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:

$|$ `['a' ; 'b' ; 'c']` $| = 3$.

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:
| `'a' :: 'b' :: 'c' :: []` | = 3.

We can define the head function like this in 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:
| `'a' :: 'b' :: 'c' :: []` | = 3.

We can define the head function like this in OCAML:

```
let head x = match x with
  | [] -> failwith "PANIC"
We want the type-system to ensure this doesn't happen.
  | (h::t) -> h
```

`head l` should only be valid if |`l`| > 0.

Let's start by natural numbers:

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

Let's start by natural numbers:

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

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

Let's start by natural numbers:

```
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:

```
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:

```
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:

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

myVec : Vec Char three
myVec = Cons 'a' (Cons 'b' (Cons 'c' Nil))
```

```
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, $|\texttt{append l l'}| = |\texttt{l}| + |\texttt{l'}|$.

```
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, |`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: `Vec A n`.

Dependent types

What have we done?

- We defined a type with a **term** as parameter: `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: `Vec A n`.
- We used these values to enforce properties… by type-checking.

Dependent types

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)`.

Dependent types

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)`.

Types depends on terms.

Dependent types

Dependent types:

Dependent types

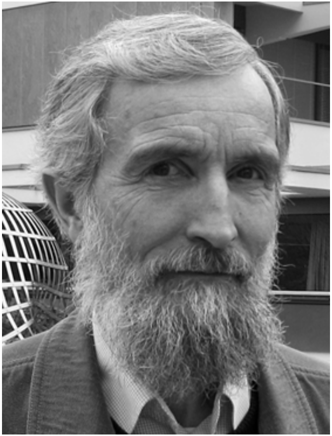
Dependent types:

- Strongly related to Curry-Howard Isomorphism.

Dependent types

Dependent types:

- Strongly related to Curry-Howard Isomorphism.
- Introduce as a type-theory by Martin-Löf in 1971. Proposed as foundation of mathematics.

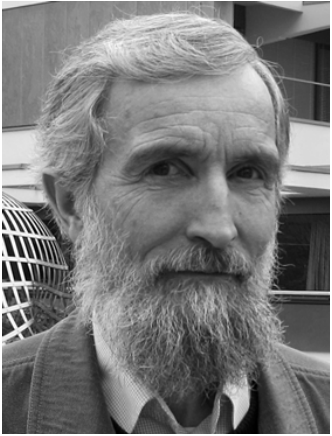


Martin-Löf

Dependent types

Dependent types:

- Strongly related to Curry-Howard Isomorphism.
- Introduce as a type-theory by Martin-Löf in 1971. Proposed as foundation of mathematics.
- Has gained popularity recently for theorem-proving with COQ,

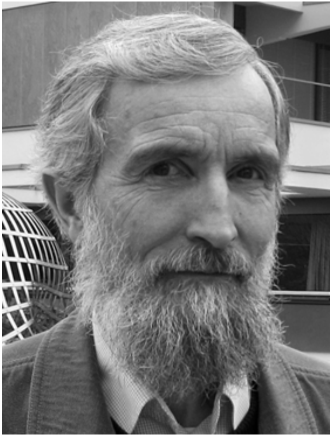


Martin-Löf

Dependent types

Dependent types:

- Strongly related to Curry-Howard Isomorphism.
- Introduce as a type-theory by Martin-Löf in 1971. Proposed as foundation of mathematics.
- Has gained popularity recently for theorem-proving with COQ,
- but also in programming: AGDA, IDRIS, ATS,...

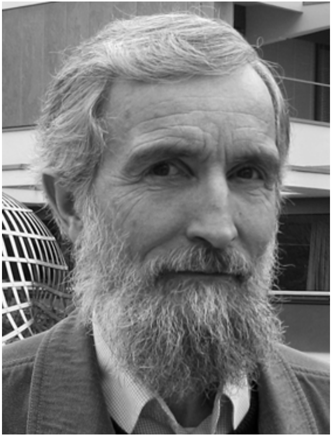


Martin-Löf

Dependent types

Dependent types:

- Strongly related to Curry-Howard Isomorphism.
- Introduce as a type-theory by Martin-Löf in 1971. Proposed as foundation of mathematics.
- Has gained popularity recently for theorem-proving with COQ,
- but also in programming: **Agda**, IDRIS, ATS,...



Martin-Löf

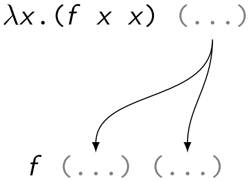
Limitations of current typecheckers

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

Efficiency issues

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.

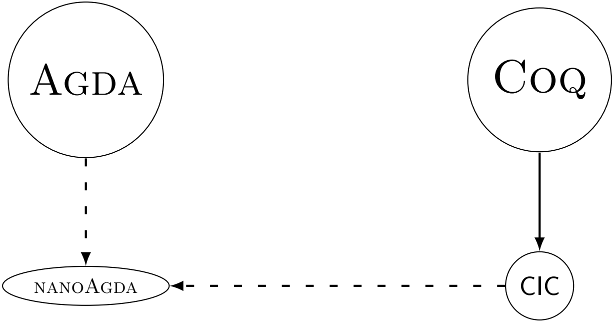
```
myFun x with f x
... | Foo = ( No knowledge that f x ≡ Foo )
... | Bar = ( No knowledge that f x ≡ Bar )
```

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).

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.

Goals

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.

Goals

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.

NANOAGDA

```
id : (a : Set) → a → a
id _ x = x
```

in AGDA

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

```
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

```
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

```
id : (a : Set) → a → a
id _ x = x
```

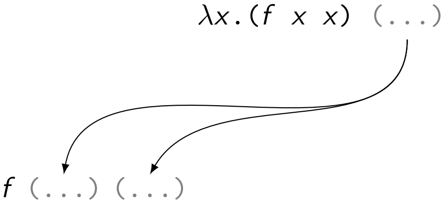
in AGDA

```
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.



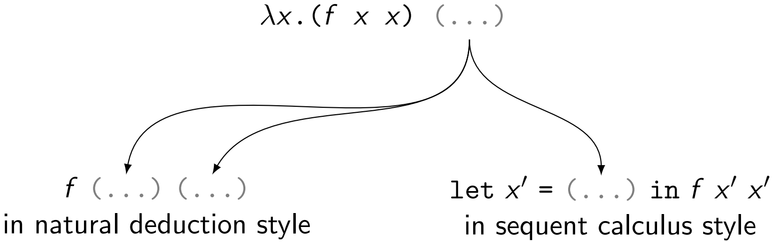
$\lambda x.(f\ \ x\ \ x)\ \ (\ldots)$

$f\ (\ldots)\ (\ldots)$

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.



$\lambda x.(f\ \ x\ \ x)\ \ (\ldots)$

$f\ (\ldots)\ (\ldots)$

in natural deduction style

$\texttt{let}\ x' = (\ldots)\ \texttt{in}\ f\ x'\ x'$

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 $\lambda x.t$ $(f\ \overline{x})$ $(x : \overline{Y}) \to T$

Presentation of the language

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

Functions	$\lambda x.t$	$(f\ \overline{x})$	$(x : \overline{Y}) \to T$
Pairs	$(\overline{x}, \overline{y})$	$x.1$	$(x : \overline{Y}) \times T$

Presentation of the language

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

Functions	$\lambda x.t$	$(f\ \overline{x})$	$(x : \overline{Y}) \to T$
Pairs	$(\overline{x}, \overline{y})$	$x.1$	$(x : \overline{Y}) \times T$
Enumerations	$\text{‘}l$	`case`	$\{\text{‘}l_1, \text{‘}l_2, \ldots\}$

Presentation of the language

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

Functions	$\lambda x.t$	$(f\ \overline{x})$	$(x : \overline{Y}) \to T$
Pairs	$(\overline{x}, \overline{y})$	$x.1$	$(x : \overline{Y}) \times T$
Enumerations	$\text{‘}l$	`case`	$\{\text{‘}l_1, \text{‘}l_2, \ldots\}$

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

Presentation of the language

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

Functions	$\lambda x.t$	$(f\ \overline{x})$	$(x : \overline{Y}) \to T$
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`

Presentation of the language

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

Functions	$\lambda x.t$	$(f\ \overline{x})$	$(x : \overline{Y}) \to T$
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.

Presentation of the language

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

Functions	$\lambda x.t$	$(f\ \overline{x})$	$(x : \overline{Y}) \to T$
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.

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
  λa → λx → x

TYPE
  (a : ⋆₁) → (x : a) → a
```

in MICROAGDA

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
  λa → λx → x

TYPE
  (a : ⋆1) → (x : a) → a
```

in microAgda

```
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

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
  λa → λx → x

TYPE
  (a : ⋆1 ) → (x : a) → a
```

in MICROAGDA

```
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

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
  λa → λx → x

TYPE
  (a : ⋆1 ) → (x : a) → a
```

in microAgda

```
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

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
  λa → λx → x

TYPE
  (a : ⋆1 ) → (x : a) → a
```

in MICROAGDA

```
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

Results

- We implemented a typechecker and evaluator for NANOAGDA.
- We introduced a new intermediate language: MICROAGDA.
- We exhibited some examples that don't typecheck in AGDA but typecheck in NANOAGDA.

Future work

- Precisely evaluate the efficiency of this new approach.
- Prove subject-reduction of nanoAgda (in Coq).
- Introduce recursion.
- Experiment with extensions of the type system (linear, colors,. . .).

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 } × Nat
Type
  ⋆0
```

in microAgda

How to encode sum types – 2^{nd} edition

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

in AGDA

How to encode sum types – 2^{nd} edition

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

in Agda

```
TERM
  Unit = { 'unit } : ⋆0 ;
  λs → ( c : { 'Foo , 'Bar } ) ×
       ( case c of {
           'Foo → s.
           'Bar → Unit.
       } )
TYPE
  ⋆0 → ⋆0
```

in microAgda

How to encode sum types – 2^{nd} edition

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

in AGDA

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

in MICROAGDA

Questions ?

Environment extension

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

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

$$\begin{aligned} \gamma + (x = d) &= \gamma \text{ with } \gamma_a \leftarrow (x = y) && \text{if } (y = d) \in \gamma_d \\ &= \gamma \text{ with } \gamma_d \leftarrow (x = d) && \text{otherwise} \end{aligned}$$

$$\gamma + (\bar{x} = c) = \gamma \text{ with } \gamma_c \leftarrow (\bar{x} = c)$$

$$\begin{aligned} \gamma + (\text{‘l} = x) &= \gamma && \text{if } (\text{‘l} = x) \in \gamma_c \\ &= \bot && \text{if } (\text{‘m} = x) \in \gamma_c \text{ for } \text{‘l} \neq \text{‘m} \\ &= \gamma \text{ with } \gamma_c \leftarrow (\text{‘l} = x) && \text{otherwise} \end{aligned}$$

Reduction rules

EvalCase

$$\frac{h(x) = (\text{‘}l_i : \_) \qquad h + (\text{‘}l_i = x) \vdash t_i \rightsquigarrow h' \vdash \overline{x}}{h \vdash \mathsf{case}\ x\ \mathsf{of}\ \{\text{‘}l_i \mapsto t_i\} \rightsquigarrow h' \vdash \overline{x}}$$

AbstractCase

$$\frac{h(x) \neq (\text{‘}l_i : \_) \qquad \forall i \quad h + (\text{‘}l_i = x) \vdash t_i \rightsquigarrow h'_i \vdash \overline{x}_i}{h \vdash \mathsf{case}\ x\ \mathsf{of}\ \{\text{‘}l_i \mapsto t_i\} \rightsquigarrow \{h_i \vdash \overline{x}_i\}}$$

EvalDestr

$$\frac{h \vdash d \rightsquigarrow h' \vdash t' \qquad h' + (x = t') \vdash t \rightsquigarrow h'' \vdash \overline{x}}{h \vdash \mathtt{let}\ x = d\ \mathtt{in}\ t \rightsquigarrow h'' \vdash \overline{x}}$$

AddDestr

$$\frac{h \vdash d \not\rightsquigarrow h' \vdash t' \qquad h + (x = d) \vdash t \rightsquigarrow h' \vdash \overline{x}}{h \vdash \mathtt{let}\ x = d\ \mathtt{in}\ t \rightsquigarrow h' \vdash \overline{x}}$$

AddConstr

$$\frac{h + (\overline{x} = c) \vdash t \rightsquigarrow h' \vdash \overline{x}}{h \vdash \mathtt{let}\ \overline{x} = c\ \mathtt{in}\ t \rightsquigarrow h' \vdash \overline{x}}$$

Concl

$$\frac{}{h \vdash \overline{x} \rightsquigarrow h \vdash \overline{x}}$$

EvalProj$_1$

$$\frac{h(y) = ((\overline{z}, \overline{w}) : \_)}{h \vdash y.1 \rightsquigarrow h \vdash \overline{z}}$$

EvalProj$_2$

$$\frac{h(y) = ((\overline{z}, \overline{w}) : \_)}{h \vdash y.2 \rightsquigarrow h \vdash \overline{w}}$$

EvalApp

$$\frac{h(y) = (\lambda w.t : \_) \qquad h \vdash t[\overline{z}/w] \rightsquigarrow h' \vdash \overline{x}}{h \vdash (y\ \overline{z}) \rightsquigarrow h' \vdash \overline{x}}$$

Equality rules

$$\gamma \vdash \texttt{let}\ x = d\ \texttt{in}\ t = t' \longrightarrow \gamma' + (x = t'') \vdash t = t'$$

$$\gamma \vdash \texttt{let}\ \bar{x} = c\ \texttt{in}\ t = t' \longrightarrow \gamma + (\bar{x} = c) \vdash t = t'$$

$$\gamma \vdash \texttt{case}\ x\ \texttt{of}\ \{`l_i \mapsto t_i\} = t \longrightarrow \forall i \quad \gamma + (x = `l_i) \vdash t_i = t$$

$$\gamma \vdash \bar{x} = \bar{y} \longrightarrow \bar{x} \equiv \bar{y} \wedge \gamma \vdash \gamma_c(\bar{x}) = \gamma_c(\bar{y})$$

$$\gamma \vdash `l = `l \longrightarrow \mathit{true}$$

$$\gamma \vdash \star_i = \star_j \longrightarrow i = j$$

$$\gamma \vdash x = y \longrightarrow x \cong y$$

$$\gamma \vdash \lambda x.t = \lambda y.t' \longrightarrow \gamma + (x = y) \vdash t = t'$$

$$\gamma \vdash (\bar{x}, \bar{y}) = (\overline{x'}, \overline{y'}) \longrightarrow \gamma \vdash \bar{x} = \overline{x'} \wedge \gamma \vdash \bar{y} = \overline{y'}$$

$$\gamma \vdash (x : \bar{y}) \to t = (x' : \overline{y'}) \to t' \longrightarrow \gamma \vdash \bar{y} = \overline{y'} \wedge \gamma + (x = x') \vdash t = t'$$

$$\gamma \vdash (x : \bar{y}) \times t = (x' : \overline{y'}) \times t' \longrightarrow \gamma \vdash \bar{y} = \overline{y'} \wedge \gamma + (x = x') \vdash t = t'$$

$$\gamma \vdash \{`l_i\} = \{`m_i\} \longrightarrow \forall i \quad `l_i = `m_i$$

$$\gamma \vdash \lambda x.t = y \longrightarrow \gamma + (\bar{x} = x) + (z = y\,\bar{x}) \vdash t = z$$

$$\gamma \vdash (\bar{x}, \overline{x'}) = y \longrightarrow \gamma + (z = y.1) \vdash \bar{x} = z \wedge \gamma + (z = y.2) \vdash \overline{x'} = z$$

Typing rules

$$\dfrac{\forall\, \mathsf{i} \quad \gamma + (\text{‘}\mathsf{l_i} = \mathsf{x}) \vdash \mathsf{t_i} \Lleftarrow \mathsf{T} \qquad \Gamma(x) = \{\text{‘}\mathsf{l_i}\}}{\gamma \vdash \mathsf{case}\ \mathsf{x}\ \mathsf{of}\ \{\text{‘}\mathsf{l_i} \mapsto \mathsf{t_i}\} \Lleftarrow \mathsf{T}}\ \textsc{Case} \qquad \dfrac{\gamma + (\bar{\mathsf{x}} = \mathsf{c}) \vdash \mathsf{t} \Lleftarrow \mathsf{T}}{\gamma \vdash \mathtt{let}\ \bar{\mathsf{x}} = \mathsf{c}\ \mathtt{in}\ \mathsf{t} \Lleftarrow \mathsf{T}}\ \textsc{Constr}$$

$$\dfrac{\gamma_c(\bar{\mathsf{x}}) = \mathsf{c} \qquad \gamma \vdash \mathsf{c} \Lleftarrow \mathsf{T}}{\gamma \vdash \bar{\mathsf{x}} \Lleftarrow \mathsf{T}}\ \textsc{Concl} \qquad \dfrac{\gamma \vdash \mathsf{d} \rightrightarrows \mathsf{T}' \qquad \gamma \vdash \mathsf{d} \rightsquigarrow \gamma' \vdash \mathsf{t}' \qquad \gamma' + (\mathsf{x} = \mathsf{t}') + (\mathsf{x} : \mathsf{T}') \vdash \mathsf{t} \Lleftarrow \mathsf{T}}{\gamma \vdash \mathtt{let}\ \mathsf{x} = \mathsf{d}\ \mathtt{in}\ \mathsf{t} \Lleftarrow \mathsf{T}}\ \textsc{Destr}$$

$$\dfrac{\gamma \vdash \mathsf{T} \rightsquigarrow \{\gamma'_{\mathsf{i}} \vdash \bar{\mathsf{X}}_{\mathsf{i}}\} \qquad \forall \mathsf{i} \quad \gamma'_{\mathsf{i}} \vdash \bar{\mathsf{X}}_{\mathsf{i}} \Lleftarrow \bar{\mathsf{X}}}{\gamma \vdash \mathsf{t} \Lleftarrow \mathsf{T}}\ \textsc{Eval}$$

Figure : Typechecking a term: $\gamma \vdash \mathsf{t} \Lleftarrow \mathsf{T}$

Typing rules

$$\textsc{App}\quad \frac{\Gamma(y) = (z : \overline{X}) \to T \qquad \gamma \vdash \bar{z} \leftleftarrows \overline{X}}{\gamma \vdash y\,\bar{z} \rightrightarrows T}$$

$$\textsc{Proj}_1\quad \frac{\Gamma(y) = (z : \overline{X}) \times T}{\gamma \vdash y.1 \rightrightarrows \overline{X}}$$

$$\textsc{Proj}_2\quad \frac{\Gamma(y) = (z : \overline{X}) \times T}{\gamma \vdash y.2 \rightrightarrows T}$$

$$\textsc{Cut}\quad \frac{\gamma \vdash \bar{x} \leftleftarrows \overline{X}}{\gamma \vdash (\bar{x} : \overline{X}) \rightrightarrows \overline{X}}$$

Figure : Inferring the type of a destruction: $\gamma \vdash d \rightrightarrows T$.

$$\textsc{TyDestr}\quad \frac{\gamma + (x = d) \vdash c \leftleftarrows T}{\gamma \vdash c \leftleftarrows \mathtt{let}\ x = d\ \mathtt{in}\ T}$$

$$\textsc{TyConstr}\quad \frac{\gamma + (\bar{x} = c) \vdash c \leftleftarrows T}{\gamma \vdash c \leftleftarrows \mathtt{let}\ \bar{x} = c\ \mathtt{in}\ T}$$

$$\textsc{TyCase}\quad \frac{\forall\, i \quad \gamma + ({}`l_i = x) \vdash c \leftleftarrows T_i \qquad \gamma \vdash x \rightrightarrows \{{}`l_i\}}{\gamma \vdash c \leftleftarrows \mathsf{case}\ x\ \mathsf{of}\ \{{}`l_i \mapsto T_i\}}$$

$$\textsc{TyConcl}\quad \frac{\gamma_c(\bar{x}) = C \qquad \gamma \vdash c \leftleftarrows C}{\gamma \vdash c \leftleftarrows \bar{x}}$$

$$\textsc{Infer}\quad \frac{\Gamma(x) = \overline{X} \qquad \gamma \vdash \overline{X} = T}{\gamma \vdash x \leftleftarrows T}$$

Figure : Typechecking a construction against a term: $\gamma \vdash c \leftleftarrows T$.

Typing rules

Pair
$$\frac{\gamma \vdash \bar{y} \Lleftarrow \overline{X} \qquad \gamma + (x = (\bar{y} : \overline{X})) \vdash \bar{z} \Lleftarrow T}{\gamma \vdash (\bar{y}, \bar{z}) \Lleftarrow (x : \overline{X}) \times T}$$

Lambda
$$\frac{\gamma + (y : \overline{X}) + (x = y) \vdash t \Lleftarrow T}{\gamma \vdash \lambda y.t \Lleftarrow (x : \overline{X}) \to T}$$

Label
$$\frac{`l \in \{`l_i\}}{\gamma \vdash `l \Lleftarrow \{`l_i\}}$$

Sigma
$$\frac{\gamma \vdash \bar{y} \Lleftarrow \star_i \qquad \gamma + (x : \bar{y}) \vdash t \Lleftarrow \star_i}{\gamma \vdash (x : \bar{y}) \times t \Lleftarrow \star_i}$$

Pi
$$\frac{\gamma \vdash \bar{y} \Lleftarrow \star_i \qquad \gamma + (x : \bar{y}) \vdash t \Lleftarrow \star_i}{\gamma \vdash (x : \bar{y}) \to t \Lleftarrow \star_i}$$

Fin
$$\frac{}{\gamma \vdash \{`l_i\} \Lleftarrow \star_i}$$

Universe
$$\frac{i < j}{\gamma \vdash \star_i \Lleftarrow \star_j}$$

Figure : Typechecking a construction against a construction: $\gamma \vdash c \Lleftarrow C$.