Workshop MERLIN

Siena, 18 June 2001

# Developing (Meta)Theory of $\lambda$-calculus in the Theory of Contexts

Marino Miculan

Università di Udine, Italy

miculan@dimi.uniud.it

# A common scenario

- represent formally (*encode*) syntax and semantics of an object language (e.g., $\lambda$-, $\pi$-calculus) in some logical framework for doing formal (meta)reasoning.
- derive some results interaction in a goal-directed manner, using tactics in some general-purpose theorem prover/proof assistant

Problem: how to render binding operators (e.g, $\lambda$, $\nu$) efficiently?

In interactive development,

efficiently $\cong$ "formal proofs should look like on paper"

Many approaches, with pros and cons: *de Bruijn indexes, first-order abstract syntax, higher-order abstract syntax* ... [HHP87,Hue94,DFH95,GM96,MM01,...].

They have to be tested on *real* case studies, in *real* proof assistants.

# In this talk

We focus on

call-by-name $\lambda$-calculus in type-theory based proof assistants (viz., Coq), using (weak) HOAS and the Theory of Contexts.

Why $\lambda$-calculus?

- complementary to $\pi$-calculus (higher-order binders, terms-for-variables substitution, ...) which has been already done [HMS01]

- well-known (meta)theory. Too well, maybe.

- customary benchmark for formal treatments of binders $\Rightarrow$ allows for comparison with other approaches ([Momigliano et al. 2001] for a survey)

**Claim:** the formal, fully detailed development of the theory of $\lambda_{\mathsf{cbn}}$ in the Theory of Contexts introduces a small, sustainable overhead with respect to the proofs "on the paper".

# Outline of the talk

- Definition of $\lambda_{\text{cbn}}$ “on the paper”
- Encoding of syntax and semantics of $\lambda_{\text{cbn}}$ in HOAS
- Some formally proved results.
- Extending the language: type systems. More results.
- Discussion
- Related work
- Future work

# A typical definition of $\lambda_{\text{cbn}}$ in 1 slide

**Syntax** The set $\Lambda$ is defined by $\Lambda$ : $\quad M, N ::= x \mid (MN) \mid \lambda x.M$
where $x, y, z, \ldots$ range over an infinite set of variables. Terms are taken up-to $\alpha$-equivalence. We denote by $M[N/x]$ the capture-avoiding substitution of $N$ for $x$ in $M$. *Free variables* $(FV)$ are defined as usual. For $X$ a finite set of variables, we define $\Lambda_X \triangleq \{M \in \Lambda \mid FV(M) \subseteq X\}$.

*Contexts*, i.e. terms with holes, are denoted by $M(\cdot)$. A (closed) term is said to be a *value* if it is not an application.

**Small-step semantics** (or *reduction*) is the smallest relation $M \longrightarrow N$ defined by

$$\frac{}{(\lambda x.M)\ N \longrightarrow M[N/x]} \qquad \frac{M \longrightarrow M'}{(M\ N) \longrightarrow (M'\ N)}$$

We denote by $\longrightarrow^*$ the reflexive and transitive closure of $\longrightarrow$.

**Big-step semantics** (or *evaluation*) is the smallest relation $M \Downarrow N$ defined by

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{M \Downarrow \lambda x.M' \quad M'[N/x] \Downarrow V}{M\ N \Downarrow V}$$

# Formalizing the theory of $\lambda_{\mathbf{cbn}}$

# Encoding of the syntax

The general methodology: define a datatype for each syntactic class of the language.

Two classes: *variables* and *terms*

```
Inductive tm : Set := var : Var -> tm
                    |  app : tm -> tm -> tm
                    |  lam : ...
```

What we put in place of `...` and for `Var` depends on the approach we will follow:

- first-order
- higher-order

# First-order approaches

*deep embedding*: write the encoding in the framework

**First-order abstract syntax** `Var` is an inductive set (e.g., `nat`)

$$\lambda \quad \rightsquigarrow \quad \texttt{lam : Var -> tm -> tm}$$

$$\lambda x.\lambda y.(xy) \quad \rightsquigarrow \quad \texttt{lam x (lam y (app x y))}$$

♠ Needs to implement and validate lots of machinery about $\alpha$-equivalence, substitution, . . .

**de Bruijn indexes** `Var`=1, the initial object

$$\lambda \quad \rightsquigarrow \quad \texttt{lam : tm -> tm}$$

$$\lambda x.\lambda y.(xy) \quad \rightsquigarrow \quad \texttt{lam (lam (app 1 0))}$$

♡ Good at $\alpha$-equivalence

♠ Not immediate to understand and needs even more technical machinery for capture-avoiding substitution than FOAS

We respect the rules of the game ⇒ Coq and Isabelle/HOL automatically provide induction principles to reason over processes

# Higher-order approaches

*Shallow embedding*: Change the rules, and write the encoding within the framework!

**Full HOAS [HHP87]** `Var` = `tm`

$$\lambda \quad \rightsquigarrow \quad \texttt{lam : (tm -> tm) -> tm}$$

$$\lambda x.\lambda y.(xy) \quad \rightsquigarrow \quad \texttt{lam [x:tm](lam [y:tm](app x y))}$$

♡ all aspects of variables management are delegated successfully to the metalanguage ($\alpha$-conversion, capture-avoiding substitution, generation of fresh names,. . . )

♠ incompatible with inductive types: the definition

```
Inductive tm : Set := app : tm -> tm -> tm
                    | lam : (tm -> tm) -> tm.
```

is not acceptable due to the negative occurrence of `tm`.

# Higher-order approaches (cont.)

**(Weak) Higher Order Abstract Syntax** `Var` is not tm, and

$$\lambda \quad \rightsquigarrow \quad \texttt{lam : (Var -> tm) -> tm}$$

$$\lambda x.\lambda y.(xy) \quad \rightsquigarrow \quad \texttt{lam [x:Var](lam [y:Var](app (var x) (var y)))}$$

♡ it delegates successfully many aspects of names management to the metalanguage ($\alpha$-conversion, capture-avoiding substitution of names/variables, generation of fresh names,. . . )

♡ compatible with inductive types ⇒ we can define functions and reason by case analysis on the syntax

♠ if `Var` is defined as inductive then *exotic terms* (= not corresponding to any real process of the object language) will arise!

? $\rightsquigarrow$ `lam [x:nat](Cases x of  0 => x  |  _ => (app (var x) (var x))  end)`

♠ metatheoretic analysis is difficult/impossible; e.g., structural induction over higher-order terms (*contexts*, terms with *holes*) is not provided

The Theory of Contexts addresses these problems from an "axiomatic standpoint".

# Encoding the syntax: avoiding exotic terms

Exotic terms arise only when a binding constructor has an inductive type in negative position (`lam : (Var -> tm) -> tm`).

Occam razor: `Var` is not required to be an inductive set

$\Rightarrow$ there is no reason to bring in induction/recursion principles and case analysis, which can be exploited for defining exotic terms

$\Rightarrow$ leave `Var` as an "open" set. Just assume it has the needed properties.

Complete definition (properties on `Var` will come later on):

```
Parameter Var : Set.
Inductive tm : Set :=  var : Var -> tm
                     |  app : tm -> tm -> tm
                     |  lam : (Var -> tm) -> tm.
Coercion var : Var >-> tm.
```

**Proposition 1** *For all $X$ finite set of variables, there is a bijection $\epsilon_X$ between $\Lambda_X$ and canonical terms of type* `tm` *with free variables in* `X`.

*Moreover, this bijection is compositional, in the sense that if $M \in \Lambda_{X,x}$ and $N \in \Lambda_X$, then $\epsilon_X(M[N/x]) = \epsilon_{X,x}(M)[\epsilon_X(N)/$*`(var x)`*$]$.*

# Encoding of substitution

Substitution of terms for variables is no longer delegated to the metalevel.

It is represented as a (functional) relation, whose derivations are syntax-driven.

```
Inductive subst [N:tm] : (Var->tm) -> tm -> Prop :=
     subst_var  : (subst N var N)
   | subst_void : (y:Var)(subst N [_:Var]y y)
   | subst_App  : (M1,M2:Var->tm)(M1',M2':tm)
          (subst N M1 M1') -> (subst N M2 M2') ->
          (subst N [y:var](app (M1 y) (M2 y)) (app M1' M2'))
   | subst_Lam  : (M:Var->Var->tm)(M':Var->tm)
          ((z:Var)(subst N [y:Var](M y z) (M' z))) ->
          (subst N [y:Var](lam (M y)) (lam M')).
```

The judgement "(subst N M M')" represents "$M' = M[N]$":

**Proposition 2** *Let $X$ be a finite set of variables and $x$ a variable not in $X$. Let $N, M' \in \Lambda_X$ and $M \in \Lambda_{X \uplus \{x\}}$. Then:*

$$M[N/x] = M' \iff \Gamma_X \vdash \_ : (\texttt{subst } \epsilon_X(N) \texttt{ [x:Var]}\epsilon_{X \uplus \{x\}}(M) \ \epsilon_X(M'))$$

# Encoding of semantics

Straightforward. The only remark is about the use of the substitution judgement.

```
Inductive red : tm -> tm -> Prop :=
      red_beta: (N,M':tm)(M:Var->tm)
                 (subst N M M') -> (red (app (lam M) N) M')
    | red_head: (M,N,M':tm)(red M M') -> (red (app M N) (app M' N)).
Inductive trred : tm -> tm -> Prop :=
    | trred_ref : (M:tm)(trred M M)
    | trred_trs : (M,N:tm)(red M N)->(P:tm)(trred N P)->(trred M P).
Inductive eval : tm -> tm -> Prop :=
      eval_var : (x:Var)(eval x x)
    | eval_lam : (M:Var->tm)(eval (lam M) (lam M))
    | eval_app : (M,M'',N,V:tm)(M':Var->tm)
                (eval M (lam M')) ->  (subst N M' M'') ->
                (eval M'' V) -> (eval (app M N) V).
```

The encoding is adequate; e.g.:

**Proposition 3** *Let $X$ be a finite set of variables; for all $M, N \in \Lambda_X$, we have $M \Downarrow N \iff \Gamma_X \vdash \_ :$ (*`eval` *$\epsilon_X(M)\ \epsilon_X(N)$).*

# Formalization of the MetaTheory of $\lambda_{\mathbf{cbn}}$

Following the methodology developed in [HMS98] and fully generalized in [HMS01]:

- Definition of occurrence predicates.
  Driven by the signature of the object language.

- Axiomatization of the Theory of Contexts.
  Parametric in the occurrence predicates.

- Development of theory (Have fun!)

# Occurrence predicates

```
Inductive notin [x:Var] : tm -> Prop :=
    notin_var : (y:Var)~x=y->(notin x y)
  | notin_app : (M,N:tm)(notin x M) -> (notin x N) -> (notin x (app M N))
  | notin_lam : (M:Var->tm)((y:Var)~x=y->(notin x (M y))) -> (notin x (lam M)).
```

```
Inductive isin [x:Var] : tm -> Prop :=
    isin_var : (isin x x)
  | isin_app1: (M,N:tm)(isin x M) -> (isin x (app M N))
  | isin_app2: (M,N:tm)(isin x N) -> (isin x (app M N))
  | isin_lam : (M:Var->tm)((y:Var)(isin x (M y))) -> (isin x (lam M)).
```

Roughly, "`(isin x M)`" means "$x$ occurs free in $M$".

Dually for `(notin x M)`: "$x$ does not occur free in $M$".

# The Theory of Contexts

A set of axiom schemata, which reflect at the theory level some fundamental properties of the intuitive notion of "context" and "occurrence" of variables. Their informal meaning is the following:

**Decidability of occurrence:** every variable either occurs or does not occur free in a term (generalizes decidability of equality on `Var`). Unnecessary if we are in a classical setting;

**Unsaturability of variables:** no term can contain all variables; i.e., there exists always a variable which does not occur free in a given term; (cfr. axiom F4 in Pitts' nominal logic)

**Extensionality of contexts:** two contexts are equal if they are equal on a fresh variable; that is, if $M(x) = N(x)$ and $x \notin M(\cdot), N(\cdot)$, then $M = N$.

**$\beta$-expansion:** given a term $M$ and a variable $x$, there is a context $C_M(\cdot)$, obtained by abstracting $M$ over $x$

# The Theory of Contexts for $\lambda_{\mathsf{cbn}}$

What of the Theory of Contexts we need in the present development:

```
Axiom LEM_OC: (M:tm)(x:Var)(isin x M)\/(notin x M).
Axiom unsat : (M:tm)(Ex [x:Var](notin x M)).
Axiom ext_tm : (M,N:Var->tm)(x:Var)
        (notin x (lam M)) -> (notin x (lam N)) ->
        (M x)=(N x) -> M=N.
Axiom ext_tm1 : (M,N:Var->Var->tm)(x:Var)
        (notin x (lam [z:Var](lam (M z)))) ->
        (notin x (lam [z:Var](lam (M z)))) ->
        (M x)=(N x) -> M=N.
```

Notice that we do not need $\beta$-expansion.

## Scared by axioms? Axioms are our friends!

The axiomatic approach helps us to split the problem in two (quite orthogonal) issues:

1. isolating a core set of fundamental properties of contexts, and to play with them in order to check their expressivity and "efficiency"

2. proving the soundness of these properties, or even deriving them from more basic (but possibly less natural) notions (like, e.g., in [Röckl et al., 2001])

Consistency of these axioms in Classical Higher Order Logic has been proved in [BHHMS01], by building a model following Hofmann's idea [Hof99]. The model is a classical tripos in a category of covariant presheaves. . . but this is another story.

Moreover, this model justifies also *recursion and induction principles* over higher-order types, which can be therefore safely assumed as needed

# Induction over `Var -> tm`

$$\frac{\begin{array}{l}(P\ \lambda x{:}\upsilon.(var\ x))\\ \forall y:Var.(P\ \lambda x{:}\upsilon.(var\ y))\\ \forall M_1{:}\upsilon \rightarrow \Lambda, M_2{:}\upsilon \rightarrow \Lambda.(P\ M_1) \wedge (P\ M_2) \Rightarrow (P\ \lambda x{:}\upsilon.(app\ (M_1\ x)\ (M_2\ x)))\\ \forall M_1 : \upsilon \rightarrow \upsilon \rightarrow \Lambda.(\forall y{:}\upsilon.(P\ \lambda x{:}\upsilon.(M_1\ x\ y))) \Rightarrow (P\ \lambda x{:}\upsilon.(\lambda(M_1\ x)))\end{array}}{\forall M{:}\upsilon \rightarrow \Lambda.(P\ M)}$$

```
Axiom tm_ind1 : (P:(Var->tm)->Prop)
       (P var) ->
       ((y:Var)(P [_:Var](var y))) ->
       ((M,N:Var->tm)(P M)->(P N)->(P [x:Var](app (M x) (N x)))) ->
       ((M:Var->Var->tm)
               ((y:Var)(P [x:Var](M x y)))->(P [x:Var](lam (M x))))
      -> (M:Var->tm)(P M).
```

but for all $n$, a similar schemata over `Var`$^n$`->tm` can be defined [HMS01b].

Similarly for recursions (recursors and equivalence (reduction) rules).

Compare it with "structural induction mod $\alpha$" in Pitts' nominal logic.

# Some results formally proved in Coq

- `subst` is deterministic (easier with higher-order inversion)
- `subst` is total (higher-order recursion)
- generation lemma for terms
- generation lemma for contexts (higher-order induction)
- substitution preserves free variables
- evaluation preserves free variables
- determinism (confluence) of evaluation
- determinism (confluence) of reduction
- equivalence of evaluation and reduction
- ...

```
Lemma subst_is_det: (M:Var->tm)(M1:tm)(subst N M M1) ->
                    (M2:tm)(subst N M M2) -> (M1 = M2).
Lemma sit: (N:tm)(M:Var->tm){M':tm | (subst N M M')}.
Lemma subst_is_total :  (N:tm)(M:Var->tm)(EX M' | (subst N M M')).
Lemma subst_isin : (M:Var->tm)(N,M':tm)(subst N M M') -> (x:Var)(isin x M') ->
                   (isin x N)\/(isin x (lam M)).
Lemma subst_notin : (M:Var->tm)(N,M':tm)(subst N M M') -> (x:Var)(notin x N) ->
                    (notin x (lam M)) -> (notin x M').
Lemma closed_generation : (M:tm)(closed M)->(EX C | (EX L | M=(lapp C L))).
Lemma closedschema_generation : (M:Var->tm)(closed (lam M))->
                  (EX C:Var->tm | (EX L:Var->ltm | M=[x:Var](lapp (C x) (L x)))).
Lemma reducts_are_values : (M,N:tm)(eval M N)->(isvalue N).
Lemma values_do_not_reduce : (N:tm)(isvalue N)->(eval N N).
Lemma eval_is_det : (M,V1:tm)(eval M V1)->(V2:tm)(eval M V2)->V1=V2.
Lemma eval_isin : (M,N:tm)(eval M N) -> (x:Var)(isin x N) -> (isin x M).
Lemma eval_notin : (M,N:tm)(eval M N) -> (x:Var)(notin x M) -> (notin x N).
Lemma values_do_not_red : (V:tm)(isvalue V)->(M:tm)(red V M)->False.
Lemma red_is_det : (M,V1:tm)(red M V1)->(V2:tm)(red M V2)->V1=V2.
Lemma red_eval : (M,N:tm)(red M N)->(V:tm)(eval N V)->(eval M V).
Lemma trred_eval : (M,V:tm)(trred M V)->(isvalue V)->(eval M V).
Lemma eval_trred : (M,N:tm)(eval M N) -> (trred M N).
```

# Functionality of substitution: some pragmatics

```
Parameter N:tm.
Lemma subst_is_det: (M:Var->tm)(M1:tm)
                    (subst N M M1) -> (M2:tm)(subst N M M2) -> M1=M2.
```

The proof goes by induction on the derivation of `(subst N M M1)`.

This gives rise to four cases:

```
  N : tm
  M : var->tm
  M2 : tm
  H : (subst N var M2)
  ============================
   N=M2
```

```
subgoal 2 is:
 (y)=M2
subgoal 3 is:
 (app M1' M2')=M0
subgoal 4 is:
 (lam M')=M2
```

For each case, inversion on `H` gives 4 subcases, 3 of which are absurd

```
subgoal 1 is:
  N : tm
  M : Var->tm
  M2 : tm
  H : (subst N var M2)
  H0 : var=var
  H1 : N=M2
  ============================
   M2=M2
...
```

```
subgoal 2 is:
  N : tm
  M : Var->tm
  M2 : tm
  H : (subst N var M2)
  y : Var
  H1 : ([_:var](y))=var
  H0 : (y)=M2
  ============================
   N=(y)
```

The inversion algorithm fails to eliminate absurd cases because the terms to discriminate on are higher-order. Absurd cases are (tediously) eliminated by using the Theory of Contexts (in particular `tm_ext`) and plain (i.e., first order) recursion.

The whole proof is 95 lines long, most of which are for dealing with the elimination of absurd cases.

# Functionality of substitution: higher-order inversion

*Higher-order inversion* lemmata can be (mechanically) proved from higher-order recursion principles (over Type).

```
Parameter subst_inv_fun  : tm -> (Var->tm) -> tm -> Prop.
Axiom subst_inv_fun_var0 : (N,M:tm)(subst_inv_fun N var M)==(N=M).
Axiom subst_inv_fun_var1 :
         (y:Var)(B,N:tm)(subst_inv_fun N [_:Var]y B)==((var y)=B).
Axiom subst_inv_fun_app : (A1,A2:Var->tm)(B,N:tm)
         (subst_inv_fun N [x:Var](app (A1 x) (A2 x)) B) ==
              (EX B1 | (EX B2 | (app B1 B2)=B /\ (subst N A1 B1)
                                               /\ (subst N A2 B2))).
Axiom subst_inv_fun_lam : (A:Var->Var->tm)(B,N:tm)
 (subst_inv_fun N [x:Var](lam (A x)) B) ==
             (EX A1 | (lam A1)=B /\ (y:Var)(subst N [x:Var](A x y) (A1 y))).

Lemma subst_inv : (A:Var->tm)(B,N:tm)(subst N A B) -> (subst_inv_fun N A B).
```

The proof of the inversion lemma is an extension of the algorithm implemented in Coq (Murthy and Cornes and Terrasse [CT96]).

Using higher-order inversion, the proof of `subst_is_det` is much easier (12 lines).

# Extending the object language: typing system

We extend the object language with a theory of simple types. Common definition: Types are defined by $\tau ::= u \mid \tau \to \tau$ where $u, v$ range over type variables.

Typing judgement: $\Gamma \vdash M : \tau$, where $\Gamma$ is the typing base, that is a finite set of pairs $x_1 : \tau_1, \ldots, x_n : \tau_n$. The usual typing rules are the following:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} x \notin dom(\Gamma)$$

The syntax of simple types is encoded trivially:

```
Parameter TVar : Set.
Inductive T : Set := tvar : TVar -> T | arr : T -> T -> T.
Coercion tvar : TVar >-> T.
```

# Modularity of the Theory of Contexts

The introduction of a typing system has a bearing on the structure of `Var`.

- before: `Var` may be any set satisfying `unsat` axiom and decidability of equality
- now: we require that every free variable is given a type, by assuming that
  - the existence of a type assignment: a map from variables to types
  - every fresh variable introduced by `unsat` must be given a type

```
Parameter typevar : Var -> T.
Axiom unsat_t : (M:tm)(s:T)(EX x | (notin x M) /\ (typevar x)=s).
```

Then, the encoding of the typing system is straightforward:

```
Inductive type : tm -> T -> Prop :=  type_var : (x:Var)(type (var x) (typevar x))
    | type_app : (M,N:tm)(s,t:T)
                 (type M (arr s t)) -> (type N s) -> (type (app M N) t)
    | type_lam : (M:Var->tm)(s,t:T)
                 ((x:Var)(typevar x)=s -> (type (M x) t))
                 -> (type (lam M) (arr s t)).
```

Since the locally assumed `x` is fresh, the assumption `(typevar x)=s` is safe

# More results formally proved in Coq

- preservation of types under renaming of variables (higher-order induction)
- preservation of types under substitution
- subject reduction for evaluation
- subject reduction for reduction (small-step semantics)

```
Lemma type_invar : (M:Var->tm)(s,t:T)
                   (x:Var)((typevar x)=s) -> (type (M x) t) ->
                   (y:Var)((typevar y)=s) -> (type (M y) t).
Lemma subst_preserves_types : (E:Var->tm)(N,M:tm)(subst N E M) ->
Lemma Subject_Reduction_eval : (M,V:tm)(eval M V)->(s:T)(type M s)->(type V s).
Lemma Subject_Reduction_red : (M,N:tm)(red M N)->(s:T)(type M s)->(type N s).
```

# Discussion

About the development

- Most of these proofs use built-in inductions (on plain terms and derivations), and the axioms `unsat`, `LEM_OC`, `ext_tm`, `ext_tm1`

- Some proofs required higher order induction (induction over contexts)

- Totality of substitution: higher-order recursion (induction in Set)

- Powerful higher-order inversion principles can be derived from higher-order recursion

- No proof has needed $\beta$-expansion — replaced by higher-order induction?

# Discussion (cont.)

The Theory of Contexts turned out to be successful

♡ smooth handling of schemata in HOAS

♡ no need of well-formedness predicate for ruling out exotic terms

♡ low mathematical and logical overhead: "proofs looks (almost) like on the paper". Almost, because of the explicit handling of substitution.

Weak points:

- compatible with Classical HOL but not with the Axiom of Unique Choice (AC!)
  $\Rightarrow$ not easily portable to metalogics containing AC!
  $\Rightarrow$ weak expressive power at the level of functions (which can be nevertheless recovered at the level of predicates)

- no automatization of inversion lemmata, yet

# Related work

[Despeyroux, Felty, Hirschowitz 1995]: closest to ours, but `Var=nat`.

+ no need of axioms

– well-formedness predicate (`valid`); all arguments are then carried out on terms which are *extensionally equivalent* to some valid term $\Rightarrow$ substantial overhead.

  E.g., for syntax, substitution, big-steps semantics, typing system and subject reduction: 500 lines in [DFH95], vs $<$ 300 lines within the Theory of Contexts.

[Momigliano, Ambler, Crole 2001] (good survey!): very similar theory and issues

- weak HOAS on an inductive set `Var`$=\{x,y\}$ $\Rightarrow$ well-formedness predicates
- overhead mitigated by automatization (higher in Isabelle than in Coq)
- totality of substitution requires the description axiom, which entails AC!, which is inconsistent with the Theory of Contexts [Hof99]

# Related work: meta-meta-logics

In previous approaches: we reason on objects of the metalogic (CIC, HOL,. . . ), in the metalogic itself.

A different perspective: add an extra logical level for reasoning over metalogics.

$FO\lambda^{\Delta N}$ [McDowell, Miller, 1997]:

- a higher-order intuitionistic logic extended with definitions, for reasoning on representations in simply typed $\lambda$-calculus
- it is possible to delegate the substitution to the metalanguage
- induction on types is recovered from induction on natural numbers via appropriate notions of measure
- it does not support a notion of "proof object" (and in the Theory of Contexts many properties of $\lambda_{\mathsf{cbn}}$ are derived by plain structural induction over proofs)

# Related work: meta-meta-logics

$\mathcal{M}_2$ [Pfenning and Schürmann, 2000]

- constructive first-order logic (based on ELF) for reasoning over (possibly open) objects of a LF encoding
- supporting higher-order induction and recursion
- aimed to a complete automatization $\Rightarrow$ difficult to compare with interactive approaches
- implemented in the theorem prover *Twelf*

# Work in progress

Abramsky's *applicative bisimulation*

- neatly encoded as a *coinductive predicate* (like strong late bisimulation in $\pi$-calculus)

```
CoInductive appsim : tm -> tm -> Prop :=
 appsim_coind : (M,N:tm)
                ((M':Var->tm)(eval M (lam M')) ->
                (EX N' | (eval N (lam N')) /\
                         (L,M'',N'':tm)(closed L) ->
                         (subst L M' M'') -> (subst L N' N'') ->
                         (appsim M'' N'')))
                -> (appsim M N).
```

- equivalence between applicative bisimulation and observational equivalence: at a good stage to its completion

# Work in progress (2)

Equivalence between different notions of $\alpha$-equivalence (Scagnetto):

- "Standard" (i.e., Barendregt's book) definition:

$$\frac{}{x \equiv_\alpha x} \qquad \frac{M \equiv_\alpha M' \quad N \equiv_\alpha N'}{(MN) \equiv_\alpha (M'N')} \qquad \frac{}{\lambda x.M \equiv_\alpha \lambda y.M[y/x]} y \notin FV(M)$$

- Alternative (Gabbay and Pitts) definition:

$$\frac{}{x \sim_\alpha x} \qquad \frac{M \sim_\alpha M' \quad N \sim_\alpha N'}{(MN) \sim_\alpha (M'N')} \qquad \frac{(zx) \cdot M \sim_\alpha (zy) \cdot N}{\lambda x.M \sim_\alpha \lambda y.N} z \text{ does not occur in } M, N$$

  where $(zx) \cdot M$ swaps *all* occurrences of $x$ by $z$.

- $\equiv_\alpha \subseteq \sim_\alpha$: done

- $\sim_\alpha \subseteq \equiv_\alpha$: in progress (difficulty: transitivity of $\equiv_\alpha$, not trivial)

## Other minor/work in progress case studies

**First Order Logic** full theory: validity judgement, substitution;
metatheory: functionality of substitution.

**spi calculus** full theory; metatheory: some algebraic laws

$\nu$**-calculus** theory

$\lambda\sigma$**-calculus** theory; some metatheoretic result

**Ambient calculus** language, congruence, logic, some result

Future work:

**Higher-order inversion** generalization of the Murthy-Cornes-Terrasse algorithm