On the formalization of λ-calculus and Tait-Girard's notion of computability (work in progress)

Frédéric Blanqui

INVENTEURS DU MONDE NUMÉRIQUE

3rd Workshop on Proof Theory and Rewriting
4-8 March 2013
Kanazawa, Japan

Goal

Formalize:

- higher-order rewriting, *i.e.* rewriting on λ-terms
- proofs of theorems on the termination of higher-order rewriting, especially those based on Tait-Girard's notion of computability (*e.g.* computability closure, HORPO)

Applications:

- formalization of calculi, type and logical systems using terms with binders, and certification of their properties (*e.g.* POPLmark challenge)
- certification of termination proofs of higher-order rewrite systems and functional programs

Higher-order rewriting

higher-order rewriting is rewriting on λ-terms

$\Rightarrow$ we first need to formalize λ-terms

problem: on λ-terms, substitution is usually defined modulo α-equivalence (renaming of bound variables)

$$(\lambda xy)_y^x = \lambda x' x$$

Curry and Feys' definition of α-equivalence (1958)

(ϱ)	$X\ \boldsymbol{R}\ X$	(reflexiveness),
(σ)	$X\ \boldsymbol{R}\ Y \Rightarrow Y\ \boldsymbol{R}\ X$	(symmetry),
(τ)	$X\ \boldsymbol{R}\ Y\ \&\ Y\ \boldsymbol{R}\ Z \Rightarrow X\ \boldsymbol{R}\ Z$	(transitivity),
(μ)	$X\ \boldsymbol{R}\ Y \Rightarrow ZX\ \boldsymbol{R}\ ZY$	(right monotony),
(ν)	$X\ \boldsymbol{R}\ Y \Rightarrow XZ\ \boldsymbol{R}\ YZ$	(left monotony).

(ξ) $X\ \boldsymbol{R}\ Y \Rightarrow \lambda x X\ \boldsymbol{R}\ \lambda x Y$.

(α) *If* y *does not occur free in* X, $\lambda y[y/x]X\ \boldsymbol{R}\ \lambda x X$.

Curry and Feys' definition of substitution (1958)

DEFINITION 1. $[M/x]X$ is the ob X^* defined as follows:

Case 1. X is a variable.

(a) If $X \equiv x$, then $X^* \equiv M$.
(b) If $X \equiv y \not\equiv x$, then $X^* \equiv X$.

Case 2. $X \equiv YZ$. Then $X^* \equiv Y^*Z^*$.

Case 3. $X \equiv \lambda y Y$.

(a) If $y \equiv x$, then $X^* \equiv X$.
(b) If $y \not\equiv x$, then $X^* \equiv \lambda z\, [M/x]\, [z/y]Y$,

where z is the variable defined as follows:

(i) If x does not occur free in Y, or if y is not free in M, then $z \equiv y$;
(ii) if x is free in Y and y is free in M, then z is the first variable in the list $e_1, e_2, \ldots$ such that $z \not\equiv x$ and z does not occur free in either M or Y.

follows then 10 pages to prove basic properties and in particular that substitution is compatible with α-equivalence (3 pages)...

Curry and Feys' Theorem 1 (1958)

THEOREM 1. *The ob* $[M/x]X$ *has the properties:*

(a) $$[x/x]X \equiv X.$$

(b) *If* x *does not occur free in* X,

$$[M/x]X \equiv X.$$

(c) *If no variables which occur free in* M *or* N *are bound in* X, *and* $N' \equiv [M/x]\,N$; *then*

(1) $$[M/x]\,[N/y]\,X \equiv [N'/y]\,[M/x]\,X$$

holds under either of the following conditions:

(c_1) y *does not occur free in* M,

(c_2) x *does not occur free in* X.

Curry and Feys' Theorem 2 (1958)

THEOREM 2. *If equality is α-convertibility, then:*

(a) *the relation*

$$X = Y \to [M/x]\,X = [M/x]\,Y \tag{2}$$

holds for all obs X, Y, M, and all variables x.

(b) *If X is an ob and $\mathfrak{v}$ a given finite set of variables; then there exists an ob Y such that*

$$X = Y,$$

and such that no variable of $\mathfrak{v}$ is bound in Y, whereas the bound variables of X not in $\mathfrak{v}$ are bound in the corresponding occurrences in Y.

(c) *If N',* (c_1), *and* (c_2) *are as in Theorem* 1c; *then*

$$[M/x]\,[N/y]\,X = [N'/y]\,[M/x]\,X \tag{3}$$

holds under either of the conditions (c_1), (c_2) *without further restriction.*

Curry and Feys approach (1958)

it takes 10 pages to prove basic properties and in particular that substitution is compatible with α-equivalence (3 pages)...

this may explain why Curry and Feys' definition has not been extensively used in formalizations and led many authors to try alternative approaches:

- de Bruijn indices (1972)
- higher-order features of meta-language [HOAS] (Pfenning and Elliot, 1988)
- de Bruijn indices for bound variables only (Huet, 1989)
- distinct name sets for bound and free variables (McKinna and Pollack, 1993)
- nominal terms (Gabbay and Pitts, 1999)
- terms with height function (Sato and Pollack, 2008)
- ...

Formalizations following Curry and Feys's approach

- 1985: Shankar in NQTHM:
  - confluence of β
- 2000: Ford and Mason in PVS:
  - confluence of Landin's call-by-value Iswim
  - CIU theorem (Closed Instances of Uses)
- 2001: Homeier in HOL:
  - confluence of $\beta\eta$
- 2003: Vestergaard and Brotherston in Isabelle/HOL:
  - finite developments
  - standardization theorem
  - ...

Formalizations of termination proofs

most formalization works consider confluence or safety properties

few consider termination properties:

- 1993: Altenkirch in Lego:
  termination of system F
  using full de Bruijn indices
- 2004: Koprowski in Coq:
  termination of HORPO
  using full de Bruijn indices
- 2006: Berger *et al* in Coq, Isabelle/HOL and Minlog:
  termination of simply typed λ-calculus
  using full de Bruijn indices or HOAS

My formalization work in Coq

See `http://color.inria.fr`.

Definitions can be seen on `http://color.inria.fr/doc/main.html`:

- `LTerm`: λ-terms
- `LSubs`: higher-order substitution
- `LAlpha`: α-equivalence
- `LBeta`: β-reduction
- `LComp`: some axiomatization of Tait's computability
- `LSimple`: simply typed λ-terms
- `LCompSimple`: termination of β on simply typed λ-terms

To see the proofs, you need to download CoLoR.

λ-terms

```
Variables F X : Type.

Inductive Te : Type :=
| Var (x : X)
| Fun (f : F)
| App (u v : Te)
| Lam (x : X) (u : Te).

(* using Coq standard library on finite sets: *)

Parameter var_notin : XSet.t -> X.

Parameter var_notin_ok : forall xs, ~In (var_notin xs) xs.
```

Substitution

We extend Curry and Feys' definition to simultaneous substitutions (like Stoughton 1988 but we rename only when it is necessary)

```
Definition var x u s :=
  let xs := fvcodom (remove x (fv u)) s in
    if mem x xs then var_notin (union (fv u) xs) else x.

Fixpoint subs (s : X -> Te) (t : Te) :=
  match t with
    | Var x => s x
    | Fun f => t
    | App u v => App (subs s u) (subs s v)
    | Lam x u => let x' := var x u s in
      Lam x' (subs (update x (Var x') s) u)
  end.
```

α-equivalence

```
Inductive aeq : relation Te :=
| aeq_refl : forall u, aeq u u
| aeq_sym : forall u v, aeq u v -> aeq v u
| aeq_trans : forall u v w, aeq u v -> aeq v w -> aeq u w
| aeq_app_l : forall u u' v,
  aeq u u' -> aeq (App u v) (App u' v)
| aeq_app_r : forall u v v',
  aeq v v' -> aeq (App u v) (App u v')
| aeq_lam : forall x u u',
  aeq u u' -> aeq (Lam x u) (Lam x u')
| aeq_alpha : forall x u y,
  ~In y (fv u) -> aeq (Lam x u) (Lam y (rename x y u)).

Infix "~~" := aeq (at level 70).
```

Compatibility of substitution wrt α-equivalence

```
Instance subs_aeq : Proper (Logic.eq ==> aeq ==> aeq) subs.
```

Most difficult theorem.
Curry and Feys' proof takes about 100 lines.
My proof is about 200 lines with no automation.

To conduct the proof, we use a renaming-free substitution `subs1`:

- equivalent to `subs` when bound and free variables are distinct
- easier to work with

Renaming-free substitution

```
Fixpoint subs1 s (t : Te) :=
  match t with
    | Var x => s x
    | Fun f => t
    | App u v => App (subs1 s u) (subs1 s v)
    | Lam x u => Lam x (subs1 (update x (Var x) s) u)
  end.

Lemma subs1_no_alpha : forall u s,
  inter (bv u) (fvcodom (fv u) s) [=] empty
  -> subs s u = subs1 s u.
```

Dealing with α-equivalence explicitly

```
Lemma aeq_notin_bv : forall xs u,
  exists v, u ~~ v /\ inter (bv v) xs [=] empty.

Lemma saeq_notin_bvcodom :
  forall ys s xs, exists s', saeq xs s s'
  /\ inter (bvcod xs s') ys [=] empty /\ dom_incl xs s'.
```

some axiomatized version of Tait's computability

CP structures (part 1/3)

```
Module Type CP_Struct.

  (** We assume given a relation [->Rh] and a predicate
    [neutral] that is compatible with alpha-equivalence.*)

  Parameter Rh : relation Te. Infix "->Rh" := Rh (at level
  Parameter neutral : Te -> Prop.
  Declare Instance neutral_aeq : Proper (aeq ==> impl) neut

  (** We denote by [->R] the monotone closure of [->Rh]
  and by [=>R] the closure by alpha-equivalence of [->R].*)

  Notation R := (clos_mon Rh). Infix "->R" := (clos_mon Rh)
  Notation R_aeq := (clos_aeq (clos_mon Rh)).
  Infix "=>R" := (clos_aeq (clos_mon Rh)) (at level 70).
```

CP structures (part 2/3)

```
(** Properties of [neutral]. *)

(* abstractions are not neutral *)
Parameter not_neutral_lam :
  forall x u, ~neutral (Lam x u).

(* variables and beta-redexes are neutral *)
Parameter neutral_var : forall x, neutral (Var x).
Parameter neutral_beta :
  forall x u v, neutral (App (Lam x u) v).

(* if [u] is neutral then [App u v] is neutral *)
Parameter neutral_app :
  forall u v, neutral u -> neutral (App u v).
```

CP structures (part 3/3) - Properties of [=>R]

```
(* [=>R] is stable by substitution *)
Declare Instance subs_R_aeq :
  Proper (Logic.eq ==> R_aeq ==> R_aeq) subs.

(* [=>R] preserves free variables *)
Declare Instance fv_Rh : Proper (Rh --> Subset) fv.

(* variables and abstractions are not head-reducible *)
Parameter not_Rh_var : forall x u, ~ Var x ->Rh u.
Parameter not_Rh_lam : forall x u w, ~ Lam x u ->Rh w.

(* the head-reduct of a beta-redex is its beta-reduct *)
Parameter Rh_bh : forall x u v w,
  App (Lam x u) v ->Rh w -> App (Lam x u) v ->bh w.

(* if [u] is neutral then [App u v] is not head-reducible *)
Parameter not_Rh_app_neutral :
  forall u v w, neutral u -> ~ App u v ->Rh w.
```

Computability predicates `P : pred := Te -> Prop`

```
(* computability is stable by alpha-equivalence *)
Definition cp_aeq (P : pred) := (Proper (aeq ==> impl) P).

(* computability implies termination wrt [=>R] *)
Definition cp_sn (P : pred) := forall u, P u -> SN R_aeq u

(* computability is stable by reduction wrt [=>R] *)
Definition cp_R_aeq (P: pred) := Proper (R_aeq ==> impl) P

(* neutrals are computable if their [=>R]-reducts so are *
Definition cp_neutral (P : pred) := forall u, neutral u ->
  (forall v, u =>R v -> P v) -> P u.

Class cp P := { cp1 : cp_aeq P; cp2 : cp_sn P;
                cp3 : cp_R_aeq P; cp4:cp_neutral P }.
```

Properties of CP structures

```
(* the set of terminating terms is a CP *)
Lemma cp_SN : cp (SN R_aeq).

(* [arr P Q] is a CP if both [P] and [Q] so are *)
Definition arr (P Q : pred) u :=
  forall v, P v -> Q (App u v).
Lemma cp_arr : forall P Q, cp P -> cp Q -> cp (arr P Q).

(* sufficient conditions for a beta-redex to be computable
Lemma cp_beta : forall P, cp P ->
    forall x u v, SN R_aeq (Lam x u) -> SN R_aeq v ->
    P (subs (single x v) u) -> P (App (Lam x u) v).
```

Simple types

```
Variable B : Type.

Inductive Ty : Type :=
| Base : B -> Ty
| Arr : Ty -> Ty -> Ty.

Infix "~~>" := Arr (at level 55, right associativity).
```

Simply typed λ-terms

```
Parameter typ : F -> Ty.

Notation En := (XMap.t Ty).

Inductive tr : En -> Te -> Ty -> Prop :=
| tr_var : forall E x T, MapsTo x T E -> tr E (Var x) T
| tr_fun : forall E f, tr E (Fun f) (typ f)
| tr_app : forall E u v V T,
  tr E u (V ~~> T) -> tr E v V -> tr E (App u v) T
| tr_lam : forall E x X v V,
  tr (add x X E) v V -> tr E (Lam x v) (X ~~> V).

Notation "E '|-' v '~:' V" := (tr E v V) (at level 70).
```

Properties

```
Lemma tr_strengthening : forall E v V, E |- v ~: V ->
  forall y, ~XSet.In y (fv v) -> remove y E |- v ~: V.

Definition wt s E F :=
  forall x T, MapsTo x T E -> F |- s x ~: T.

Lemma tr_subs : forall E v V, E |- v ~: V ->
  forall s F, wt s E F -> F |- subs s v ~: V.

(* [tr] is stable by alpha-equivalence *)
Instance tr_aeq_impl :
  Proper (Equal ==> aeq ==> Logic.eq ==> impl) tr.

(* subject reduction for [beta_aeq] *)
Instance tr_beta_aeq :
  Proper (Logic.eq ==> beta_aeq ==> Logic.eq ==> impl) tr.
```

Termination of β for simply typed λ-terms (part 1/2)

```
Module SN (Import P : CP_Struct).

  Variables (Bint : B -> pred)
    (cp_Bint : forall b, cp (Bint b)).

  Fixpoint int T :=
    match T with
    | Base b => Bint b
    | Arr A B => arr (int A) (int B)
    end.

  Definition valid E s :=
    forall x T, MapsTo x T E -> int T (s x).

  Lemma valid_id : forall E, valid E id.
```

Termination of β for simply typed λ-terms (part 2/2)

```
  Variable comp_fun : forall f, int (typ f) (Fun f).

  Lemma tr_int : forall v E V, E |- v ~: V ->
    forall s, valid E s -> int V (subs s v).

  Lemma tr_sn : forall E v V, E |- v ~: V -> SN R_aeq v.

End SN.
```

Conclusion (part 1/2)

I have formalized in Coq:

- pure λ-calculus by strictly following Curry and Feys' definitions:
  - λ-terms are first-order terms with named variables
  - substitution explicitly renames bound variables when necessary
  - α-equivalence is considered explicitly
    but I do not consider α-equivalence classes
- some axiomatized version of Tait-Girard's comp. predicates
- Curry-style simple types on the pure λ-calculus
- termination of β-reduction on simply-typed λ-terms

Conclusion (part 2/2)

- "basic" properties of substitution are indeed difficult to prove (*e.g.* stability by α: 200 lines vs 100 lines in Curry and Feys)
- but current proof could be improved by using more automation (general tactic for reasoning on finite sets + specific tactics)
- and, once this is done, things are not so difficult so far and similar to paper-and-pen proofs by:
  - checking that functions and predicates are invariant by α-equivalence (these properties are then automatically taken into account by Coq thanks to Mathieu Sozeau's type class system, 2008)
  - using explicit renamings when necessary
- but we need to formalize more complex calculi/proofs to draw more general conclusions

Thank you for your attention!

Questions?