A sequent-calculus presentation of type-theory

Gabriel Radanne
Under the supervision of Jean-Philippe Bernardy

ENS Rennes — Chalmers University of Technology

February 20, 2014
Plan

1. An Introduction to dependent types

2. Limitations of current typecheckers
   - Efficiency issues
   - The “case decomposition” issue
   - The monolithic approach

3. NANOAgda and MICROAgda
   - Goals
   - NANOAgda
   - MICROAgda
   - Results

4. Conclusion
Imagine we want to define lists, but with guarantees on the length of the list.

We have the length operation:

\[
\left| \left[ 'a'; 'b'; 'c' \right] \right| = 3.
\]
Imagine we want to define lists, but with guarantees on the length of the list.

We have the length operation:

```
| 'a' :: 'b' :: 'c' :: [] | = 3.
```
Imagine we want to define lists, but with guarantees on the length of the list.
We have the length operation:
\[ \text{\texttt{\char92a :: \char92b :: \char92c :: \\}} = 3. \]

We can define the head function like this in OCAML:

```ocaml
let head x = match x with
  | [] -> failwith "PANIC"
  | (h::t) -> h
```
Imagine we want to define lists, but with guarantees on the length of the list.

We have the length operation:
\[
| \text{'a'} :: \text{'b'} :: \text{'c'} :: [] | = 3.
\]

We can define the head function like this in OCAML:

```ocaml
let head x = match x with
  | [] -> failwith "PANIC"
  | (h::t) -> h

head l should only be valid if |l| > 0.
```
Let’s start by natural numbers:

```agda
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat
```

three : Nat
three = Succ ( Succ ( Succ Zero ))

myVec : Vec Char three
myVec = Cons 'a' ( Cons 'b' ( Cons 'c' Nil ))
Let’s start by natural numbers:

```agda
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat

three : Nat
three = Succ (Succ (Succ Zero))
```

Let’s start by natural numbers:

```haskell
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat

three : Nat
three = Succ (Succ (Succ Zero))
```

We can now define a special kind of list:

```haskell
data Vec (A : Set) : Nat → Set where
  Nil : Vec A Zero
  Cons : {n : Nat} → A → Vec A n → Vec A (Succ n)
```
Let’s start by natural numbers:

\[
\begin{align*}
\text{data & Nat : Set where} \\
\text{Zero : Nat} \\
\text{Succ : Nat } \rightarrow \text{ Nat} \\
\text{three : Nat} \\
\text{three} &= \text{ Succ (Succ (Succ Zero))}
\end{align*}
\]

We can now define a special kind of list:

\[
\begin{align*}
\text{data & Vec (A : Set) : Nat } \rightarrow \text{ Set where} \\
\text{Nil : Vec A Zero} \\
\text{Cons : \{n : Nat\} } \rightarrow \text{ A } \rightarrow \text{ Vec A n } \rightarrow \text{ Vec A (Succ n)} \\
\text{myVec : Vec Char three} \\
\text{myVec} &= \text{ Cons 'a' (Cons 'b' (Cons 'c' Nil))}
\end{align*}
\]
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat

data Vec (A : Set) : Nat → Set where
  Nil : Vec A Zero
  Cons : {n : Nat} → A → Vec A n → Vec A (Succ n)

The head function:
head : forall {A n} → Vec A (Succ n) → A
head (Cons x xs) = x
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat

data Vec (A : Set) : Nat → Set where
  Nil : Vec A Zero
  Cons : {n : Nat} → A → Vec A n → Vec A (Succ n)

The head function:
head : forall { A n } → Vec A (Succ n) → A
head (Cons x xs) = x

head Nil ← This is a type error.
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat

data Vec (A : Set) : Nat → Set where
  Nil : Vec A Zero
  Cons : {n : Nat} → A → Vec A n → Vec A (Succ n)

When we concatenate two vectors, \( |\text{append } l \ l'| = |l| + |l'|.\)
data Nat : Set where
  Zero : Nat
  Succ : Nat → Nat

data Vec (A : Set) : Nat → Set where
  Nil : Vec A Zero
  Cons : {n : Nat} → A → Vec A n → Vec A (Succ n)

When we concatenate two vectors, \( |\text{append } l \ l'| = |l| + |l'| \).

append : forall {n m A} → Vec A n → Vec A m → Vec A (n + m)
append Nil ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
Dependent types

What have we done?

- We defined a type with a **term** as parameter: \( \text{Vec } A \ n. \)
Dependent types

What have we done?

- We defined a type with a **term** as parameter: \( \text{Vec} \ A \ n \).
- We used these values to enforce properties.
Dependent types

What have we done?

- We defined a type with a **term** as parameter: \( \text{Vec} ~ A ~ n \).

- We used these values to enforce properties... by type-checking.
What have we done?

- We defined a type with a **term** as parameter: `Vec A n`.
- We used these values to enforce properties... by type-checking.
- We manipulated these values inside the type: `Vec A (n+m)`. 
What have we done?

- We defined a type with a **term** as parameter: $\text{Vec } A \ n$.
- We used these values to enforce properties... by type-checking.
- We manipulated these values inside the type: $\text{Vec } A \ (n+m)$.

Types depends on terms.
Dependent types

Dependent types:

Strongly related to Curry-Howard Isomorphism.


Has gained popularity recently for theorem-proving with Coq, but also in programming: Agda, Idris, ATS, ...
Dependent types:

- Strongly related to Curry-Howard Isomorphism.
Dependent types:

- Strongly related to Curry-Howard Isomorphism.
Dependent types:

- Strongly related to Curry-Howard Isomorphism.
- Has gained popularity recently for theorem-proving with CoQ,
Dependent types:

- Strongly related to Curry-Howard Isomorphism.
- Has gained popularity recently for theorem-proving with Coq,
- but also in programming: Agda, Idris, ATS,...
Dependent types:

- Strongly related to Curry-Howard Isomorphism.
- Has gained popularity recently for theorem-proving with Coq,
- but also in programming: Agda, Idris, ATS,…
Limitations of current typecheckers

- Efficiency issues
- The “case decomposition” issue
- The monolithic approach

NANO Agda and MICRO Agda
- Goals
- NANO Agda
- MICRO Agda
- Results

Conclusion
Agda’s type checker uses a natural deduction style:

- Inference duplicates parts of terms.
- These parts are not shared in the Agda core representation anymore.
- Typechecking must be done multiple times, causing performance penalties.

\[
\lambda x. (f \ x \ x) \ (\ldots)
\]

\[
f \ (\ldots) \ (\ldots)
\]
The “case decomposition” issue

Natural deduction style makes propagating typing constraints to subterms difficult.
For example, Agda’s typechecker has no knowledge of which branch was taken while it typechecks the body of a case.

\[
\text{myFun } x \ \text{with } f \ x \\
\text{... | } \text{Foo} = \text{( No knowledge that } f \ x \equiv \text{Foo )} \\
\text{... | } \text{Bar} = \text{( No knowledge that } f \ x \equiv \text{Bar )}
\]
**The monolithic approach**

**Agda** currently does not have a core language that can be reasoned about and formally verified. **Coq**, on the other hand, is built as successive extensions of a core language (CIC).

![Diagram showing the difference between Agda and Coq's approach]

**Agda**

Coq

CIC
**The monolithic approach**

**AGDA** currently does not have a core language that can be reasoned about and formally verified. **Coq**, on the other hand, is built as successive extensions of a core language (CIC).
Goals

Our goals are to have a language that is:

- A type-theory: Correctness should be expressible via types.
Our goals are to have a language that is:

- A type-theory: Correctness should be expressible via types.
- Low-level: One should be able to translate high-level languages into this language while retaining properties such as run-time behaviour, complexity, etc.
Our goals are to have a language that is:

- A type-theory: Correctness should be expressible via types.
- Low-level: One should be able to translate high-level languages into this language while retaining properties such as run-time behaviour, complexity, etc.
- Minimal: The language should be well defined and it should be possible to formally verify the type-checking algorithm.
id : (a : Set) → a → a
id _ x = x

in Agda
NANOAgda

id : (a : Set) → a → a
id _ x = x

in AGDA

TERM
f = \lambda a \rightarrow (f' = \lambda x \rightarrow (r=x; r); f') ;

f

TYPE
set = \star_0 ;
f_ty = (a : set) \rightarrow (a' = a ; a2a = (x : a') \rightarrow a' ;
   a2a ) ;
f_ty

in NANOAgda
An Introduction to dependent types

Limitations of current typecheckers

NANOAgda and microAgda

Conclusion

nanoAgda

id : (a : Set) → a → a

id _ x = x

in Agda

TERM
f = λa → (f' = λx → (r=x; r); f') ;

f

TYPE
set = ⋆0 ;

f_ty = (a : set) → (a' = a ; a2a = (x : a') → a' ; a2a ) ;

f_ty

in nanoAgda
**nanoAgda**

\[ \text{id : (a : Set) \to a \to a} \]

\[ \text{id _ x = x} \]

in **Agda**

\[ \text{TERM} \]
\[ f = \lambda a \to ( \]
\[ \quad f' = \lambda x \to (r=x; r); \]
\[ \quad f' ) ; \]
\[ f \]

\[ \text{TYPE} \]
\[ \text{set} = \star_0 ; \]
\[ \text{f_ty} = (a : \text{set}) \to ( \]
\[ \quad a' = a ; \]
\[ \quad a2a = (x : a') \to a' ; \]
\[ \quad a2a ) ; \]
\[ \text{f_ty} \]

in **nanoAgda**
An Introduction to dependent types

Limitations of current typecheckers

NanoAgda and MicroAgda

Conclusion

NanoAgda

id : (a : Set) → a → a

id _ x = x

in Agda

TERM
f = λa → (
    f' = λx → (r=x; r);
    f') ;

f

TYPE
set = *0 ;
f_ty = (a : set) → (a' = a ;
a2a = (x : a') → a' ;
a2a)

f_ty

in NanoAgda
Sequent calculus

There are various definitions of sequent calculus. Here, we mean that every intermediate result or sub-term are bound to a variable.

\[ \lambda x. (f \ x \ x) (\ldots) \]

in natural deduction style
Sequent calculus

There are various definitions of sequent calculus. Here, we mean that every intermediate result or sub-term are bound to a variable.

\[ \lambda x. (f \ x \ x) \ (\ldots) \]

- in natural deduction style
- \[ f \ (\ldots) (\ldots) \]
- \[ \text{let } x' = (\ldots) \text{ in } f \ x' \ x' \]
  - in sequent calculus style
Presentation of the language

- Variables: Hypotheses $x$ and Conclusions $\overline{x}$
Presentation of the language

- **Variables**: Hypotheses $x$ and Conclusions $\overline{x}$

- **Functions**: $\lambda x. t$  $(f \overline{x})$  $(x : \overline{Y}) \rightarrow T$
Presentation of the language

- **Variables**: Hypotheses $x$ and Conclusions $\overline{x}$
  - **Functions**: $\lambda x. t$  $(f \overline{x})$  $(x : \overline{Y}) \rightarrow T$
  - **Pairs**: $(\overline{x}, \overline{y})$  $x.1$  $(x : \overline{Y}) \times T$
Presentation of the language

- **Variables**: Hypotheses $x$ and Conclusions $\bar{x}$

  - Functions: $\lambda x.t$ \hspace{1cm} $(f \bar{x})$ \hspace{1cm} $(x : \bar{Y}) \rightarrow T$
  - Pairs: $(\bar{x}, y)$ \hspace{1cm} $x.1$ \hspace{1cm} $(x : \bar{Y}) \times T$
  - Enumerations: `l` \hspace{1cm} case \hspace{1cm} `{l_1, l_2, \ldots}`
Presentation of the language

- **Variables**: Hypotheses $x$ and Conclusions $\bar{x}$
  
  - Functions $\lambda x. t$  
    
  - Pairs $(\bar{x}, \bar{y})$  
    
  - Enumerations `l`  
    
- **Constructions and Destructions**:  
  
  let $\bar{x} = c$ and let $x = d$
Presentation of the language

- **Variables**: Hypotheses $x$ and Conclusions $\bar{x}$
  
  - Functions: $\lambda x. t$ (f $\bar{x}$) $(x : \overline{Y}) \rightarrow T$
  
  - Pairs: $(\bar{x}, \bar{y})$ $x.1$ $(x : \overline{Y}) \times T$
  
  - Enumerations: 
    
    - case $\{ l_1, l_2, \ldots \}$

- **Constructions and Destructions**: 
  
  - let $\bar{x} = c$ and let $x = d$

- **Universes**: 
  
  - $\star_i$ with $i \in \mathbb{N}$ 
  
  - $\star_0$ is equivalent to Set
Presentation of the language

- **Variables:** Hypotheses $x$ and Conclusions $\overline{x}$
  - Functions: $\lambda x.t$  \hspace{1cm} $(f \ x)$  \hspace{1cm} $(x : \overline{Y}) \rightarrow T$
  - Pairs: $(\overline{x}, \overline{y})$  \hspace{1cm} $x.1$  \hspace{1cm} $(x : \overline{Y}) \times T$
  - Enumerations: `/`  \hspace{1cm} case  \hspace{1cm} $\{l_1, l_2, \ldots \}$

- **Constructions and Destructions:**
  - `let x = c` and `let x = d`

- **Universes:**
  - $\star_i$ with $i \in \mathbb{N}$  \hspace{1cm} $\star_0$ is equivalent to Set

- **Relation between Conclusions and Hypotheses:**
  - `let $\overline{x} = y$` A conclusion can be defined as an hypothesis.
Presentation of the language

- **Variables:** Hypotheses \( x \) and Conclusions \( \overline{x} \)
  
  **Functions** \( \lambda x. t \) \( (f \overline{x}) \) \( (x : \overline{Y}) \rightarrow T \)
  
  **Pairs** \( (\overline{x}, \overline{y}) \) \( x.1 \) \( (x : \overline{Y}) \times T \)
  
  **Enumerations** \( l \) case \( \{ 'l_1, 'l_2, \ldots \} \)

- **Constructions and Destructions:**
  
  let \( \overline{x} = c \) and let \( x = d \)

- **Universes:**
  
  \( \star_i \) with \( i \in \mathbb{N} \) \( \star_0 \) is equivalent to Set

- **Relation between Conclusions and Hypotheses:**
  
  let \( \overline{x} = y \) A conclusion can be defined as an hypothesis.
  
  let \( x = (\overline{y} : \overline{Z}) \) The cut construction.
MICROAGDA

A new syntax, easier to manipulate, and that can be translated easily into NANOAGDA.
A new syntax, easier to manipulate, and that can be translated easily into NANOAGDA.

\[
\text{id : (a : Set) \to a \to a} \\
\text{id \_ x = x}
\]

in AGDA

\[
\text{TERM} \\
\lambda a \to \lambda x \to x
\]

\[
\text{TYPE} \\
(a : \star_1) \to (x : a) \to a
\]

in MICROAGDA
A new syntax, easier to manipulate, and that can be translated easily into NANOAGDA.

id : (a : Set) → a → a
id _ x = x

in AGDA

TERM
\( \lambda a \to \lambda x \to x \)

TYPE
(\( a : \star_1 \) → (\( x : a \) → a))

in MICROAGDA

TERM
f = \( \lambda a \to (\)
  f' = \( \lambda x \to (r=x; r); \)
  f' (\)

f

TYPE
set = \( \star_0 \);

f_ty = (\( a : set \) → (\( a' = a \);
  a2a = (\( x : a' \) → a');
  a2a

f_ty

in NANOAGDA
MICROAGDA

A new syntax, easier to manipulate, and that can be translated easily into NANOAGDA.

\[
\text{id} : (a : \text{Set}) \to a \to a
\]
\[
\text{id} \_ x = x
\]

in AGDA

\[
\text{TERM}
\lambda a \to \lambda x \to x
\]

\[
\text{TYPE}
(a : \star_1) \to (x : a) \to a
\]

in MICROAGDA

\[
\text{TERM}
f = \lambda a \to (f' = \lambda x \to (r=x; r); f')
\]

\[
\text{TYPE}
set = \star_0 ;
\]

\[
f_{\text{ty}} = (a : \text{set}) \to (a' = a ; a2a = (x : a') \to a' ; a2a ) ;
\]

in NANOAGDA
A new syntax, easier to manipulate, and that can be translated easily into NanoAgda.

\[
\text{id} : (a : \text{Set}) \rightarrow a \rightarrow a
\]
\[
\text{id}_\_x = x
\]

in Agda

\[
\text{TERM}
\]
\[
\lambda a \rightarrow \lambda x \rightarrow x
\]

\[
\text{TYPE}
\]
\[
(a : \star_1) \rightarrow (x : a) \rightarrow a
\]

in MicroAgda

\[
\text{TERM}
\]
\[
f = \lambda a \rightarrow (f' = \lambda x \rightarrow (r=x; r); f')
\]

\[
\text{TYPE}
\]
\[
\text{set} = \star_0 ;
\text{f_ty} = (a : \text{set}) \rightarrow (a' = a ; a2a = (x : a') \rightarrow a' ; a2a)
\]

in NanoAgda
A new syntax, easier to manipulate, and that can be translated easily into NANOAGDA.

\[
\begin{align*}
\text{id} & : (a : \text{Set}) \to a \to a \\
\text{id} \_ x &= x \\
\text{in AGDA}
\end{align*}
\]

\[
\begin{align*}
\text{TERM} \\
\lambda a \to \lambda x \to x \\
\text{TYPE} \\
(a : \star_1) \to (x : a) \to a
\end{align*}
\]

\[
\begin{align*}
\text{TERM} \\
f &= \lambda a \to ( \\
f' &= \lambda x \to (r=x; r); \\
f' \ ) ; \\
f \\
\text{TYPE} \\
\text{set} &= \star_0 ; \\
f_{-ty} &= (a : \text{set}) \to ( \\
a' &= a ; \\
a2a &= (x : a') \to a' ; \\
a2a
\end{align*}
\]

\[
\begin{align*}
\text{in MICROAGDA}
\end{align*}
\]

\[
\begin{align*}
\text{in NANOAGDA}
\end{align*}
\]
Results

- We implemented a typechecker and evaluator for \texttt{NANOAGDA}.
- We introduced a new intermediate language: \texttt{MICROAGDA}.
- We exhibited some examples that don’t typecheck in \texttt{AGDA} but typecheck in \texttt{NANOAGDA}. 
Future work

- Precisely evaluate the efficiency of this new approach.
- Prove subject-reduction of \texttt{NANOAgda} (in \texttt{Coq}).
- Introduce recursion.
- Experiment with extensions of the type system (linear, colors, \ldots).
Questions ?
Questions ?
How to encode sum types

data MySumtype (s : Set) : Set where
  Foo : s → MySumtype s
  Bar : MySumtype s

  in Agda
How to encode sum types

data MySumtype (s : Set) : Set where
  Foo : s → MySumtype s
  Bar : MySumtype s

in Agda

TERM
Unit_t = { 'unit } ;
Unit_ty = *0 ;
Unit = Unit_t : Unit_ty ;
f = λs → (  
tag = { 'Foo , 'Bar } ;
f' = (c : tag) ×  
(case c of {  
  'Foo → s' = s ; s' .  
  'Bar → Unit' = Unit ;  
  Unit'  
})) ;
f') ;
f
TYPE
star0 = *0 ;
f_ty = ( s : star0) → star0 ;
f_ty
How to encode stupidly simple sum types

data SimpleSum : Set where
  Foo : Nat → SimpleSum
  Bar : Nat → SimpleSum

in Agda
How to encode stupidly simple sum types

data SimpleSum : Set where
   Foo : Nat → SimpleSum
   Bar : Nat → SimpleSum

   in Agda

   TERM
   { ‘Foo ; ‘Bar } × Nat
   Type
   *0

   in MicroAgda
How to encode stupidly simple sum types

data SimpleSum : Set where
  Foo : Nat → SimpleSum
  Bar : Nat → SimpleSum

  in Agda

TERM

\{ 'Foo ; 'Bar \} \times Nat

Type

\star_0

in MicroAgda
How to encode sum types – 2\textsuperscript{nd} edition

data MySumtype (s : Set) : Set where
  Foo : s \rightarrow MySumtype s
  Bar : MySumtype s

in AGDA
How to encode sum types – 2nd edition

```agda
{data MySumtype (s : Set) : Set where
  Foo : s → MySumtype s
  Bar : MySumtype s
}

{TERM
  Unit = { ‘unit } : ∗₀ ;
  λs → ( c : { ‘Foo , ‘Bar } ) ×
    ( case c of {
      ‘Foo → s.
      ‘Bar → Unit.
    } )
}

{TYPE
  ∗₀ → ∗₀
}
```

How to encode sum types – 2\textsuperscript{nd} edition

data MySumtype (s : Set) :
  Set where
\begin{align*}
\text{Foo} & : s \rightarrow \text{MySumtype } s \\
\text{Bar} & : \text{MySumtype } s
\end{align*}

in AGDA

TERM
\begin{align*}
\text{Unit} & = \{ \text{‘unit } \} : \star_0 \\
\lambda s \rightarrow ( c : \{ \text{‘Foo , ‘Bar } \} ) \times \\
& ( \text{case } c \text{ of } \\
& \begin{align*}
& \text{‘Foo } \rightarrow s. \\
& \text{‘Bar } \rightarrow \text{Unit.}
\end{align*}

) \\
\end{align*}

TYPE
\begin{align*}
\star_0 & \rightarrow \star_0
\end{align*}

in MICROAGDA
Questions ?
Environment extension

\[ \Gamma : x \mapsto \overline{y} \quad \text{The context heap, containing the type of hypotheses.} \]
\[ \gamma_c : \overline{x} \mapsto c \quad \text{The heap from conclusion to constructions.} \]
\[ \gamma_a : x \mapsto y \quad \text{The heap for aliases on hypotheses.} \]
\[ \gamma_d : x \mapsto d \quad \text{The heap from hypotheses to cuts and destructions.} \]

\[ \gamma + (x : \overline{Y}) = \gamma \text{ with } \Gamma \leftarrow (x : \overline{Y}) \]

\[ \gamma + (x = d) = \gamma \text{ with } \gamma_a \leftarrow (x = y) \quad \text{if } (y = d) \in \gamma_d \]
\[ = \gamma \text{ with } \gamma_d \leftarrow (x = d) \quad \text{otherwise} \]

\[ \gamma + (\overline{x} = c) = \gamma \text{ with } \gamma_c \leftarrow (\overline{x} = c) \]

\[ \gamma + (\text{'l} = x) = \gamma \quad \text{if } (\text{'l} = x) \in \gamma_c \]
\[ = \bot \quad \text{if } (\text{'m} = x) \in \gamma_c \text{ for } \text{'l} \neq \text{'m} \]
\[ = \gamma \text{ with } \gamma_c \leftarrow (\text{'l} = x) \quad \text{otherwise} \]
Reduction rules

**EvalCase**

\[
\frac{h(x) = ('l_i : _) \quad h + ('l_i = x) \vdash t_i \leadsto h' \vdash \bar{x}}{h \vdash \text{case } x \text{ of } \{ 'l_i \mapsto t_i \} \leadsto h' \vdash \bar{x}}
\]

**AbstractCase**

\[
\frac{h(x) \neq ('l_i : _) \quad \forall i \quad h + ('l_i = x) \vdash t_i \leadsto h'_i \vdash \bar{x}_i}{h \vdash \text{case } x \text{ of } \{ 'l_i \mapsto t_i \} \leadsto \{ h'_i \vdash \bar{x}_i \}}
\]

**EvalDestr**

\[
\frac{h \vdash d \leadsto h' \vdash t' \quad h' + (x = t') \vdash t \leadsto h'' \vdash \bar{x}}{h \vdash \text{let } x = d \text{ in } t \leadsto h'' \vdash \bar{x}}
\]

**AddDestr**

\[
\frac{h \vdash d \not\leadsto h' \vdash t' \quad h + (x = d) \vdash t \leadsto h' \vdash \bar{x}}{h \vdash \text{let } x = d \text{ in } t \leadsto h' \vdash \bar{x}}
\]

**AddConstr**

\[
\frac{h + (\bar{x} = c) \vdash t \leadsto h' \vdash \bar{x}}{h \vdash \text{let } \bar{x} = c \text{ in } t \leadsto h' \vdash \bar{x}}
\]

**Concl**

\[
\frac{h \vdash \bar{x} \leadsto h \vdash \bar{x}}{}
\]

**EvalProj\_1**

\[
\frac{h(y) = ((\bar{z}, \bar{w}) : _)}{h \vdash y.1 \leadsto h \vdash \bar{z}}
\]

**EvalProj\_2**

\[
\frac{h(y) = ((\bar{z}, \bar{w}) : _)}{h \vdash y.2 \leadsto h \vdash \bar{w}}
\]

**EvalApp**

\[
\frac{h(y) = (\lambda w. t : _)}{h \vdash (y \bar{z}) \leadsto h' \vdash \bar{x}}
\]
Equality rules

\[ \gamma \vdash \text{let } x = d \text{ in } t = t' \quad \rightarrow \quad \gamma' + (x = t'') \vdash t = t' \]
\[ \gamma \vdash \text{let } x = c \text{ in } t = t' \quad \rightarrow \quad \gamma + (x = c) \vdash t = t' \]
\[ \gamma \vdash \text{case } x \text{ of } \{ 'l_i \mapsto t_i \} = t \quad \rightarrow \quad \forall i \quad \gamma + (x = 'l_i) \vdash t_i = t \]
\[ \gamma \vdash x = y \quad \rightarrow \quad x \equiv y \land \gamma \vdash \gamma_c(x) = \gamma_c(y) \]

\[ \gamma \vdash 'l = 'l \quad \rightarrow \quad \text{true} \]
\[ \gamma \vdash *_{i} = *_{j} \quad \rightarrow \quad i = j \]
\[ \gamma \vdash x = y \quad \rightarrow \quad x \not\equiv y \]
\[ \gamma \vdash \lambda x.t = \lambda y.t' \quad \rightarrow \quad \gamma + (x = y) \vdash t = t' \]
\[ \gamma \vdash (x, y) = (x', y') \quad \rightarrow \quad \gamma \vdash x = x' \land \gamma \vdash y = y' \]
\[ \gamma \vdash (x : y) \rightarrow t = (x' : y') \rightarrow t' \quad \rightarrow \quad \gamma \vdash y = y' \land \gamma + (x = x') \vdash t = t' \]
\[ \gamma \vdash (x : y) \times t = (x' : y') \times t' \quad \rightarrow \quad \gamma \vdash y = y' \land \gamma + (x = x') \vdash t = t' \]
\[ \gamma \vdash \{ 'l_i \} = \{ 'm_i \} \quad \rightarrow \quad \forall i \quad 'l_i = 'm_i \]

\[ \gamma \vdash \lambda x.t = y \quad \rightarrow \quad \gamma + (x = x) + (z = y \, x) \vdash t = z \]
\[ \gamma \vdash (x, x') = y \quad \rightarrow \quad \gamma + (z = y.1) \vdash x = z \land \gamma + (z = y.2) \vdash x' = z \]
Figure: Typechecking a term: $\gamma \vdash t \iff T$
Typing rules

\[
\text{App} \quad \Gamma(y) = (z : X) \to T \quad \gamma \vdash z \iff X \\
\quad \gamma \vdash y \overline{z} \Rightarrow T
\]

\[
\text{Proj}_1 \quad \Gamma(y) = (z : X) \times T \\
\quad \gamma \vdash y.1 \Rightarrow \overline{X} \\
\text{Proj}_2 \quad \Gamma(y) = (z : X) \times T \\
\quad \gamma \vdash y.2 \Rightarrow T
\]

\[
\text{Cut} \quad \gamma \vdash x \iff X \\
\gamma \vdash (\overline{x} : X) \Rightarrow \overline{X}
\]

**Figure:** Inferring the type of a destruction: \(\gamma \vdash d \Rightarrow T\).

\[
\text{TyDestr} \quad \gamma + (x = d) \vdash c \iff T \\
\text{TyConstr} \quad \gamma + (\overline{x} = c) \vdash c \iff T
\]

\[
\text{TyCase} \quad \forall i \quad \gamma + ('l_i = x) \vdash c \iff T_i \\
\gamma \vdash x \Rightarrow \{ 'l_i \mapsto T_i \} \\
\gamma \vdash c \iff \text{case } x \text{ of } \{ 'l_i \mapsto T_i \}
\]

\[
\text{TyConcl} \quad \gamma_c(\overline{x}) = C \\
\gamma \vdash c \iff C
\]

\[
\text{Infer} \quad \Gamma(x) = \overline{X} \\
\gamma \vdash \overline{x} \Rightarrow T \\
\gamma \vdash x \iff T
\]

**Figure:** Typechecking a construction against a term: \(\gamma \vdash c \iff T\).
**Typing rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PAIR</strong></td>
<td>[ \gamma \vdash \bar{x} \equiv \bar{x} \quad \gamma + (x = (\bar{y} : \bar{X})) \vdash \bar{z} \equiv T \quad \gamma \vdash (\bar{y}, \bar{z}) \equiv (x : \bar{X}) \times T ]</td>
</tr>
<tr>
<td><strong>SIGMA</strong></td>
<td>[ \gamma \vdash \bar{y} \equiv \star_i \quad \gamma + (x : \bar{y}) \vdash t \equiv \star_i \quad \gamma \vdash (x : \bar{y}) \times t \equiv \star_i ]</td>
</tr>
<tr>
<td><strong>FLN</strong></td>
<td>[ \gamma \vdash {'l_i} \equiv \star_i ]</td>
</tr>
<tr>
<td><strong>UNIVERSE</strong></td>
<td>[ i &lt; j \quad \gamma \vdash \star_i \equiv \star_j ]</td>
</tr>
<tr>
<td><strong>LAMBDA</strong></td>
<td>[ \gamma + (y : \bar{X}) + (x = y) \vdash t \equiv T \quad \gamma \vdash \lambda y. t \equiv (x : \bar{X}) \to T ]</td>
</tr>
<tr>
<td><strong>LABEL</strong></td>
<td>[ 'l \in { 'l_i } \quad \gamma \vdash 'l \equiv { 'l_i } ]</td>
</tr>
</tbody>
</table>

**Figure:** Typechecking a construction against a construction: \[ \gamma \vdash c \equiv C. \]