiar with Haskell.

As before, here is the code for the select example, which should be readable for those familiar with Haskell.

```agda
G :: Type = data male | female
Names g :: Type = case g of
  male -> data Tom | Jim
  female -> data Jill | Sara;
g
AllNames :: Type = sig
  f g :: G;
  n :: Names g.
```

The "select"-Name Example in Agda

Although we haven't introduced yet a notation for algebraic data types, the following is readable for those familiar with Haskell.

As before, here is the code for the select example, which should be readable for those familiar with Haskell.

```
G :: Type = data male | female
Names g :: Type = case g of
  male -> data Tom | Jim
  female -> data Jill | Sara;
g
select :: (g :: G) -> Names g
select = (g :: G)
  case g of
    male -> Tom @
    female -> Jill @
      g
```
The convention is that all rules can as well be weakened by a common context. This means that when introducing a rule \( \Gamma \vdash A : Type \) we implicitly introduce the following rules:

\[
\theta \vdash \Gamma : h \vdash (\forall x : \alpha) \forall x : \alpha \vdash \Gamma : h, x : \alpha \\
\theta \vdash \Gamma : h \vdash (\forall x : \alpha) \forall x : \alpha \vdash \Gamma : h, x : \alpha \\
\theta \vdash \Gamma : h \vdash x : \alpha \\
\theta \vdash \Gamma : h \vdash x : \alpha \\
\theta \vdash \Gamma : h \vdash x : \alpha \\
\theta \vdash \Gamma : h \vdash x : \alpha
\]

For any choice of \( \alpha \), we implicitly introduce as well the following rules:

\[
\theta \vdash \Gamma : h \vdash (\forall x : \alpha) \forall x : \alpha \\
\theta \vdash \Gamma : h \vdash (\forall x : \alpha) \forall x : \alpha \\
\theta \vdash \Gamma : h \vdash (\forall x : \alpha) \forall x : \alpha \\
\theta \vdash \Gamma : h \vdash (\forall x : \alpha) \forall x : \alpha \\
\theta \vdash \Gamma : h \vdash (\forall x : \alpha) \forall x : \alpha \\
\theta \vdash \Gamma : h \vdash x : \alpha
\]

This means that when introducing a rule we implicitly introduce a common context.
Weakening of Axioms

If we have an axiom, we need to be sure that the context, we weakened with, is well-formed. This requires the context judgment\[ x_1 : A_1, \ldots, x_n : A_n \vdash \text{Context}. \]
is well-formed.

Let expressions in Agda (cont.)

Let expressions in Agda

Let expressions have the form

\[ \text{let}\ a_1 :: A_1 = s_1 \ldots \text{let}\ a_n :: A_n = s_n \]

Agda allows to introduce temporary variables. Using „let-expressions”.

Weakening of Axioms
Let expressions in Agda (Cont.)

If we are in a goal, we can use the command `agda-let`. Make `let expression`

determine a template of the form:

```
let a :: f!!g = f!!g a
```

We have to write down the variables, separated by a blank.

Example of Let expressions

Here follows a simple concrete example, which computes

```
(n+n) \cdot (n+n)
```

for natural numbers \(n\) and \(m\).

```
\begin{align*}
f(\mathsf{n}::\mathsf{N}) :: \mathsf{N} &= \text{let} \\
m :: \mathsf{N} &= \mathsf{n} + \mathsf{n} \\
in f \mathsf{!} m
\end{align*}
```

Step 1

We want to derive in Agda

(\text{Example I})

Derivations and the Corresponding Agda Code

Following each step in the development of the Agda code,

\textbf{Then} we will look at how the corresponding derivations are developed.

\textbf{First} we will go through the development of the Agda code.

We consider various examples.

In this subsection we look at the relationship between Agda code and the
corresponding derivations.

Let \(a::A\) 

\textbf{Postulate \(a::A\) type} 

\textbf{Postulate (i.e. assume) one type \(a\):} 

\begin{align*}
&\text{Since we want to have the definition for an arbitrary type} \ a, \\
&\text{we need to introduce the type} \ a \ \text{first.}
\end{align*}

\textbf{Step I} 

\textbf{Example I}

\textbf{(Inserted Section (c.i))}

Derivations and the Corresponding Agda Code

(\text{Example I})
Step 2:

We state our goal:

\[ \forall x : A \to A = \forall x : A \to \forall y : f \]

Agda-lead-lag (lead buffer)

and call edit everything.

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

agda-lead-lag (lead buffer)

If there is a mode where the goals are converted to ordinary symbols

Otherwise the changes will not be known by Agda.

Agda lead-lag (lead buffer)

* In the end of any editing one should execute:

Agda-lead-lag (lead buffer)

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

Agda-lead-lag (lead buffer)

* We can always edit the current code.

Agda-lead-lag (lead buffer)

* This can be done by simple editing.

Agda-lead-lag (lead buffer)

— It is a good idea to rename the variable to something. For instance to \( a \):

Step 4 (cont.)

Example 1 (cont.)

Step 3:

{\{ i \} \leftarrow (\forall \ \psi : \forall \ \forall y : f \}

\[ \forall \ \orall y : f \]

Agda-lead-lag (lead buffer)

* Has to be executed while the cursor is inside one goal.

Agda-lead-lag (lead buffer)

* This has a command agda-intro (intro) which does this step.

Agda-lead-lag (lead buffer)

— Agda has a command agda-solve (solve) which does this step.

Agda-lead-lag (lead buffer)

— It introduces a \( \exists \)-term. The term in question will be of the form.

Agda-lead-lag (lead buffer)

— Elements of the function type \( \forall x : \forall y \) are introduced by using \( \exists \)-terms.

Agda-lead-lag (lead buffer)

— We want to derive an element of function type \( \forall x : \forall y \) —

Step 4 (cont.)

Example 1 (cont.)

Step 4 (cont.)

It is a good idea to rename the variable to something. For instance to \( a \):

This can be done by simple editing.

We can always edit the current code.

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

Agda lead-lag (lead buffer)

— It is a good idea to rename the variable to something. For instance to \( a \):

Step 4 (cont.)

Example 1 (cont.)

Step 4 (cont.)

It is a good idea to rename the variable to something. For instance to \( a \):

Step 4 (cont.)

Example 1 (cont.)
Step 5 (Cont.):

We can inspect the context.

The context contains everything we can use when solving our goal. It contains:

- \( A :: Type \)
- \( f :: A ! A \)
- \( a :: A \)

Since we are defining a function of type \( A \) depending on \( a :: A \), we can use \( a \).

This can be used.

The type checker allows defining functions recursively, independently of whether the recursion terminates or not. This appears, since the type checker allows defining functions recursively.

On the last slide, we had \( a \) in the context.

Termination Check (Cont.)
Example 1, Using Rules (Cont.)

In Agda step 1, we stated our goal:

\[ V \leftarrow V : \text{id} \]

We write for this something \( V \) and get as conclusion of our derivation:
\[ \{ i \ i \} = V \leftarrow V : f \]

In Agda step 2, we state our goal:

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

In Agda step 3, we postulated \( A : \text{id} \).

Critical Systems, CS 411, Lent Term 2003, Sec.B2

\[ \text{Example 1, Using Rules (Cont.)} \]

Example 1, Using Rules (Cont.)

In Agda step 3 and 4 we replaced:
\[ \{ i \ i \} \leftarrow (V :: v') \gamma \]

In Agda step 5, we replaced:
\[ \{ i \ i \} \leftarrow (V :: v) \gamma \]

Essentially it allows to derive if \( y : x \) of the context that \( x \).

The assumption rule will be discussed later:

\[ V \leftarrow V : \text{id}(V : v) \gamma \]

\[ V : v \leftarrow V : \text{id} \]

In terms of rules, this means that we replace \( V : \text{id} \) by \( v \).

\[ v \leftarrow (V :: v) \gamma = V \leftarrow V :: f \]

In Agda step 5, we replaced:

\[ \{ i \ i \} \leftarrow (V :: v') \gamma \]

\[ \{ i \ i \} \leftarrow (V :: v) \gamma \]

\[ :\{ i \ i \} \leftarrow (V :: v') \gamma \]

\[ :\{ i \ i \} \leftarrow (V :: v) \gamma \]

and are done.

\[ \text{We obtain:} \]

\[ V \leftarrow V : \text{id} \]

\[ \text{We write in into the goal and then use the command} \]

\[ \text{of \text{id} in} \]

\[ \text{We take} \].

\[ \text{Now everything with result type} A \]

\[ \text{which has at the right side of the} \]

\[ \text{function} \]

\[ \text{result type} A \]

\[ \text{which has at the right side of the} \]

\[ \text{function} \]

\[ \text{result type} A \]

\[ \text{which has at the right side of the} \]

\[ \text{function} \]
We consider a derivation of
\[
\begin{align*}
     \{ i \} x & \quad (V \leftarrow (V \leftarrow V) :: x) \gamma = \\
           V & \leftarrow (V \leftarrow (V \leftarrow V)) :: f
\end{align*}
\]

We obtain
\[\text{Step } 1\]

We state our goal:
\[\text{proposition } A :: \text{Type}\]

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

\[\text{Step } 1\]

(See example \texttt{example5example2.agda})

\[\text{We consider a definition of } V \leftarrow (V \leftarrow (V \leftarrow V)) :: (\text{universe } (\text{universe } A) \text{ :: Type}) \gamma \]

\[\text{Step } 2\]

So that \texttt{agda-realizes} this change:

\[\text{We rename the variable } i \text{ to } x \text{ and use \texttt{agda-load-buffer} (load buffer)}\]

\[\text{Step } 2\]

\[\text{cont}.\]

\[\text{cont}.\]

\[\text{cont}.\]
Step 4:

- The type of the new goal is $\mathit{A} \rightarrow \mathit{A}$. This is since $x :: (\mathit{A} \rightarrow \mathit{A}) \rightarrow \mathit{A}$ needs to be applied to an element of type $\mathit{A}$. We try $\mathit{a} :: \mathit{A}$.

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

$$f :: ((\mathit{A} \rightarrow \mathit{A}) \rightarrow \mathit{A}) \rightarrow \mathit{A} = (x :: (\mathit{A} \rightarrow \mathit{A}) \rightarrow \mathit{A}) \rightarrow x (h :: f !!! g) !!! f$$

We wanted to define an element of $\mathit{A} \rightarrow \mathit{A}$ so the domain of the $\lambda$ term

$$\{ \{ \cdot \} \} \left( (\mathit{V} :: \mathit{A}) \mathit{x} \left( \mathit{V} \mathit{V} \mathit{V} \right) \mathit{x} \right) = \mathit{V} \left( \mathit{V} \mathit{V} \mathit{V} \right) \mathit{x} \mathit{a} \mathit{h}$$

Step 4 (Cont.)
Example 2, Using Rules

Postulating $\Gamma \vdash A$ corresponds to assuming $A$ in the rules without deriving it.

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

$$\Gamma \vdash (\forall x : \mathbb{A} \Rightarrow \mathbb{B}) x \vdash \Gamma \vdash \mathbb{B}$$

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

$\{i \mid i\} \vdash (\forall x : \mathbb{A} \Rightarrow \mathbb{B}) x$.

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

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

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

$\{i \mid i\} \vdash (\forall x : \mathbb{A} \Rightarrow \mathbb{B}) x$.

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

$\{i \mid i\} \vdash (\forall x : \mathbb{A} \Rightarrow \mathbb{B}) x$.

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

The last top judgement can be derived by an assumption rule (more about this later).
Finally we used rewrite with \(a\), which replaced the goal by \(a\).

Example 2, Using Rules

This corresponds to replacing \(a\) by \(a\).

Critical Systems, CS 411, Lent Term 2003, Sec. B2

Example 2, Using Rules

(See exampleProductIntro.agda)

The right hand derivation can again be derived by an assumption rule (more about this later).

\[
\begin{align*}
V &\;\vdash\; (V \mapsto (V \mapsto V)) : (V' (V : V) x (V \mapsto (V \mapsto V) : x) \\
V &\;\vdash\; V : (V' (V : V) x (V \mapsto (V \mapsto V) : x) \\
V &\;\vdash\; V : (V' (V : V) x (V \mapsto (V \mapsto V) : x)
\end{align*}
\]

Finally we used rewrite on \(a\), which replaced the goal by \(a\).

Example 3, Using Rules

We derive an element of type \(A \times B = \{a : A, b : B\}\).

Example 3 (Cont.)

We introduce the product of \(A, B\):

\(\text{AB} : \text{Type} = \text{sig} a : A; b : B\)
Step 3:

We use intro.

A nelement of $A \not\in B$ will be of the form

$$(a_0 :: A) \not\in (b_0 :: B) \not\in f \not\in g$$

which is introduced by two introduction steps. Agda will immediately carry out both of them, which is introduced by two introduction steps.

After applying agda-solve and renaming of variables we get

$$\{ i \cdot i \} \not\in (f \not\in g)$$

Step 3 (cont.)

Step 4:

Thenewgoalisoftype

$AB$ which is a record type.

Elements of type $AB$ introduced by the introduction principle will have

- the form

$\text{record} \varphi \not\in \varphi$,

where $\varphi$ is a record type.

- $\varphi$ is a record type.

- new goal is of type $AB$ which is a record type.

Step 4 (cont.)
Step 5 (Cont.):

We insert $a$, use renaming and solve the first goal.

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

Using intro again means that we place \( d_1 \) by \( h \), \( d_2 \) by \( i \), which can be introduced by an introduction rule:

\[
\begin{align*}
\text{a}_0 & : A ; b_0 : B \\
\text{a}_0 & : A ; a_0 : A \cdot B \\
\text{b}_0 & : B ; b_0 : B \cdot \text{a}_0 \\
\text{a}_0 & : A \cdot B ; a_0 : A \cdot B \\
\text{b}_0 & : B \cdot \text{a}_0 ; b_0 : B \cdot \text{a}_0 \\
\end{align*}
\]

\[ 
\text{Step I} : \\
\]

Example 4 (cont.)

We derive an element of type \( \text{prop} \) from \( \text{prop} \).

Where \( BC \) is the product of \( B \) and \( C \), (see exampleProductElim.agda).

\[ 
\begin{align*}
\text{a}_0 & : A ; b_0 : B ; c_0 : C \\
\text{a}_0 & : A ; a_0 : A \cdot B \cdot C \\
\text{b}_0 & : B ; b_0 : B \cdot \text{a}_0 ; c_0 : C \\
\text{b}_0 & : B \cdot \text{a}_0 ; b_0 : B \cdot \text{a}_0 \cdot C \\
\end{align*}
\]

\[ 
\text{Example 4 (cont.)}
\]

Example 3, Using Rules (Cont.)

Solving the goals by refining them with \( a \cdot \) means that we replace \( c \) by \( b \).

\[ 
\]

\[ 
\text{Example 4 (cont.)}
\]
Example 4 (Cont.)

Step 2:

Our goal is:

\[ f :: (A \rightarrow B \rightarrow C) \rightarrow A \rightarrow B \]

\[ f = g \]

---

Example 4 (Cont.)

Step 3:

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

\[ f :: (A \rightarrow B \rightarrow C) \rightarrow A \rightarrow B \]

\[ f = \]
Step 6: 

For solving the first goal (definition of \(bc\)) we can refine \(x\) to an element of type \(BC\).

\[
\text{let } bc :: \text{BC} = x \text{ in } f :: (A \to BC) \to A \to B \overset{f}{\Rightarrow} (a :: A \to BC) \to (a :: A \to (BC \to V :: x) = B \to V \to (BC \to V :: x) : f)
\]

We get •

In order to get close to the derivations, we omit in the Agda derivation the let expression, and replace in the body of \(bc\) by its definition \((x:a)\).

In our rule calculus we don’t introduce a let construction (we could add this).

Step 7: •

The new goal can be solved by refining \(x\) with variable \(a\).
Using rules we start with our goal $d_0$.

\[ B \leftarrow V \leftarrow ((C \times B) \leftarrow V) : p'(V : a)x((C \times B) \leftarrow V : x) \]

This can be introduced by two applications of elimination rules:

- In our rule calculus, this needs $\alpha$.
- In Agda, we then replace the goal corresponding to $d_0$ by $\lbrack \alpha \rbrack$.
Similarly, for $=$ $x$: $A$, $y$: $B$, the expression $z$: $D$ stands for the context $x$: $A$, $y$: $B$, $z$: $D$.

$\emptyset$ is the empty context (no variables are bound in it).
Example Derivation (Context Rules)

The following derives:

\[ x : N \]
\[ y : N \]
\[ z : N \]

Context

\[ N : \text{Type} \]

Example Derivation (Assumption Rule; Cont)

The full derivation of first judgement on the previous slide is as follows:

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

Context

\[ N : \text{Type} \]
\[ x : N \]
\[ y : N \]
\[ z : N \]

Context

\[ N : \text{Type} \]

Example Derivation (Assumption Rule)

We extend the derivation of B2-124 to a derivation of:

\[ z \leq N : \text{Type} \]
\[ y : N \]
\[ x : N \]

Context

\[ N : \text{Type} \]

The following derives:

\[ x \]
\[ y \]
\[ z \]

Context

\[ N : \text{Type} \]
When we define a function:

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

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

This is an application of the assumption rule:

\[
\frac{\mathcal{V} : \mathcal{A}(\forall x : \mathcal{A}) \mathcal{Y}}{\mathcal{V} : \mathcal{A} \Leftarrow \mathcal{V} : \mathcal{A}}
\]

When solving \( f!!g \) we essentially define \( a :: A \) in the context.

\[
\frac{\mathcal{V} : \mathcal{A} \Leftarrow \mathcal{V} : \mathcal{A}}{\mathcal{V} : \mathcal{A} \Leftarrow \mathcal{V} : \mathcal{A}}
\]

More generally we might in the derivation of \( a :: A \) make

\[
\mathcal{V} : \mathcal{A} \Leftarrow \mathcal{V} : \mathcal{A}
\]
Weakening Rule

\[ \frac{\text{\textit{Context}}}{\text{\textit{Context}}} \]

\[ \frac{0}{0} \]

Remark: One can in fact show that the Thinning rule can be weakly derived.

\[ \text{\textit{Weakly derived}} \]

\[ \text{\textit{Derived means: whenever the assumptions of the rule can be derived in the complete set of rules we provide, then so well the conclusion.}} \]

\[ \text{\textit{Weakened means: whenever the assumptions of the rule can be derived from the premises the conclusion directly.}} \]

\[ \text{\textit{An exception is when we additionally assume some judgements for instance.}} \]

\[ \text{\textit{Example Derivation (Weakening Rule)}} \]

\[ \text{\textit{Example Derivation 2 (Weakening Rule)}} \]

\[ \theta \in \Gamma \]

\[ \text{\textit{The judgement } \Gamma \text{ is weakened by } \eta.}} \]

\[ \text{\textit{\eta \textit{ is a judgement: }}} \]

\[ \text{\textit{This rule allows to add an additional context piece } (\eta) \text{ to the context of \textit{an arbitrary non-dependent judgement.}} \]

\[ \frac{\theta \in \Gamma. \eta'}{\theta \in \Gamma. \eta'} \]
General Equality Rules

Reexivity

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

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

Symmetry

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

Example Derivation (General Equality Rules)

\[ N : \text{Type} \]
\[ y : N \]
\[ y : N \]
\[ y + 0 = y : N \]

Context

\[ y : N \]
\[ y : N \]
\[ y + 0 = y : N \]

\[ x : N \]
\[ x : N \]
\[ y : N \]
\[ y : N \]

Example Derivation (General Equality Rules; Cont.)

In the previous derivation, the most complicated step was:

Example Derivation (General Equality Rules; Cont.)

\[ \text{Transfer} \]

\[ V : \alpha = a \]
\[ V : q = q \]

\[ \text{Symmetry} \]

\[ V : a = q \]
\[ V : q = a \]

\[ \text{Reflexivity} \]

\[ V : a = a \]
\[ V : a = a \]
Example Derivation (General Equality Rules; Cont.)

Notethatfromthepremisesofthatrule $y : N; x : N$  $y ; y : N$  

$(x : N)$

$\text{N} : x \leftarrow \text{N} : x$  $\text{N} : x \leftarrow \text{N} : x$

$\text{y} : \text{N}$  $\text{y} : \text{N}$  

$\text{Example Derivation (Substitution)}$

\begin{align*}
[v := x] q & : [v := x] q \rightarrow [v := x] q \leftarrow [v := x] \theta \theta \\
V : \theta & \leftarrow \text{J} \theta & \leftarrow \text{J} V : x \leftarrow \text{J}
\end{align*}

\text{Substitution 3}

$\text{v} \in \text{J}, \text{L} \vdash [v := x] q = [v := x] q \leftarrow [v := x] \theta \theta \\
V : \theta \leftarrow \text{J} \theta & \leftarrow \text{J} V : x \leftarrow \text{J}
\text{Substitution 2}$

$\text{v} \in \text{J}, \text{L} \vdash [v := x] q = [v := x] q \leftarrow [v := x] \theta \theta \\
V : \theta \leftarrow \text{J} \theta & \leftarrow \text{J} V : x \leftarrow \text{J}$

\text{Substitution 1}$

The following rules can be weakly derived:

Substitution Rules

\text{Example Derivation (Substitution: Cont.)}

\begin{align*}
\text{N} : \text{N} \leftarrow \text{N} + 0 (\text{N} : \text{N}) & \quad \text{N} : \text{N} \leftarrow \text{N} + 0 (\text{N} : \text{N}) \\
\text{N} : \text{N} \leftarrow \text{N} & \quad \text{N} : \text{N} \leftarrow \text{N}
\end{align*}

\text{Example Derivation (General Equality Rules; Cont.)}

\text{Example Derivation (Substitution)}

\text{Example Derivation (Substitution: Cont.)}

\text{Example Derivation (Substitution: Cont.)}
In order to derive $x : A; y : B$ $C : Type$ we need to show:

1. $x : A$
2. $y : B$
3. A $\vdash_{Type} C : Type$

Then the judgement $x : A; y : B$ $C : Type$ implicitly contains the judgements $A : Type; x : A$ $B : Type$

::: Presuppositions

The next slide shows the presuppositions of judgements.

$\vdash_{Type} C : Type$

$\vdash_{Type} C : Type$

Example: Substitution (Cont.)

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

$N : \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha = \forall \alpha. (\vdash : \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$
Presuppositions

Judgement

Presuppositions

Example of Presuppositions

Remark on $A \land B$.

Presuppositions

Example of Presuppositions

Judgement

Presuppositions
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

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

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.

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

Addition •

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.

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

The problem is that for this we need

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

and our rules allow this only if we had

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

The problem is that for this we need

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

We would like to add operations on types, such as

\[ \text{Type} : \text{Type} \rightarrow \text{Type} \rightarrow \text{Type} \]
Instead we introduce a new type

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

\[
\text{N} \rightarrow \text{Set} 
\]

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

\[
\text{Type} 
\]

• A set is a small type.

Set is the type of sets.

Set : Type

Since Set : Type we get

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

and we can assign to prod above the type

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

We add rules asserting that if \( A : \text{Set} \) then \( A : \text{Type} \).
However, we cannot use \( \text{prod} \) in order to form the product of two sets, i.e., we cannot introduce \( \text{Set} : \text{Set} \times \text{Set} \). Since \( \text{Set} : \text{Set} \) does not hold, that would result in the same inconsistency as \( \text{Type} : \text{Type} \).

---

**Example: \( \text{prod} \)**

We can now introduce \( \text{prod} : \text{Set} \to \text{Set} \). First, we derive \( X : \text{Set} ; Y : \text{Set} \Rightarrow X : \text{Set} ; Y : \text{Set} \).

---

**Critical Systems, CS 411, Lent 2003, Sec. B2**

**B2-152**
Example: prod (cont.)

Now we can derive our desired judgement:

Example: prod (cont.)
Rules for Type as a Kind

Every Type is a Kind

A : Type  A : Kind

Closure of Kind under the dependent function type

\[ \text{Type \times Type} \vdash \text{Type} \]

\[ \forall \text{Type} \vdash \text{Kind} \]

Type is a Kind

Plus equality versions of the above rules.

Equality of Kind under the dependent function type

\[ \forall \text{Kind} \vdash \text{Kind} \]

\[ \forall \text{Type} \vdash \text{Type} \]

Critical Systems, CS 411, Lent Term 2003, Sec. B2

Hierarchies of Types (Cont.)

This can be iterated further, forming

Type = Type₁ ; Kind = Type₂ ; Type₃ ; Type₄ ;

\[ \vdots \]

\[ \text{Type} = \text{Type₁} ; \text{Kind} = \text{Type₂} ; \text{Type₃} ; \text{Type₄} ; \text{Type₅} ; \]

\[ \vdots \]

...