HOAS

Randy Pollack

Version of November 2, 2011

Outline

Outline

HOAS by Example: Syntax of FOL

Signature to encode syntax of FOL:

```
i     : type.                % base type of Individuals
zero  : i.                   % constants in signature ...
suc   : i -> i.
plus  : i -> i -> i.

o       : type.              % base type of Propositions
imp     : o -> o -> o.       % constants in signature ...
and     : o -> o -> o.
or      : o -> o -> o.
forall  : (i -> o) -> o.
eq      : i -> i -> o.
```

Simply typed framework handles binding (consider `forall`).

HOAS by Example: Some Natural Deduction Rules

Signature to encode judgements using dependent types.

$$\frac{\begin{matrix}[A]\\ B\end{matrix}}{A \implies B} \qquad \frac{(A \implies B) \quad A}{B} \qquad \frac{A}{A \vee B} \qquad \frac{A \vee B \quad \begin{matrix}[A]\\ C\end{matrix} \quad \begin{matrix}[B]\\ C\end{matrix}}{C}$$

```
prf  : o -> type.

impi : {A, B : o} (prf A -> prf B) -> prf (A imp B).
impe : prf (A imp B) -> prf A -> prf B.
oril : prf A -> prf (A or B).
ore  : prf (A or B) ->
        (prf A -> prf C) -> (prf B -> prf C) ->
       prf C.
```

Dependent types to represent judgements.
Framework handles hypothetical and schematic judgements.

HOAS by Example: Some more rules

$$\frac{A(x)}{\forall x.A}\ (x \text{ fresh}) \qquad \frac{\forall x.A}{A(t)} \qquad \frac{(A \Longrightarrow B) \qquad \begin{array}{c}\left[\begin{array}{c}[A]\\ B\end{array}\right]\\ C\end{array}}{C}\ (\mathrm{SH})$$

```
foralli : {A : i -> o}
                ({x:i} prf (A x)) -> prf (forall A).
foralle : prf (forall A) -> {t:i} prf (A t).
-- A Schroeder-Heister version of imp elim
SH_impe : prf (A imp B) ->
             ((prf A -> prf B) -> prf C) ->
             ---------------------------
             prf C
```

Framework handles freshness and substitution.

The SH rule is clear in the framework (it is third order).

A look at actual Twelf: Pure lambda terms

```
tm : type.    %name tm M x.         % type of lambda terms
app : tm -> tm -> tm.
lam : (tm -> tm) -> tm.

%abbrev @ = app.                    % infix application
%infix left 10 @.

% relation of beta reduction on lambda terms
step : tm -> tm -> type.
s-beta : step ((lam F) @ M) (F M).   % contraction
s-1 : step (M1 @ M2) (M1'@ M2)       % congruence ...
       <- step M1 M1'.
s-2 : step (M1 @ M2) (M1 @ M2')
       <- step M2 M2'.
```

In `s-beta`, `(F M)` is application at framework level.

A look at actual Twelf: Simple type assignment

```
tp : type.          %name tp A B.    % simple types
o    : tp.                           % base type
arr : tp -> tp -> tp.

%abbrev => = arr.                    % infix arrow
%infix right 10 =>.

tpng : tm -> tp -> type.
tapp : tpng (M1 @ M2) B
          <- tpng M1 (A => B)
          <- tpng M2 A.
tlam : tpng (lam F) (A => B)
          <- ({x} tpng x A -> tpng (F x) B).
```

No typing contexts; instead natural deduction style discharge.

Frameworks for relations over syntax

- Syntax of expressions (λ -terms, FOL terms and formulas) can be represented in HOAS style using simple types.
- Used dependent types (but predicative, logically weak) to represent relations over syntax (FOL provability, typing of λ -terms).
- There are at least two other approaches to representing relations over syntax:
  - The *two layer approach* of a logic over a simply-typed framework (Abella, Hybrid, . . .)
  - Simply typed higher order logic (Isabelle, [Felty 1991]).
- Can we reason by induction over structure of expressions or derivations?
  - That is another long story.

Outline

Rules of Inference, Formal Systems

Let $\mathcal{J}$ be a set (call these judgements).

A rule of inference is a partial function from $\mathcal{J}$ -lists into $\mathcal{J}$.

If $\mathcal{R}$ is a rule and $A_1, \ldots, A_n \in \mathcal{J}$, we say

$$\frac{A_1, \ldots, A_n}{\mathcal{R}\,[A_1, \ldots, A_n]}$$

is an instance of $\mathcal{R}$, with premises $A_1, \ldots, A_n$ and conclusion $\mathcal{R}\,[A_1, \ldots, A_n]$.

An axiom is a rule all of whose instances have an empty list of premises.

A formal system, $\mathcal{F}$, is a set of rules.

Derivations, Theorems

Let $\mathcal{F}$ be a formal system with language of judgements $\mathcal{J}$.

Derivation is a relation between $\mathcal{J}$-lists and $\mathcal{J}$, defined inductively

$$\frac{}{js \vdash r(js)}\ r \in \mathcal{F} \qquad \frac{js \vdash j \quad jsl@(j::jsr) \vdash c}{jsl@(js@jsr) \vdash c}$$

$j \in \mathcal{J}$ is a theorem iff $\vdash j$. Also say that a theorem is provable, and a derivation of form $\vdash j$ is a proof.

- **No schematic derivations** in this model.
  - By talking about instances we avoid talk of “variables” and “substitutions”.
- We will need rules for weakening and permutaion of premises.

Remarks on Formal Systems

- **Relational rules** Some presentations generalize the notion "rule" to decidable relations between $\mathcal{J}$ -lists and $\mathcal{J}$.
  - Closer to informal notion; e.g. side conditions may refer to the conclusion as well as to the premises.
  - For relational rules, the definition of derivation is modified

$$\frac{}{js \vdash c} \quad r \in \mathcal{F},\; r(js, c) \qquad\qquad \cdots$$

- **Number of premises** Some presentations restrict each rule to a fixed number of premises.
  - In reasonably strong meta systems lists of premises can still be represented.

Derivable Rules

Let $\mathcal{F}$ be a formal system over judgements $\mathcal{J}$.

A rule, $\mathcal{R}$, is derivable in $\mathcal{F}$ if, for every instance of $\mathcal{R}$, there is a derivation in $\mathcal{F}$ of its conclusion from its premises.

$$\frac{\begin{array}{c}\text{premises of } \mathcal{R} \\ \vdots\end{array}}{\text{conclusion of } \mathcal{R}}$$

- Informally, a rule is derivable in $\mathcal{F}$ if it can be proved by gluing rules of in $\mathcal{F}$ together.
- If $\mathcal{R}$ is derivable in $\mathcal{F}$, then $\mathcal{R}$ is derivable in any extension of $\mathcal{F}$ by new axioms and rules.
- Adding derivable rules to a formal system does not change its set of theorems.

A Derivable Rule: forgetting an assumption

$$\frac{G \vdash b}{G \vdash a \to b}$$

A derivation of its conclusion from its premise:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{G, b, a \vdash b}\,\text{AXIOM}}{G, b \vdash a \to b}\,\text{INTRO}}{G \vdash b \to (a \to b)}\,\text{INTRO} \qquad G \vdash b}{G \vdash a \to b}\,\text{ELIM}$$

Admissible Rules

A rule, $\mathcal{R}$, is admissible in $\mathcal{F}$ if, for every instance of $\mathcal{R}$, whenever its premises are provable (from no assumptions), then so is its conclusion.

- Every derivable rule is admissible.
- $\mathcal{R}$ is admissible in $\mathcal{F}$ **iff** adding $\mathcal{R}$ to $\mathcal{F}$ does not change $\mathcal{F}$'s set of theorems.
- Admissibility in $\mathcal{F}$ is not necessarily preserved by extending $\mathcal{F}$ with new axioms or rules.
- For axioms, admissible and derivable coincide.

An Admissible Rule: weakening

$$\frac{K \vdash a}{G \vdash a} \quad (K \subseteq G)$$

Prove weakening is admissible by induction on the derivation of $K \vdash a$:

$$\dfrac{\dfrac{\dfrac{}{K, b, a \vdash b}\ \text{AXIOM}}{K, b \vdash a{\rightarrow}b}\ \text{INTRO}}{K \vdash b{\rightarrow}(a{\rightarrow}b)}\ \text{INTRO} \quad \Longrightarrow \quad \dfrac{\dfrac{\dfrac{}{G, b, a \vdash b}\ \text{AXIOM}}{G, b \vdash a{\rightarrow}b}\ \text{INTRO}}{G \vdash b{\rightarrow}(a{\rightarrow}b)}\ \text{INTRO}$$

- We must eliminate the particular premise to construct a derivation of the particular conclusion.
- Weakening cannot be proved by gluing rules together.

Structural Completeness

- By definition every derivable rule is admissible.
- In some formal systems (*structurally complete*) every admissible rule is derivable.
- Classical proposotional logic (CPC) (Hilbert style rules, including a constant for False), is structurally complete.
  - Let $\dfrac{A}{B}$ be a rule that is not derivable.
  - Since CPC is complete for truth tables, there is an assignment, σ , of truth values to propositional constants such that $\sigma(A)$ is True and $\sigma(B)$ is False.
  - Since CPC is sound for truth tables, $\sigma(A)$ is a theorem and $\sigma(B)$ is not a theorem.
  - Thus $\dfrac{A}{B}$ is not admissible. $\square$

Logical strength

- The Edinburgh Logcal Framework (ELF, used in Twelf) types no more functions than STLC.
  - Being dependently typed, it gives more informative types.
  - We can erase the dependency in an ELF derivation to get simple typing [Luo, Paulin-Mohring].
- Normalization of ELF (hence consistency) can be proven in Peano Arithmetic.
- The representation of an object system describes its syntax, not its logical strength.
  - In most representations, exactly the derivable rules of a judgement are typed in the framework.
  - To prove the admissible rules (requires induction over structure) is a whole different story.
- E.g. we can represent ZFC, and check derivations of ZFC judgements in ELF without committing to consistency of ZFC.

Expressiveness

- ELF can represent many formal systems.
  - “Natural deduction” style sytems are most naturally represented.
  - Many other styles can be “coded”: explicit contexts, explicit mention of variables, . . .
- But some formal systems cannot be represented in ELF
  - A standard example is any linear system, since ELF has a multiplicative context.
    - A new *Linear Logical Framework LLF* must be defined.
  - Unary substitution comes for free, but what about simultaneous substitution . . .
  - . . . or binding a set of variables at once?

Outline

A simply typed framework: Signatures and Contexts

- Atomic types of the framework: A set of atomic types, $A, B, \ldots$. These are the types being represented.
  - To represent pure λ terms, only base type is `tm`.
  - To represent FOL, base types are `i` and `o`.
- Simple types over the set of atomic types, $S, T, \ldots$.
- A signature Σ of constants for the representation.
  - To represent pure λ terms, the signature is:

    `app:tm->(tm->tm), lam:(tm->tm)->tm`.
  - To represent FOL,

    `zero:i, suc:i->i, ..., forall:(i->o)->o ...`
  - `app, lam, zero, forall,...` are a class of framework constant names.
- A context, Γ, as usual, binding atomic types to *framework variable* names.

A simply typed framework: Typing rules

$$\frac{\Sigma \text{ valid} \quad \Gamma \text{ valid} \quad C{:}T \in \Sigma}{\Sigma; \Gamma \vdash C : T} \qquad \frac{\Sigma \text{ valid} \quad \Gamma \text{ valid} \quad X{:}A \in \Gamma}{\Sigma; \Gamma \vdash X : A}$$

$$\frac{\Sigma; \Gamma \vdash m : A \to B \qquad \Sigma; \Gamma \vdash n : A}{\Sigma; \Gamma \vdash m \cdot n : B}$$

$$\frac{\Sigma; (\Gamma, Y{:}A) \vdash m : B}{\Sigma; \Gamma \vdash [Y]m : A \to B}$$

- Contexts only contain atomic types.
  - We only need to abstract over objects being represented.

Representation of object systems

- We represent pure λ terms by
  - specifying `tm` as the only atomic type,
  - giving the signature

$$\Sigma \triangleq \texttt{app:tm->tm->tm, lam:(tm->tm)->tm}$$

- There is a representation function of informal lambda terms into framework:

$$\ulcorner x \urcorner \triangleq x$$
$$\ulcorner m\,n \urcorner \triangleq \texttt{app}\cdot\ulcorner m \urcorner\cdot\ulcorner n \urcorner$$
$$\ulcorner \lambda x.m \urcorner \triangleq \texttt{lam}\cdot([x]\ulcorner m \urcorner)$$

  $-\cdot-$ and $[-]-$ are the framework level constructors.
- $\ulcorner - \urcorner$ should be an adequate representation:
  - A substitution preserving isomorphism between informal λ terms and canonical forms in the framework with signature Σ.

Outline

Adequate representation

- Let Σ be the signature of λ terms, and m be an (object level) λ term,
- Let $\Gamma = \texttt{x1:tm, ..., xn:tm}$
  - where `x1, ..., xn` includes all the free variables of m.
- $\ulcorner - \urcorner$ **is well typed:** $\Sigma; \Gamma \vdash \ulcorner m \urcorner : \texttt{tm}$
- $\ulcorner - \urcorner$ **respects substitution:** $\ulcorner [m/X]n \urcorner = [\ulcorner m \urcorner / X] \ulcorner n \urcorner$.
  - Substitution comes for free in this style of framework.
- $\ulcorner - \urcorner$ **is injective.**
- $\ulcorner - \urcorner$ **is surjective?** Here we must be careful.
  - Surjection on canonical forms of the framework.

Canonical Forms: $\alpha\beta$-equivalence

- We want a compositional isomorphism between object language expressions and canonical forms of the LF.
  - We have implicitly been speaking up-to α-equivalence.
- In order for the framework to handle substitution, we identify meta-terms up to β.
- The framework definition of β-contraction:

```
step : tm -> tm -> type.
s-beta : step ((lam F) @ M) (F M).   % contraction
```

- `step` $((\texttt{lam}\cdot([x]\ulcorner m\urcorner)\ @\ \ulcorner n\urcorner)\quad(([x]\ulcorner m\urcorner)\cdot\ulcorner n\urcorner)$
- In object system $(\lambda x.m)\, n \;\rightarrow^{\beta}\; [n/x]m$.
- So the framework must identify $(([x]\ulcorner m\urcorner)\cdot\ulcorner n\urcorner)$ with $[\ulcorner n\urcorner/x]\ulcorner m\urcorner$, i.e. up to β-equivalence.
- Canonical forms are β-normal.

Canonical Forms: η-equivalence

- β is not enough, we need η-equivalence in the notion of *canonical*.
  - The two β-normal, well-typed framework terms

    `suc` `[x:i](suc x)`

    represent the same object expression (a function symbol of the language of FOL).
- η-equivalence identifies these two terms

$$\lambda x.(M\,x) \quad \overset{\eta}{\sim} \quad M \qquad (x \notin M)$$

- **Problem:** η reduction is not Church–Rosser on raw terms of ELF:

$$[x:A]C \quad \overset{\beta}{\leftarrow} \quad [x:A](([y:B]C)x) \quad \overset{\eta}{\rightarrow} \quad [y:B]C \quad x \notin B, C$$

- Equality becomes mutually recursive with the typing relation.
- Thus, the meta theory of $\beta\eta$-equivalence for the ELF language (with type tags) is complicated.

Canonical Forms: η-equivalence

- Further, many extensions (singleton types, dependent pairs, linear types, ...) interact badly with η.
- Consider a unit type $* \in 1$. η equivalence is

$$m \stackrel{\eta}{\sim} * \qquad (\text{if } m \in 1)$$

With η-reduction (left to right), confluence is lost, because for any variable $f : S \to 1$ we have two normal forms:

$$\lambda x.* \stackrel{\eta}{\leftarrow} \lambda x.(f\,x) \stackrel{\eta}{\to} f$$

But η-expansion preserves confluence.

- η-expansion (left to right) must be controlled by typing

$$\begin{array}{lll} f & \stackrel{\eta}{\leftarrow} & \lambda x.f\,x \qquad \text{if } f : S \to T \text{ is a variable} \\ m & \stackrel{\eta}{\leftarrow} & * \qquad\qquad \text{if } m : 1 \end{array}$$

With this we have $\beta\eta$ strong normalization and confluence.

Outline

Canonical LF

- Accepts exactly the canonical judgements accepted by ELF.
- Developed by Watkins, Pfenning, and co-workers [WCPW02, HL06].
- This work precedes both Adams' TF and Plotkin's DMBEL.
- Only β-normal terms are syntactically formed.
- Only canonical terms (η-expanded, β-normal) are well-typed.
  - Equality is syntactic identity of expressions.
  - A notion of hereditary substitution is needed.
- Bi-directional typechecking: syntax directed and decidable.

Canonical LF: Syntax

identifiers	a (type constants), c (term constants), x (term variables)	
kinds	K	$::= \mathsf{type} \mid \Pi x{:}A.K$
types	A, B	$::= P \mid \Pi x{:}A.B$
atomic types	P	$::= a \mid P\ M$
terms	M	$::= R \mid \lambda x.M$
atomic terms	R	$::= c \mid x \mid R\ M$
contexts	Γ, Δ	$::= \bullet \mid \Gamma, x{:}A$
signatures	Σ	$::= \bullet \mid \Sigma, a{:}K \mid \Sigma, c{:}A$

Canonical LF: Judgement Forms

Validity of signatures and contexts:

$$\vdash \Sigma \text{ sig} \qquad\qquad \vdash_\Sigma \Gamma \text{ cxt}$$

Checking correctness of kinds and types:

$$\Gamma \vdash_\Sigma K \text{ kind} \qquad\qquad \Gamma \vdash_\Sigma A \text{ type}$$

Inferring kinds of atomic types, and the types of atomic terms:

$$\Gamma \vdash_\Sigma P \Rightarrow K \qquad\qquad \Gamma \vdash_\Sigma R \Rightarrow A$$

Checking the types of terms:

$$\Gamma \vdash_\Sigma M \Leftarrow A$$

- *Bi-directional* typechecking, completely syntax directed.
- Cannot infer the type of $\lambda x.M$: no type annotation on x.

Canonical LF: Rules

Signatures

$$\frac{}{\vdash \bullet \ \mathsf{sig}} \qquad \frac{\vdash \Sigma \ \mathsf{sig} \qquad \vdash_\Sigma K \ \ \mathsf{kind}}{\vdash \Sigma, a{:}K \ \mathsf{sig}} \qquad \frac{\vdash \Sigma \ \mathsf{sig} \qquad \vdash_\Sigma A \ \ \mathsf{type}}{\vdash \Sigma, c{:}A \ \mathsf{sig}}$$

Contexts

$$\frac{\vdash \Sigma \ \mathsf{sig}}{\vdash_\Sigma \bullet \ \mathsf{cxt}} \qquad \frac{\vdash_\Sigma \Gamma \ \mathsf{cxt} \qquad \Gamma \vdash_\Sigma A \ \ \mathsf{type}}{\vdash_\Sigma \Gamma, x{:}A \ \mathsf{cxt}}$$

Canonical LF: Rules

Kinds

$$\frac{\vdash_\Sigma \Gamma \ \mathsf{cxt}}{\Gamma \vdash_\Sigma \mathsf{type} \ \ \mathsf{kind}} \qquad \frac{\Gamma, x{:}A \vdash_\Sigma K \ \ \mathsf{kind}}{\Gamma \vdash_\Sigma \Pi x{:}A.K \ \ \mathsf{kind}}$$

Types

$$\frac{\Gamma, x{:}A \vdash_\Sigma B \ \ \mathsf{type}}{\Gamma \vdash_\Sigma \Pi x{:}A.B \ \ \mathsf{type}} \qquad \frac{\Gamma \vdash_\Sigma P \Rightarrow \mathsf{type}}{\Gamma \vdash_\Sigma P \ \ \mathsf{type}}$$

Atomic Types

$$\frac{\vdash_\Sigma \Gamma \ \mathsf{cxt}}{\Gamma \vdash_\Sigma a \Rightarrow K} \ (a{:}K \in \Sigma) \qquad \frac{\Gamma \vdash_\Sigma P \Rightarrow \Pi x{:}A.K \qquad \Gamma \vdash_\Sigma M \Leftarrow A}{\Gamma \vdash_\Sigma P\, M \Rightarrow [M^{(A)^-}/x]K}$$

Canonical LF: Rules

Terms

$$\frac{\Gamma, x{:}A \vdash_\Sigma M \Leftarrow B}{\Gamma \vdash_\Sigma \lambda x.M \Leftarrow \Pi x{:}A.B} \qquad \frac{\Gamma \vdash_\Sigma R \Rightarrow P}{\Gamma \vdash_\Sigma R \Leftarrow P}\ (*)$$

Atomic Terms

$$\frac{\vdash_\Sigma \Gamma\ \mathrm{cxt}}{\Gamma \vdash_\Sigma x \Rightarrow A}\ (x{:}A \in \Gamma) \qquad \frac{\vdash_\Sigma \Gamma\ \mathrm{cxt}}{\Gamma \vdash_\Sigma c \Rightarrow A}\ (c{:}A \in \Sigma)$$

$$\frac{\Gamma \vdash_\Sigma R \Rightarrow \Pi x{:}A.B \qquad \Gamma \vdash_\Sigma M \Leftarrow A}{\Gamma \vdash_\Sigma R\ M \Rightarrow [M^{(A)^-}/x]B}$$

$(*)$ This restriction to atomic types, P, makes judgements canonical.

Hereditary Substitution: Pure λ-terms

Let L, M, N be lambda terms.
Let A, B, C be simple types

$$
\begin{array}{lcll}
[M^A/x]x & = & M^A & \\
[M^A/x]y & = & y & \text{if } x \neq y \\
[M^A/x](\lambda y.N) & = & \lambda y.[M^A/x]N & y \text{ fresh} \\
[M^A/x](L\,N) & = & ([\hat{N}^B/y]M')^C & \text{if } \hat{L} = (\lambda y.M')^{B \to C} \\
 & = & \hat{L}\,\hat{N} & \text{otherwise}
\end{array}
$$

$$\text{where } \hat{L} = [M^A/x]L \text{ and } \hat{N} = [M^A/x]N$$

- If M and N have no β-redexes, then $[M/x]N$
  - is β-equal to the usual substitution $[M/x]N$
  - contains no β-redexes
- However $[M/x]N$ may not terminate.
- We use types to control how deep the substitution goes.

Hereditary Substitution: Simply typed terms [Abe06]

Let L, M, N be lambda terms.
Let A, B, C be simple types

$$
\begin{array}{lcll}
[M^A/x]x & = & M^A & \\
[M^A/x]y & = & y & \text{if } x \neq y \\
[M^A/x](\lambda y.N) & = & \lambda y.[M^A/x]N & y \text{ fresh} \\
[M^A/x](L\ N) & = & ([\hat{N}^B/y]M')^C & \text{if } \hat{L} = (\lambda y.M')^{B \to C} \\
 & = & \hat{L}\ \hat{N} & \text{otherwise}
\end{array}
$$

$$\text{where } \hat{L} = [M^A/x]L \text{ and } \hat{N} = [M^A/x]N$$

- If M and N have no β-redexes, then $[M/x]N$
  - is β-equal to the usual substitution $[M/x]N$
  - contains no β-redexes if the type annotation is correct
- Always terminates: newly created substitutions operate on subterms, or have syntactically smaller types.

Hereditary Substitution: Canonical LF

- LF types exactly the same lambda-terms as simple types …
- …just throw away the dependency.

$$
\begin{array}{lcl}
(a)^- & = & a \\
(P\ M)^- & = & (P)^- \\
(\Pi x{:}A.B)^- & = & (A)^- \to (B)^-
\end{array}
$$

  Only the shape of the simple type matters: can take $(a)^- = \bullet$.
- One more problem: canonical terms cannot contain redexes.
  - Instead of returning a term with a redex (when the given type is incorrect), the canonical hereditary substitution algorithm must return failure.
- Finally, $[M^{(A)^-}/x]\_$ is a congruence, generated by the replacement of term variables in terms, as explained above.

Judgements are Canonical by Stratified Syntax

We want to know that no two derivable judgements differ only by $\beta\eta$.

- Let $\Sigma = a\mathord{:}\mathrm{type}$.
  - $(a{\to}a)$ is a non-atomic type, syntactic class A, not class P.

Direction $\Rightarrow$:

- $f{:}(a{\to}a) \vdash_\Sigma f \Rightarrow (a{\to}a)$ is derivable,
- $f{:}(a{\to}a) \vdash_\Sigma \lambda x.f\ x \Rightarrow (a{\to}a)$ is not derivable:
  - $\lambda x.f\ x$ is of class M, not of class R,
  - there are no rules of shape $\Gamma \vdash_\Sigma M \Rightarrow A$.

Judgements are Canonical by Stratified Syntax

Direction $\Leftarrow$:

- $f{:}(a{\to}a) \vdash_\Sigma f \Leftarrow (a{\to}a)$ is not derivable.
  - f is not of shape $\lambda x.M$ and $a{\to}a$ is not of shape P, so no rule applies.
- $f{:}(a{\to}a) \vdash_\Sigma \lambda x.f\,x \Leftarrow (a{\to}a)$ is derivable:
  Let $\Gamma = f{:}(a{\to}a), x{:}a$.

$$\dfrac{\dfrac{f{:}(a{\to}a) \vdash_\Sigma a \Rightarrow \text{type}}{f{:}(a{\to}a) \vdash_\Sigma a \Leftarrow \text{type}} \quad \dfrac{\dfrac{\Gamma \vdash_\Sigma f \Rightarrow (a{\to}a) \quad \dfrac{\Gamma \vdash_\Sigma x \Rightarrow a}{\Gamma \vdash_\Sigma x \Leftarrow a}(*) \quad [x/\_]a = a}{\Gamma \vdash_\Sigma f\,x \Rightarrow a}}{\Gamma \vdash_\Sigma f\,x \Leftarrow a}(*)}{f{:}(a{\to}a) \vdash_\Sigma \lambda x.f\,x \Leftarrow (a{\to}a)}$$

$(*)$ At atomic type.

Outline

References

[Abe06] Andreas Abel. Implementing a normalizer using sized heterogeneous types. In *Workshop on Mathematically Structured Functional Programming, MSFP 2006*.

[Ada08] Robin Adams. Lambda-free logical frameworks. arXiv:0804.1879v2 [cs.LO].

[Gog05] Healfdene Goguen. A syntactic approach to eta equality in type theory. In *POPL 2005*.

[HL06] Robert Harper and Daniel R. Licata. Mechanizing Metatheory in a Logical Framework. Submitted for publication. Available from `http://www.cs.cmu.edu/~drl/`, Oct. 2006.

[HP05] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. on Computational Logic*, 6(1), 2005.

[Luo03] Zhaohui Luo. PAL $^{+}$: a lambda-free logical framework. *Journal of Functional Programming*, 13(2):317–338, 2003.

[Sal] Anne Salvesen. The Church-Rosser theorem for LF with beta/eta reduction. Talk given at First Workshop on Logical Frameworks, Antibes, 1990.

[WCPW02] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[CNV11] James Cheney, Michael Norrish and René Vestergaard. Formalizing adequacy: a case study for higher-order syntax. Journal of Automated Reasoning, 2011.