Towards Formalising the Guard Checker of Coq

MPRI M1 Internship Project

Yee-Jian Tan

27 August, 2024

Advisor: Yannick Forster Cambium, Inria Paris

Eliminators and Fixpionts

- · Coq is based on the Calculus of *Inductive* Constructions (CIC) [1].
- To construct: constructors

To eliminate: eliminators (aka recursors, destructors), or <u>fixpoints + match</u>.

```
Fixpoint add (m \ n : nat) {struct m} := match m with 0 => n \mid S \mid m' => add \mid m' \mid (S \mid n) \mid end.
```

· Advantage: extracted code to e.g. OCaml is more idiomatic

Eliminators and Fixpionts

Unrestricted fixpoints can be non-terminating...

```
#[bypass_check(guard)]
Fixpoint boom (n : nat) : False := boom n.
and break consistency!
Check (boom 0). (* False *)
```

Eliminators and Fixpionts

Check (boom 0). (* False *)

Unrestricted fixpoints can be non-terminating...

```
#[bypass_check(guard)]
Fixpoint boom (n : nat) : False := boom n.
and break consistency!
```

How does Coq avoid non-termination?

The guard checker! It checks for **structural recursion**.

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

How does Coq avoid non-termination?

The guard checker! It checks for **structural recursion**.

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

How does Coq avoid non-termination?

The guard checker! It checks for **structural recursion**.

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Simple!

Can we trust Coq?

If the theory of Coq is consistent (and the implementation faithful).

Consistency: there is no term of Empty type in the empty context.

Ingredients for consistency

- 1. (Weak) Normalisation: every term has a normal form.
- 2. Subject Reduction: reduction preserves typing.
- 3. **Canonicity**: for inductive types, normal forms begin with a constructor in the empty context.

Proof: In the empty context, any term of the Empty type must have a normal form (1) of the same type (2). Since the context is empty, it must begin with a constructor (3), but the Empty type has none.

Can we trust Coq?

If the theory of Coq is consistent (and the implementation faithful).

Consistency: there is no term of Empty type in the empty context.

Ingredients for consistency

- 1. (Weak) Normalisation: every term has a normal form.
- 2. Subject Reduction: reduction preserves typing.
- 3. **Canonicity**: for inductive types, normal forms begin with a constructor in the empty context.

Proof: In the empty context, any term of the Empty type must have a normal form (1) of the same type (2). Since the context is empty, it must begin with a constructor (3), but the Empty type has none.

Another example:

```
Fixpoint minus (a b : nat) {struct a} :=
  match a, b with
  | 0 , _ => 0
  | _ , 0 => a
  | S a', S b' => minus a' b'
  end.
```

```
Fixpoint div (m n : nat) {struct m} :=
  match m with
  | 0 => 0
  | S k => S (div (minus k n) n)
  end.
```

```
Fixpoint minus (a b : nat) {struct a} :=
  match a, b with
  | 0 , _ => 0
  | _ , 0 => a
  | S a', S b' => minus a' b'
  end.

div is not guarded! Why?
```

```
Fixpoint div (m n : nat) {struct m} :=
  match m with
  | 0 => 0
  | S k => S (div (minus k n) n)
  end.
```

```
Fixpoint minus (a b : nat) {struct a} :=
  match a, b with
  | 0 , _ => 0
  | _ , 0 => a
  | S a', S b' => minus a' b'
  end.
```

Because 0 is not a subterm of m!

```
Fixpoint div (m n : nat) {struct m} :=
  match m with
  | 0 => 0
  | S k => S (div (minus k n) n)
  end.
```

This is structural!
Things are not as simple as they seem.

```
Fixpoint div (m n : nat) {struct m} :=
  match m with
  | 0 => 0
  | S k => S (div (minus k n) n)
  end.
```

The Guard Checker of Coq

	rel_min = renw.rel_min*n;	match kind o' with	let () = memort (Lint.in_empty large) in	(match mubt with	elt :: stack' →		check_rec_call_state renv needreduce_fix non_absorbed_stack rs (fun	match stack with
(* Check if t is a subterm of Rel n, and gives its specification, assuming lat already gives index of	genv = iterate (fun ge -> lazy Not_mubters::ge) n renv.genv }	Prod(na,a,b) -> let ienv' - ienv_push_ver ienv (na,a,ek_norec) in	let recarg = build_recargs innv (list.hd trees) b in let innv' = innv_push_var innv (ra,b,ek_norec) in	Subters (, ,s, wf) -> (" We take the subters specs of the constructor of the record ")	(** Y2: run below until stack is empty. First who [ar] into [t], *) let t = whd_all ?evers env ar in	(* Check if [def] is a guarded fixpoint body with decreasing arg. given [recpos], the decreasing arguments of each mutually defined	() → (* we try hard to reduce the fix away by looking for a	elt :: stack -> let rs - check_inert_subterm_rec_call remv rs a in
subterms with corresponding specifications of recursive arguments *)	let push_fix_renv renv (_,v,_ as recdef) = let n = Array_length v in	ienv_decompose_prod Tevars ienv' (n-1) b → masert false	recargs_constr_rec ienv' (List.tl trees) (recarg::lrec) d	let wf_args = (dest_mubterss wf).(8) in (* We extract the tree of the projected argument *)	eatch kind t with (** YJ: There are only 2 cases for the type of [o] here: Prod or anythin	fispoint. *) slet check one fix Tevers remy recoon trees def *	constructor in [decrarg] (we unfold definitions too) *) if List.length stack <- decrarg them None else	let renv', stack', body' - pop_argument NoNeedReduce renv elt stack x a body in
(* A powerful notion of subterm *)	let n = Array.length v in { env = push_rec_types recdef renv.env; rel min = renv.rel min=n;	(* This removes clotal parameters of the inductive types in 1c (for	List.rev lrec	let n = Projection.arg p in spec of tree (List.nth wf aros n)	else. We only care if type of [p] is a Prod. Otherwise we map the stack	let ofi - Array.length recpos in	match List.oth stack decrarg with Sara -> (" A match on the way ") None	check_nested_fix_body illformed renv' (decr-1) stack' rs body
(* To each inductive definition corresponds an array describing the structure of recursive arguments for each constructor, we call it	genv = iterate (fun ge -> lazy Not_subters::ge) n renv.genv }	nested inductive types only) *) Let dummy_univ = Level.(make (Oliobal.make (OirPath.make [Id.of_string	recargs_constr_rec ienv trees [] c	Dead_code -> Dead_code Not_subters -> Not_subters	We only care if type of $[p]$ is a Prof. Otherwise we map the stack into Sirgs. *) Prof $(a,a,cb) \rightarrow$ (** Y2: open the binder $[ctx:a]$ into ctx *)	(* Checks if [t] only make valid recursive calls [stack] is the list of constructor's argument specification and	%lower (n,reckrg) → let c = whd_all ?evers renv.env (lift n reckrg) in	let renv' = push_var_renv renv (redex_level rs) (x,a) in check_nested_fix_body illformed renv' (decr-1) [] rs body
the recursive spec of the type (it has type recards vect), for	type fix_check_result = Needleduce of env * fix_guard_error	"implicit"[] == 0])	(* starting with ra_env = [] seems safe because any unbounded Rel will be	Internally_bound_subters s + Internally_bound_subters s)	(** YJ: open the binder [ctx : s] into ctx *)	arraments that will be arelied after reduction		end -> illformed ()
checking the guard, we start from the decreasing argument (Rel n) with its recursive spec. During checking the guardness condition,	Needleduce of env " fix_guard_error NeNeedleduce	let dummy_implicit_mort = mkType (Universe_make dummy_univ) let lembde_implicit n m =	sszigned Norec ") build_recargs_nested (env,[]) tree (ind, args)	Const c →	let d = localAssum (n, a) in let cts, a = whd_decompose_prod_decls ?evers env a in	example u in t where we have (match with + t end) u; [rx] is the stack of redexes traversed u/o having been triggered *)	match kind hd with Construct _ → Some (contract_fix fix, stack)	
we collect patterns variables corresponding to subterms of m, each of them with its recursive spec. They are organised in a list lat	(* Definition and manipulation of the stack *)	let mon = Context.make_annot Anonymous Sortx.Relevant in let lambds_implicit a = mklambds (anon, dummy_implicit_mort, a) in	* [restrict_spec env spec p] restricts the size information in spec to what i	begin try is let _ = Environ.constant_value_in renv.env c in Not_subters	<pre>let env = push_rel_context ctx env in (** YJ: whd [a], which should have form of an application [ty args]. *.</pre>	let rec check_rec_call_stack renv stack rs t = match kind t with	Cofix Ind Lambda Prod Latin Sort Int Float Array -> assert false	and check_rec_call_state renv meedreduce_of_head stack rs espand_head = (* Text if either the head or the stack of a state
of type (int * recarge) list which is morted with respect to the first argument.	type stack_element * (" arguments in the evaluation stack ") (" [countr] is typed in [guard_env] and [int] is the number of	iterate lambda_implicit n a	allowed to flow through a match with predicate p in environment env. *) ** TODO TJ: what is this predicate thing about? check the usage to find out.	with NotivaluableCorat (InFrieitive (_u,op)) when list.length 1 >=	let ty, args = decompose_app_list (whd_all Tevars env a) in let elt = match kind ty with	App (f,args) → begin	Pel _ Nar _ Coret _ App _ Case _ Fix _ Proj _ Cast _ Meta _ Evar _ → None)	needs the state to be reduced before continuing checking *) match needreduce_of_head needreduce_of_stack stack with
	binders added in the current env on top of [quard env.env] *)	let abstract_mind_lc otype spars mind lc = let lc = Array.mag (fun (ctx. c) → Term.it skProd or LetIn c ctx) lc in) ** YI: if your is [Not without atr than on restriction to it. Otherwise.	O'rimitives.arity op -> orimitive specif Severs reny on 1	(** N): if the binder type [a] is some inductive smalled with [arms]	begin let rs, stack = Array.fold right (fun a (rs.stack) ->	Const (km. u as cu) →	Noticediation or m
(*************************************	Stlomare of fix_check_result * guard_env * int * constr (* arguments applied to a "match": only their spec traverse the match *) Skrg of mubters_spec Lary.t	let rec replace ind k c =	unwrap [p] as much as possible - forall, lambda. If the head of application of [p] is an inductive, then return the intersection of spec with spec of	m NotfvaluableConst _ → Not_subterm	then it could be a subtame. ") ("" II: If the spec of [alt] is subtame, intersect and return subtame, otherwise return spec. ")	let medicaluse, rs = check_rsc_call renv rs a in	<pre>check_rec_call_state renw NoBeedReduce stack rs (fun () -> if evaluable_constant kn renv.env then Some (constant_value_in</pre>	(* Expand if possible, otherwise, last chance, propagate need for expansion, in the loose to be executed by grant *)
(* tells whether it is a strict or loose subters *)	SArg of subters spec Lazy.t	let the replace_ins c = " let th, args = decompose_app_list c in match kind hd with Ind ((mind',i),_) when Nutled_CasDed.equal mind mind' → ([unerapped p]. ") "" NOTE U: this is to fix propest after beta-iota cuts. Lennard's code has	Var _ Sort _ Cast _ Prod _ LetIn _ App _ Ind _	subterm. otherwise return spec. *)	let stack = push_stack_closure renv needreduce a stack in (rs,stack)) args (rs,stack)	renv.env cu, [])	for expansion, in the hope to be eventually erased *) match expand/head () with Nime > a :: List.tl rs
type size = Large Strict	Let () x y = metch x with		more info about the rule and ones remarks about the lovin but it is	Construct Coffs Let Float Arms on Not subtarn	let year a whark planent year if Tourne alt in	check_rec_call_stack remy stack rx f	Lambda (x.s.b) ->	Some (c, stack') → check_rec_call_stack renv (stack'@stack) rs :
(* merging information *)	Needleduce → x Neleedleduce → y	:: args when n > 0 -> drop_params (n-1) args args -> lambda_implicit n (ferm.applist (mkRel (ntype-n-k-i),	basically translating code to english. The top of his document says he do not understand why the restriction either. Maxime Denex who implemented th	15	let marg = lary (match Lazy.force spec with Not_mubtern Dead_code Internally_bound_nubtern _ am spec -	end .	begin	and check_inert_subtarm_rec_call renv rs c =
let sim_glb a1 s2 - match s1,s2 with	let per perfective of start a function	List.Smart.map (replace_ind (e-k)) args)) in	first version of this could perhaps give an explanation.") at restrict_spec ?evers env spec p =	(* Other terms are not subterms *)	spec Subters(1, s, path) →	Rel p → begin	let medreduce, rs - check_rec_call renv rs a in eatch stack with	(* Check rec calls of a term which does not interact with its immediate context and which can be possibly erased at higher
Strict, Strict → Strict → Large	[] → NoNeedReduce Skrg _ :: 1 → needreduce_of_stack 1	drop_params npars args _ → map_with_binders succ replace_ind k c	match spec with Not_subtern Internally_bound_subtern _ → spec	and lazy_materm_specif Tevars renv stack t = lazy (matterm_specif Tevars renv stack t)	(** Y): (bestion: in the final intersection, the contribution of [p] boils down to [args] only. *)	(* Test if [p] is a fixpoint (recursive call) *) if remurel_min <- p EE p < remurel_min of i then	elt :: stack -> let renv, stack, b - pop_argument ?evers needreduce renv elt	level of the redex stack *) let need_reduce, rs = check_rec_call renv rs c in
(* rearible marifications for a turn-	Stlosure (needreduce, , , , ,) :: 1 → needreduce needreduce_of_stack 1	in Array.map (replace ind 0) lc		and stack element specif Tevars - function	(** YJ: Find oud how get_recarge_approx uses args (which does not really sake sense). *)	(* the position of the invoked fispoint: *) let alob = renv.rel min-ofi-1-o in	stack x a b in check rec call stack reny stack rs b	check_rec_call_state renv meed_reduce [] rs (fum () -> None)
- Not subterm: when the miss of a term is not related to the recursive argument of the fixpoint	let redex_level rs = List_length rs	but to extend the continue contains are a	(* Optimization: if the predicate is not dependent, no restriction is needed	Sloure (, h_renv, _, h) → lazy_subterm_specif Tevers h_renv [] h Skrg x → x	<pre>let recarge = get_recarge_approx Tevars env path ind args in let path = inter_wf_paths path recargs in</pre>	(* the decreasing arg of the rec call: *) let up = recos.(glob) in	[] → check_rec_call_stack (push_var_renv_renv (redex_level_re)	and check_rec_call renv rs c =
- Internally_bound_subters: when the recursive call is in a subters of a redex and the recursive arousent is bound to a variable	let push_stack_closure renv medreduce c stack = (Stlosure (medreduce, renv. 6, c)) :: stack	match env.retroknowledge.Retroknowledge.retro_array with	if noccur_with_mets 1 (Context.Rel.length abacts) or then spec	and extract stack Tevers - function	Subters(1,x,path))	if List.length stack <- op then set need reduce too renv.erv (Notinouchirossentsforfis(all)	(x,n) [] rs b	(* either fails if a non guarded call occurs or tells if there is rec call on a variable bound at the top of [c] and update the need for reduction in the redex stack with rec calls on
which will be instantiated by reductor the reduct the inteners	lat such stark classes new 1 stark s	Some c' when (Constant.equal env c c' -> true -> false	let env = push_rel_context absctx env in	[] → Lary.from.wal Not.mobilers, [] elt :: 1 → stack_element_specif Tevers elt, 1	Sing many > Sing (met into specif or (lasy Not subtorm))	glob) rs	ens	variables bound at higher levels of the redex stack ") List.me first (check rec call stack reny () (Selece(Bekuce:rs) c)
refer to the number of redeces stacked, with 1 counting for the variables bound at head in the body of the fix (ex e.g. [x] in	List.fold_right (push_stack_closure renv NoNeedNeduce) 1 stack	(* [get_recargs_approx env tree ind args] builds an approximation of the	let arcts, s = whd_decomposs_prod_decis Yevers env ar in let env = push_rel_context arcts env in		in .	(" Retrieve the expected tree for the argument ")	Prod (x,s,u) → (* Note: we cannot ensure that the stack is empty because	List.mp_first (check_rec_dall_stack renv [] (someonouch::rs) c)
[fix f n := fun x => f x]); there may be several such indices because [match] subterms may have combine several results;	let push_stack_args 1 stack =		let i,args = decompose_app_list (whd_all ?evers env s) in match kind i with	and primitive_specif Towars renv op args = let open Offinitives in	elt :: filter_stack (push_rel d env) of stack' (** YJ: if the stack element is not an inductive, it must not be a	(* Check the decreasing arg is smaller *) let x = List.nth stack up in	non-accessible branches of "match" expressions can have arbitrary types (see #17873) *)	in let need_reduce, rx = check_rec_call renv [] def in
 Subterm: when the term is a subterm of the recursive argument the wf_paths argument specifies which subterms are recursive; 	List fold_right (fun spec stack \Rightarrow SArg spec :: stack) 1 stack	nested types should be traversed, pruning the tree otherwise. This code is very close to check positive in indtypes.el, but does no positivity check and does	Ind i -> begin match spec with	match op with Arrayget Arraydefault →	minters. *) _ → List.fold_right (fun _ 1 → SArg (set_ints_specif or (lary	match check_is_subters (stack_element_specif Newers z) tree (glob) with	e. Let rs - check inert_matterm_rec_call renv rs a in (" Note: can recursive calls on [x] be also than inert "dead code"?	memort (List.is_empty rs); match need_reduce with
the [int list] is used in the [match] case where one branch of the [match] might be a subterm but (an arbitrary number of)	let lift_stack k = List_esp (function	not complete the number of recursion arranged a *)	Dead_code -> spec Gibbare() at tree) us	(* t.[i] and default t can be seen as strict subteres of t, with a sotertially rested rectree. *)	Not_subters)) :: 1) stack []	NeedleduceSubters 1 → set_meed_reduce renv.env 1 (illens) rec call renv plob x) rs	*) theck rec call stack (push var reny reny (redex level rs) (x.a)) []	Sandadora (any arr) sa raina (Fishmedirece (any arr))
others are calls to bound variables - Dead code: when the term has been built by elimination over an	Silonure (needreduce, z, n, c) \rightarrow Silonure (needreduce, $z, n+k, c$) $x \rightarrow x$)	let get_recarge_approx Swars env tree ind args = let rec build_recarge (env, ra_env as ienv) tree c =	let recarge = get_recarge_approx Newers env tree i args in let recarge = inter_wf_paths tree recarge in	let arg = List.nth args 1 in (* the result is a strict subters of the	filter_stack env ar stack	Invalid&bbtes -> raise (fixQuardfror (resv.esv, illegal_rec_call resv glob z))	rs u	let inductive_of_mutfix ?evers env ((nvect_bodynus),(names,types,bodies as
empty type		let x,largx = decompose_app_list (whd_all ?evarx env c) in	Subters(1,st,recargs)	Let mubt - mubterm_specif Tevars renv [] arg in	let find_uniform_parameters recinds margs bodies -	elze	Cofix (i.(.tuparray.bodies as recdef)) →	let notice Array length bodies in
	let lift1_stack = lift_stack 1	match kind x with Prod (ma,b,d) → assert (Listin, mapty large);	_ → assert false and	begin match subt with Subterm (_, _m, wf) → let wf_args = (dest_subterms wf).(0) in	let shodies = Array.length bodies in let min_inds = Array.fold_left min mergs recinds in (" Me work only on the i-th body but are in the context of n bodies ")	<pre>check_rec_call_state renv NoNeedReduce stack rs (fun () -></pre>	typarray in	let nbfix = Array.length bodies in if Int.equal nbfix 0 not (int.equal (Array.length nwect) nbfix)
type subterm_spec = Subterm of (Int.Set.t * size * wf_paths)	(* (6 Computing the recursive subterms of a term (propagation of size information through Cases).) *)	build_recargs (ienv_push_var ienv (na, b, mk_norec)) tree d	_ → Not_subtern	<pre>let wf_args = (dest_subterms wf).(0) in spec_of_tree (List.nth wf_args 0) (* first and only parameter of 'array</pre>	let rec aux i k nuniformparams c =	match lockup_rel p revo.env with local# -> None local# -> None (lift p c, []))	let rerv' = push_fix_rerv rerv recdef in Array_fold_left (fum rx body →	not (Int.equal (Array.length types) nbfix)
Dead_code Not subtern	information through Cases).] *)	(* Free variables are allowed and assigned Norsc *)	* [subters_specif renv t] computes the recursive structure of [t] and compare its size with the size of the initial recursive argument of	*) Dead code → Dead code	let f, 1 = decompose_app_list c in match kind f with	end	let medreduce', rx = check_rec_call renv' rx body in check rec_call state renv medreduce' stack rx (fun -> None))	not (Int.equal (Array.length names) nbfix) bodemum < 8
Internally_bound_subters of Int.Set.t	let lookup_mibterss env ind = let (_,wip) = lookup_mind_specif env ind in	(try and (List.nth ra_env (k-1)) with failure _ Invalid_argument _ → sk_morec)	the fispoint we are checking. [renv] collects such information about variables.	Not_matters → Not_matters Internally_bound_matters n → Internally_bound_matters n	Rel n → (" A recursive reference to the i-th body ")	Case (ci, u, pms, ret, iv, c_0, br) \rightarrow (" iv ignored: it's just a ranks ")	rx bodiex	bodynum >= nbfix then anomaly (Pp.str "Ill-formed fix term.");
let eq_wf_paths = Rtree.equal Declareops.eq_recarg	elp.eind_recergs	Ind ind kn → (* When the inferred tree allows it, we consider that we have a		end → Not subtern	if Int.equal n (ntodies * k = i) then List.fold left i (fun numiformarams a →	let (ci, $(p, _)$, $_iv$, c_0 , $bra)$ = expand_case renv.env (ci, u, per ret. iv , c 0, br) in	, Ind _ Construct _ →	
let inter_recarg r1 r2 = if eq_recarg r1 r2 then Some r1 else None	let match_inductive ind re -	potential	et rec subters_specif Tevers rem stack t -	let met_intm_specif or spec -	match kind a with Nai a when Int.equal a (k - j) ->	let ra - check inert subtem rec call renv rs c 8 in let rs - check inert subtem rec call renv rs p in	CHECK, FRE, CALL, STATE FROM ADMINISTRACE STATES FIX (FUR () -> NOTE)	let vdefj = judgment_ef_fispoint recdef in let raiss_err env i err = error_ill_formed_rec_body env (Type_errors_fisbumentFror err) names i
let inter_wf_paths • Rtree.inter Declareops.eq_recarg inter_recarg Norec	Mrsc (RecArgind i) → Ind.CanOrd.equal ind i	begin match dest_recarg tree with	let f,1 = decompose_app_list (whd_all ?evers renv.env t) in	lary (match Lary, force spec with	(* a reference to the j-th parameter *)			
let incl_wf_paths = Rtree.incl Declareops.eq_recarg inter_recarg Norec	Norec Mrec (ReckrgPrim _) -> false	New (Neckryled ind') when (End.equal env (fat ind.kn) ind' -> build_recarge_nested ienv tree (ind.kn, large) Norec New> sk_morec	match kind f with Rel k → subterm_var k renv	Not_subters -> if or >= 1 then Internally_bound_subters (Int.Set_singleton or) also Not_subters	nuniforeparans	let rs' = NoNeedNeduce:rs in let rs = redex_level rs' in	<pre>let meedreduce', rs = check_rec_call renv rs c in check_rec_call_state renv meedreduce' stack rs (fun () →</pre>	(* Check the i-th definition with recorg k *) let find ind i k def =
let spec of tree t =	(* In {match c as z in ci y_s return P with C_i x_s ⇒ t end) [branches_specif renv c_spec ci] returns an array of x_s specs knowing	Norec Nec> ek_morec	Case (ci, u, pes, p, iv, c, lbr) - (* iv ignored: it's just a cache *) let (ci, (p,), iv, c, lbr) - expand_case renv.env (ci, u, pes, p, iv,	spec -> spec)	(* not a parameter: this puts a bound on the size of an extrudable prefix of uniform arguments *)	<pre>let case_spec = branches_specif renv (set_iots_specif nr (lary_subters_specif ?</pre>	(" we try hard to reduce the proj sway by looking for a constructor in c (we unfold definitions too) ")	(* check fi does not appear in the k-1 first abstractions, gives the type of the k-1-eme abstraction (must be an inductive) *)
if eq.wf_paths t sk_norec	r mar (f)	Const (c,_) when is primitive_positive_container env c -> c	, lbr) in let stack' = push_stack_closures resv 1 stack in	(**************************************	min j muniformparame) 8 muniformparames 1	evers renv [] c_0)) ci in let stack' = filter_stack_domain ?evers renv.env nr p stack in let rs' =	let c = whd_all Tevers renv.env c in	let rec check occur env n def *
them Not_subterm else Subterm (Int.Set.empty, Strict, t)	let brancher specif rem c_spec ci = let car = (^* YI: an array of ints, arities of constructors of C *) (* We fetch the regular tree associated to the inductive of the match.	begin match dest_recarg tree with New (Neckspfris c') when (Constant.equal env c c' → build_recarge_mented_primitive ienv tree (c, large)	let cases apec = branches apecif renv (lazy_subters_specif Tevers renv [] c) ci	exception fixSuardError of env * fix_guard_error	nuniforeparams - + fold_constr_with_binders succ (aux i) k nuniforeparams c	let rs' - Array.fold_left_i (fun k rs' br' →	<pre>let hd, args = decompose_mpp c in let hd, args = match kind hd with Cofix cofix →</pre>	match kind (whd.sll ?evers erw def) with Lambda (x.s.b) -> if roccur_with_mets n rbfix a then
Set merge_internal_mubterms 11 12 - Int.Set.union 11 12	This is just to get the number of constructors (and constructor arities) that fit the match branches without forcing a mac.	None Free -> ek_morec	in	let illegal_rec_call new fx = function	in Arraw.fold left i (fun i -> mux i 0) min inde bodies	let stack br = push_stack_args case_spec.(k) stack' in check rec call stack rew stack br rs' br') rs' brs in		let env' - push rel (LocalAssum (x.a)) env in
	Note that c_spec might be more precise than [v] below, because of	end _arr →	Array most (fun i hr' six	Singre (_arg_rev,_arg) -> let le_lt_wrs - lary (let (_,le_wrs,lt_wrs) -		let needreduce_br, rs = List.sep_first rs' in	→ hd, args in	<pre>if Int.equal n (k = 1) then (* get the inductive type of the fixpoint *)</pre>
let inter_spec s1 s2 = match s1, s2 with	matted inductive types. *) (* let (_,mip) = lookup,mid_specif renv.env ci.ci_ind in *) (* let v = dest_mobterss mip_mind_recorps in *)	sk_norsc	let stack br = push_stack_args (cases_spec.(i)) stack' in subterm_specif ?evers_renv_stack_br_br')		("* Disen a fispoint [fix f x y z n := phi(f x y u t,, f x y u' t')] structural recursive on [n],	check_rec_call_state renv (needreduce_br needreduce_c_0) stack rs (fun () \Rightarrow	Construct -> Some (args.(Projection.moars p * Projection.arg	let (mind, _) = try find_inductive ?evers env a
_, Dand_code → x1 Dand_code, _ → x2	(* let v = dest_mabtersm sdp.sind_recorps in *) let v = dest_mabtersm (lookup_mabtersm renv.env ci.ci_ind) in Array.map List.length v in	and build_recorgs_mented (env,_rs_env as ienv) tree (((eind,i),u), largs) = (" If the inferred tree already disallows recursion, no need to go further	he in let mpc - mbterm_mpc_glb stl in restrict_mpcc Tewars renv.env spcc p	(fun (i,le,lt) mbt → match Lazy.force mbt with	with [x] not uniform we build in context [x:A, y:B(x), x:C(x,y)] a term [fix f z n := phi(f u t ,, f u ' t ')], may [pni], of some type [formil (x:C(x,y)) (n:I(x,y,z)), $I(x,y,z,n)$], so that	(" we try hard to reduce the match away by looking for a constructor in c.0 (we unfold definitions too) ")	p), []) Cofix Ind Lambda Prod Latin	with Not_found -> raise_err env i (NecursionNotUnInductiveType a) in
Not_matern, _ + x1 _, Not_matern + x2	Array.map List.length v in (** Y.l: for each constructor, generates a list of size=[sca]. *)	") if eq.wf_paths tree ek_norec then tree	restrict_spec Tevars renv.env spec p	(Subters(_,Strict,_) Dend_code) → (i*1, le, i::lt) (Subters(_,Large,_)) → (i*1, i::le, lt)	[forall $(z:C(x,y))$ $(n:I(x,y,z))$, $I(x,y,z,n)$], so that [fun x y z \Rightarrow pxi z] is of same type as the original term *)	let c 8 = whd all Tevars renv.env c 8 in let hd, args = decompose_app_list c 8 in	Cofix Ind Lambda Prod Latin Sert Int Float Array ** assert false Del West Description Comp. Fire	let mib lookup mind specif onv (out punion mind) in if mib.mind finite + finite then
	(** YJ: if discriminant spec [c_spec] is the same inductive as *)	else let mib = Environ.lookup mind mind mny in	fix ((recindes,i),(_,typerray,bodies as recdef)) →	→ (i•1, le .lt))	Let don uniform consequent moniformerane budies a	let hd, args = match kind hd with Cofix cofix →	Rel _ Nar _ Coret _ App _ Case _ Fix _ Proj _ Cast _ Mota _ Evar _ → None)	raise err env i (RecursionNotOnInductiveType a):
Internally_board_mbterm (mergs_internal_mbterms 11 12) Subterm (11,s1,t1), Internally_board_mbterm 12 → Subterm (mergs_internal_mbterms 11 12,s1,t1)	(fun i nca → (* i=1-th estructor has arity nca *)	let summar - mib.mind_mparams_rec in	(* when proving that the fispoint f(x)-e is less than n, it is enough to prove that e is less than n assuming f is less than n	(1,[],[]) renv.genv in (le_vars,lt_vars)) in	let mbodies - Array, length bodies in	decompose_app_list (whd_all Tevers renv.env (Term.applist	603	(mind, (mov', b)) else check_occur env' (n+1) b
	let lyrs : spec Lary.t = lary (match Lary.force c_spec with	let non-ecpar = mib.mind_mparams = aummpar in let (lpar,_) = List.chop aumpar large in	furthermore when f is applied to a term which is strictly less than n, one may assume that x itself is strictly less than n	RecursionOnlllegalTerm(fx,(arg_renv.env, arg),le_lt_vars) SArg _ →	let rec max i k c = let f, 1 = decompose_app_list c in	(contract_cofix cofix, args)))		<pre>else anomaly -label:"check_one_fix" (Pp.str "Bad occurrence of recursive call.")</pre>
<pre>(merge_internal_midsems 11 12,s2,t2) Subtems (11,s1,t1), Subtems (12,s2,t2) → Subtems (merge_internal_midsems 11 12, mire_glb a1 s2, inter_wf_paths t1</pre>	Subterm (_,_,t) when match_inductive ci.ci_ind (dest_recarg t)	let awartyp = mib.mind.ntypes in (* Extends the environment with a variable corresponding to	') if not (check_inductive_codomain Tevars renv.env typarray.(i)) then	(* Typically the case of a recursive call encapsulated under a rewriting before been applied to the parameter of a constructor *) Notfromph@rgmentsforfucfall fx	match kind f with	match kind hd with Construct cstr → Some (apply_branch cstr args ci brs, [])	let open! Contest.Named.Declaration in match lookup_named id renv.env with	_ → raise_err env i Notfnough/batractionInfisBody in
Subters (merge_internal_subterss 11 12, size_glb s1 s2, inter_wf_paths t1 t2)	let vrs = Array.of_list (dest_subterms t).(i) in assert (int.equal ncs (Array.length vrs)):	the inductive def ") let (env', as ienv') = ienv push inductive Swarz ienv ((mind.u).lpar) in	ot_subters size		(* A recursive reference to the i-th body *) if Int.equal n (rbodies * k = i) then	Cofix Ind Lambda Prod Letin	Localization> None Localizat (.c.) -> Some (c. []))	let ((ind, u), _) as res = check_occur fixenv 1 def in let . min = lookup mind specif env ind in
let solders over old a	Array.map spec_of_tree vrs	(* Parameters expressed in env' *) let lpar' = List.map (lift accetyp) lpar in	let (ctxt,clfix) = whd_decompose_prod Tevars renv.env typarray.(i) in	let set_meed_reduce_one env nr err rs =	<pre>let new_args = List.skipn_at_best numiforeparams 1 in Yerm.applist (f, new_args)</pre>	Rel _ Yer _ Coret _ App _ Case _ Fix _ Proj _ Cast _ Meta _ Ever _ + Nore)	Listin (s e t b) us	(* recursive sprop means non record with projections -> squashed *)
<pre>let subterm_spec_glb = Array.fold_left inter_spec Dead_code</pre>	Dead_code = Array_make ros Gend_code Dead_code = Array_make ros Gend_code Internally_bound_makeros _ as x \rightarray_make ros x Subtern _ Not_maker \rightarray_make ros Not_makero) in	(* In case of mutual inductive types, we use the recarge tree which was computed statically. This is fine because rested inductive types with	<pre>let oind = let env' = push_rel_context ctxt renv.env in try Some(fat (find_inductive Tewars env' clfix))</pre>	let er = List.length re in let rel, rel = List.chep (er-or) re in let _, rel = List.sep_first rel in	else	(* Enables to traverse fixpoint definitions in a more intelligent	Letin (x,c,t,b) => let nedreduce_c, rs = check_rsc_call renv rs c in let needreduce_t, rs = check_rsc_call renv rs t in	if Environ.is_type_in_type env (GlobRef.IndRef ind) then () else match relevance_of_ind_body mip u with
type guard_env =	List.init nos (fun j -> lary (Lazy.force lvrs).(j)))	mutually recursive containers are not supported. *)	with Set Franci as Sons in	rs1 @ NeedReduce (env, err) :: rs2	_ → map_with_binders mucc (mux i) k c	way, is, the rule : if - g = fix g (y1:11)(yp:1p) (struct yp) :- s &	begin match needreduce of stack stack needreduce c needreduce t	Serts, Irrelevant Serts, Relevance/or as ried ->
type guaro_env = { env : rev; (" dit of last fixpoint ")	car	if Int.equal monthyp 1 then [dest_subterms tree]	(match oind with None → Not_mabters (* happens if fix is polymorphic *)	let set_meed_reduce env 1 err rs -	in Array.mapi (fun i → aux i 0) bodies	 f is guarded with respect to the set of pattern variables S 	with	<pre>if not (Sorts.relevance_equal names.(i).Context.binder_relevance rind)</pre>
(" dS of variables denoting subterms ")	(** YJ: checks if the codomain of predicate [p] is an inductive by wh- unerapping. *)	<pre>also Array.map (fun mip -> dest_subterms mip.mind_recargs) mib.mind_packets</pre>	Some (ind, _) → let obfix = Array.length typarray in	Int.Set.fold (fun n -> set_need_reduce_one env n err) 1 rs	let filter_fix_stack_domain or decrarg stack muniformparams =	in al se & - f is guarded with respect to the set of pattern variables S	NoticedReduce -> (* Stack do not require to beta-reduce; let's look if the	then raise_err env i fixpointDelrelevantInductive Sorts_Relevant -> ()
genv : subterm_spec Lazy.t list;	let check inductive_codomain Tevars env p = let abschz, ar = whd_decompose_lambds_decls Tevars env p in let env = push_rel_context abschx env in	in let sk_inecarge j mip =	let recargs = lookup_xubterms renv.env ind in (" pushing the fixpoints ")	let set_med_reduce_top erw err rs = set_med_reduce_one env (List.length rs) err rs	let rec mux i numiformparams stack = match stack with	in [] Tp & - ap is a sub-term of the formal argument of f & - f is guarded with respect to the set of pattern variables	body of the let needs ") let spec = lazy_subters_specif Tevers renv [] c in	in res
let sale reny eny recarp tree -	let env = push_rel_context absctx env in let arctx, s = whd decempose arcd decls ?evers env ar in	(* The nested inductive type with parameters removed *) let auxloyect * abstract mind ic summing auxnor mind min.mind of ic in	let ren' = push_fix_renv renv recdef in let renv' =	tipe check subtern result	I I at II		let stack = lift1_stack stack in check rec call stack (sush let reny (s.c.t.spec)) stack rs b	in (* Do it on every firmint *)
f env a env	let env = push_rel_context arctx env in let i,_l' = decempose_app (whd_all ?evers env s) in	but within a favor word	(* Why Strict here ? To be general, it could also be	Invalidables Needleducalables of Int.Set.t (* entry - NoNeedleduce *)	let uniform, nuniformparams - if nuniformparams - 8 then false, 8 else true, nuniformparams -1 in	then f is guarded with respect to S in (g s1 am). Eduardo 7/9/98 *)	NeedReduce> check_rec_call_stack renv stack rs (subst1 c b)	let rv = Array.msp2_i find_ind reset bodies in (Array.msp fst rv, Array.msp and rv)
rel_min = recarg-2; (* recarg = 0 \Rightarrow Rel 1 \Rightarrow recarg; Rel 2 \Rightarrow fix *) genv = [tany.from_val(Subtare(Int.Set.empty, Large,tree))] }	isled i	(fun k c → let c' = hef_prod_applixt Tevarx env' c lpar' in	Large") assign_var_spec renv'		let a -	Fix ((recindxs,i),(_,typarray,bodies as recdef) as fix) →	603	(Array.map rat rv, Array.map and rv)
let push_var renv (x,ty,spec) -	(** YJ : I guess [ierv] meses inductive environment, and this is code to suild wf oaths. *)	("skip non-recursive parameters") let (ienv',c') = ienv_deccepose_prod ?evarx ienv' nonrecpar c' in build_recarge_constructors ienv' trees.(j).(k) c')	<pre>(ebfix-i, bary (Subters(Int.Set.empty,Strict,recarge))) in let decrarg - recindex.(1) in let theBody - bodies.(1) in let rbBfAbat - decrarge1 in</pre>	(* Check term c can be applied to one of the mutual fixpoints. *) let check_ix_mubters x tree =	if uniform Int.equal i decrarg them a else	let decrArg = recindum.(i) in let mbodies = Array.length bodies in	Cast (c, , t) -> let rs = check_inert_materm_rec_call remv rs t in let rs = check_rec_call_stack_remv stack_rs c in	let check_fix ?evers env ((nvect,_),(names,_,bodies ax recdef) ax fix) =
[pro a rush red [[oralization (v to]] reno error		build_recergs_constructors ienv' trees.(j).(k) c')	let theBody = bodies.(i) in	let check in matter x tree = match Lazy.force x with Subtern (need_reduce,Strict,tree') →	(* descrivate the status of non-uniform parameters since we cannot guarantee that they are preserve in the recursive	let nbodies = Array.length bodies in let rs' = Array.fold_left (check_inert_subterm_rec_call renv) (NoMesSkeduce::rs) typerray in	let rs = check_rec_call_stack renv stack rs c in rs	Let check_fix Tevers env ((nvect,_),(names,_,bodies ax recoef) ax fix) = let (minds, rdef) = inductive_of_mutfix Tevers env fix in let flags = Environ.typing_flags env in
rel_min = renv.rel_min=1;	(* The following functions are almost duplicated from indtypes.al, except	auxlevect						
cel_min = renv.rel_min=1; genv = mpec:: renv.genv }	(* The following functions are almost duplicated from indtypes.ml, except that they carry here a poorer environment (containing less information). *) let isev comb var (em. 17m) (x.a.c.) :	in	let sign, strippedBody = whd_decompose_lambda_n_assum Yevers renv.env bDfAbst theBody in	else Invalidabters	calls ") SArm (set jots specif or (lary Not subters)) in	let rens' = push_fix_renv renv rectef in let numiformograms = find uniform oprameters recindum (List_lengt		let get tree (kn.i) =
genv = spec:: renv.genv)	(* The following functions are almost duplicated from indtypes.el, except that they carry here a poorer environment (containing less information). *)	in sk_paths (Wroc (RecArglind (mind,j))) paths in	Let sign, troppeddory = vnd_occeptose_immode_n_assim revers renv.env bDfAbst theBody in	also InvalidSubters Deed rote at NeedSubters Let Set among	calls ") Skeg (set_ists_specif or (lary Not_subters)) in a :: aux (i) numiforepares stack	let numiformparams - find_uniform_parameters recindus (List.lengti stack) bodies in		let get_tree (km,i) = let mib = Environ.lookup mind kn env in
gene = spec: rew.gene ; let push_let renv (s,c,ty,spec) = { enc = push_rel (t,c,ty,spec) = rel min = rev.v.rel min-!; rel min = rev.v.rel min-!;	(* The following functions are almost deplicated from intippes.al, except that they carry here a pourse endorment (containing less information). *) let imm.push.vov (env., ira) (x,x,x_0) = (push.red (iccatismum (x,x_0)) env., (torce,re)::Ira) let imm.red iractive house (env., re, env) (finindum).harr =	in	Let sign, strappedbody - wno_oscompose_lambde_n_assum revers renv.env bDFAbst theBody in	else Invalidabters	a :: aum (i+1) muniformparams stack in aum @ numiformparams stack	let remi" = pun_riz_rem vemv recom in let numiformparams = find_uniform_parameters recindum (List.lengt) stack) bodiem in let bodiem = drop_uniform_parameters numiformparame bodiem in let fin stack = filter fin stack domain (redem level rs) decrArm	h Sert _ Int _ Fleet _ → (* See [Prod]: we cannot ensure that the stack is empty *) rs	In Trags.Conce, guarone train but put true (bn.1) = let süb = Environ.lookup.sind kn env in süb.sünd.packets.(1).sünd_recarge in
gate = space; rem.pare; fam = pash red (consider (p.c.,ty)) rem.eme; red_star = rem.red_star(; rem.rem.pare; rem.pare;	(* The following functions was about duplicated from intipmental, except that they carry here a papere environment (unstaining laws information). *) let inter, push, you (mor, you) (x,x,yo.) ** (push,yoi (iccaliums (x,a)) waw, (inter,x,n)::ira) let inter, push, informit before (mor, push, you (iccaliums (x,a)) way. I let inter (more information forms from you, re, gam) ((mind,yo), ipar) - let inter forms in best of whome is the other as the field where it is the three as it has distributed in the field.	in sk_paths (Free (Backegind (mins,j))) paths in the company of th	Let tagh, trappendory " who occupous_lambos_n_assum revers renv.env bDfAbet theBody in (" pushing the fix parameters ") Let stack' " push_stack_closures renv l stack in	if not of plant one tree the measurements need reach leading leads to Residentialities and the same of	a :: aux (4:1) nuniforsparams stack in aux 8 nuniforsparams stack lat pp_argument. Towars medicular new all stack x a b = marks meantendown all twill.	let ment "plann, tal. fann renk recomer an let ment "plann, tal. fann renk recomer an let hodies - drop uniform, parameters soniformparame thodies let bodies - drop uniform, parameters soniformparame bodies in let fla. tabet - filter fla. tabet, domain (renks level re) derenk tabet antiformparame in a filt is bomb bathe despetent best let frante.	Sert_ Int_ Flast_ + (* See [Fred]: we cannot ensure that the stack in ampty *) rs Army (_m, t, dr, t, y) - (* In [Fred]: we cannot ensure that the stack in ampty *)	If Yagu.check_garein twon let ppt_tree (pn,1) = let min = fretron_inolep_mind kn enw in min.min.pubcet.i().mind_precapy in trees = frety.map (fun (mind_p) = gpt_tree mind) minds in for 1 = 8 in frete health before 1. 1 do:
gene = spec: rew.gene ; let push_let renv (s,c,ty,spec) = { enc = push_rel (t,c,ty,spec) = rel min = rev.v.rel min-!; rel min = rev.v.rel min-!;	(* the following functions are almost deplicated from independal, smooth that they carry here a purere enricement (containing lines information). *) lest inser, gant, wor (new, 'ze) ('z., ze') . (gant, red (containing (ze)) new, ('tever, re)::iris) lest inser, pand, insertates brane (new, re, new) ((mind, ne), pare) : let in the "notion closely paties sind one is let delignes with a bold of departs into response and part of the containing into the response in the containing into the response in the response and forthis part of the response in the response and forthis part of the response in the response and forthis part of the response in the response and red part of the response in the response and red part of the response in the response	in myster (track (teckrylist (sint,)))) paths in the companion of the firecept - Array, mapt at Irecepts sth. sind. packets in (three map, re. trencys). (1) and built - (recept, rest of partial firecept, rest of partial firecept, rest of partial firecept, rest of the firecept, rest of partial firecept, rest of the firecept firecept firecept, rest of the firecept firecep	the taghtropomony was personal passar nears near near near near near near near near	If the Lyt place the tree was assumed an enconcernment many removal and bond indicated and the first place of the Removal and	s :: mur (4-1) numiformparams stack in mur @ runiformparams stack Let too aroument Tevers needreduce renv elt stack x s b =	In rest " part, right men incern tack) bett med fordpragness "fold, undere presenter recindes (tist.length tack) bett med fordpragness. Fold qualiform, presenter recindes (tist.length tack) better in the second fordpragness better in let first, parts "filter first, tack, dense (redex_lendt_r)" dearling tack mediformprass in let first, tack it filt.length tack of dearling the dearling that the dearly the tist.first. (dearling) "first, tack it is let tack, tack it is let tack, the "lift, tack it of the second first, tack it.	is Sert _ Int _ Flant _ → (* See [Fred]: we convolument that the stack is empty *) rs Mrmy (_x,t_def,ty) → (* See [Fred]: we convolument that the stack is empty *) but rs = Army_fold_left (Gence_inset_molter=,rcc_mall rew) rs t is	if risp(context_glueres to me) let up (true (ps,1) = let up (true landsp,sind be new in sub-air (pstrine, landsp,sind be new in sub-air (pstrine, lo_sind_recepy let true = true, new (fun (stad_n) -> pst_true sind) sinds in let (rew = hard, new (fun (stad_n) -> pst_true sind) sinds in let (rew, land) - risk ((l) in let rew = name, new form most(1)) true.(i) in
pure *part; retrujent y bar part; retrujent y en partyral (monible (se, tyl) retrusen; rajain - retrusen, part) pure *part; retrujent y the stanging retrusen (spart) (retrusen) pure * (stanging retrusen (spart) (retrusen pure v (stanging retrusen (spart) term with pure * (stanging retrusen (spart) term with pure * (spart) term with pure v (spart)	(* In failuring frontiers we short applicated from integrand, sough that they corred to a power contensed (continuity less infraredition). *) (pair, yell (conditions (x, x)) (contens, yell-red) (pair, yell (conditions (x, x)) (we, (we, x, y)) (rize) integrand (conditions (x, x)) (we (we, x, y)) (rize) integrand (conditions) (we (we (we (we (w, x)))) (we (we (w, x))) (in finger = with anticipates in the stage = with anticipates in the conditions (we (we (we (w, x)))) (we (we (w, x))) (in time = bent conditions) (we (we (w, x))) (we (we (w, x))) (in time = bent conditions) (we (we (w, x))) (we (we (w, x))) (in time = bent conditions) (we (we (w, x))) (we (we (w, x))) (we (w, x)) (in time = bent conditions) (we (we (w, x))) (we (w, x)) (we (w,	in a jobs (Proc (blockydid (sick,)))) pubbs a substitution of the strongs a ferranges derivancy and hold position in that forces a feet forces after a ferrange (i) and halfs company (a) and halfs company (a) feet (sick, a) and halfs company (a) feet (sick, a) and halfs company (a) and halfs company (a) and a feet (sick, a) and (sick, a)	the tags.trapements was personal season, asset were reversed. And tags.trapements was presented in the tags. Let tack "push tack, classes rever lates in let rever" push, tack "endificat then rever" also let rever me tack dement meet Thera teacher in let rever meet "tack dement meet Thera teacher in	If the problem the communications was processed and institutions of the communications and the communication of the communications and the communication of	a :: no (**) matterpress stack in no # of the property of the set o	is read - year, year one format - read - year of the year of the read - year of the read	Sert_ Int_ Flast_ + (* See [Fred]: we cannot ensure that the stack in ampty *) rs Army (_m, t, dr, t, y) - (* In [Fred]: we cannot ensure that the stack in ampty *)	or long-reads, garden term into the third term long-read to see in which products (1), sind, rearrys and term a term page (reg (seed,) as gattern size) for 1 = 8 to seep, long (seed,) as gattern size) sinds in for 1 = 8 to seep, long the size = 1 to last (seep, about) - reads (1) in last rear - main, rear form most (1) themme. (1) in ty chack, out, if I however now never them toby
que « que recupe y la que que recupe y la que que recupe y que en que « que recupe y que en que « que recupe») let misque « que recupe») let misque « que recupe») let misque « que recupe» (que) » let misque « que recupe» (que » que « que » que « que » que	(" in this disary functions are about adjustent from indigental, amough the flow query lives a given evidenteed (modified) as information, ") let inter, gain, we fair, "by (1, s, s, s) = (mar, s) (lines, s), which is the second of the secon	in	of the transfer of the transfe	It has provided by the constraints of the constrain	a :: no (**) matterpress stack in no # of the property of the set o	is now 'page, 'page 'now 'now 'now 'now 'now 'now 'now 'now	Sint I list I list I lists and the state is empty ") as given more than the state is empty ") as given the more than the state is empty ") as provided in the state is empty "). It was provided in the state is empty ") and in a form, foliable from the state is empty ") and in a form, foliable from the state is empty. The state is empty and in a form, foliable from the state is empty and in a form, foliable from the state is empty and in the state is emp	of Trady, Contact, Specials: Used in the State
gas vagues rescapes; if an exploit resc (exploit) (see - page, discalable (exploit) resc.ess; (see - page, discalable (exploit) resc.ess; identifying a resc (exploit) list mainty a resc resc (exploit) list mainty a resc resc (exploit) list mainty are part (exploit) list mainty are part (exploit) list mainty are part (exploit) list mainty are resc.ess; list ma	(* The Ministry functions are about adjustment from indigmental, amough the time query two are non-indigment (indigment), amough the time query two are non-indigment (indigment), also indirectable, ") (and, not, (including ()) are, (including ()) are, (including ()) are, (including ()) are in the foreign contained and are in the foreign contained an	in a jobs (Proc (blockydid (sick,)))) pubbs a substitution of the strongs a ferranges derivancy and hold position in that forces a feet forces after a ferrange (i) and halfs company (a) and halfs company (a) feet (sick, a) and halfs company (a) feet (sick, a) and halfs company (a) and halfs company (a) and a feet (sick, a) and (sick, a)	of the stage of th	If the production we can be associated to the product and benefit of the control	a row (cf) and frequency stack is as k row (cf) and frequency stack in any fraction proposed. Now to make the core with take $k = k$ is such manifolding, which contains the contains thas the contains the contains the contains the contains the contai	is read - year, year one format - read - year of the year of the read - year of the read	1 Set, [Set, [Set,]] Find, a " ("in Second is execute search that the stant is septy") "I server, [La, [La, La, La, La, La, La, La, La, La, La,	or long-reads, gardent com- int case - reads -
gas vagues rescapes; if an exploit resc (exploit) (see - page, discalable (exploit) resc.ess; (see - page, discalable (exploit) resc.ess; identifying a resc (exploit) list mainty a resc resc (exploit) list mainty a resc resc (exploit) list mainty are part (exploit) list mainty are part (exploit) list mainty are part (exploit) list mainty are resc.ess; list ma	(* The Ministry functions are about adjustment from indigmental, amough the time query two are non-indigment (indigment), amough the time query two are non-indigment (indigment), also indirectable, ") (and, not, (including ()) are, (including ()) are, (including ()) are, (including ()) are in the foreign contained and are in the foreign contained an	in graphic (Vive (Uniteplie) (size, (J)) paths let irrange - bray and it, jurange sit hair jurkits in let irrange - bray and it, jurange sit hair jurkits in let irrange - bray and it, jurange sit hair jurkits in let irrange - bray and it. and hair jurkits - bray and it. and hair jurkits - bray and it. and party - let it. ("Viriation ignore matched by nor-matched induction let it. and in space - littles ((inc.)) = c./pharmill regard ((i)) pages in let ippe - littles ((inc.)) = c./pharmill regard ((i)) pages in let ippe - littles ((i)) (bray inc.) (inc.) (inc.) (inc.) (inc.) (inc.) let ippe - littles ((i)) (bray inc.) (inc.) (inc.) (inc.) (inc.) (inc.) (inc.) let ippe - littles ((i)) (bray inc.) (inc.) (inc.) (inc.) (inc.) (inc.) let impe - littles ((i)) (bray inc.) (inc.) (inc.) (inc.) (inc.) (inc.) let impe - littles ((i)) (bray inc.) (inc.) (inc.) (inc.) (inc.) (in.) let impe - littles ((i)) (bray inc.) (in.) (in.) (in.) (in.) let impe - littles ((i)) (in.)	of the stage of th	It that, y just to the true is assumed to the supplier of the property of the	a :: no (**) matterpress stack in no # of the property of the set o	is not fragment. For any other properties of the contract of t	Sint I list I list I lists and the state is empty ") as given more than the state is empty ") as given the more than the state is empty ") as provided in the state is empty "). It was provided in the state is empty ") and in a form, foliable from the state is empty ") and in a form, foliable from the state is empty. The state is empty and in a form, foliable from the state is empty and in a form, foliable from the state is empty and in the state is emp	of Tagle_Mack_games are in the first of the
gain o figure (section) for the first blue may (Long time) real part of the first blue may (Long time) real part of the first blue first blu	(* The Millering functions are about adjustment from indigmental, amough the time query two are non-indicated functionally laws of the function of the contract of the contrac	in	To high of commons of war, accommon among a man man's reason and a state of the common among a man and a state of the common among a man and a state of the common among a	and in-distribution. The Barry was a second or before the second or bef	at 1 and (cc) conformation which is not found by the proposed for the second state of a seal to propose the two second state of a set is a found by the proposed for the second state of	In anti-represe. Find address, among marine (Inst. address, among marine)).	1 Inst., 1 inc., 1 fanc., 2 or 1 fanc., 2 or 1 fanc., 2 or 1 fanc. 2 or 2 o	of Tragit-Long Lipsonia town in the Silve Tragit-Long Lipsonia town in which Tragit-London Lipsonia Lipsonia town in which Lipsonia Lipson
gain vigorii veriginii ; (astronii error (c.c.)c.; (astronii error (c	(* The Millings functions are about algorithm from indigmental, amount that the query laws are non-indigment functionally also information, **) (making laws information, **) (making lams information), **)	on grates (Nove (Univerplan (sinte, ())) paths on grates (Nove (Univerplan (sinte, ())) paths in the property - brough and property shadow, grates in the Markovan Arm. Interplan (See See See See See See See See See Se	Section 1. In the common of th	It may a pain the two two managements and a sun- plements with the control of the	a) in a ((a)) maniferences units. b) and ((a)) maniferences units. b) and ((a)) maniferences units. b) and ((a)) maniferences. b) and ((a)) maniferences. b) and ((a)) maniferences. c) and ((a)) maniferences. d) and ((a)) manif	in distillations of the distillation of the di	Set 1, 1 for 1, 1 feat 1, 2 m. ("In (Profit) a count many that the stant is empty") If you (A_AAAA) = 1 Army (A_AAAA) = 1 Army (A_AAAA) = 1 Army (A_AAAA) = 1 Army (A_AAAAA) = 1 Army (A_AAAAA) = 1 Army (A_AAAAAA) = 1 Army (A_AAAAAA) = 1 Army (A_AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA	of Tragit-Long Lipsonia town in the Silve Tragit-Long Lipsonia town in which Tragit-London Lipsonia Lipsonia town in which Lipsonia Lipson
gain o figure (section) for the first blue may (Long time) real part of the first blue may (Long time) real part of the first blue first blu	(*) The Millings functions are about adjustment from designands, a rough (*) and (*) a	in	Settless should be a promision of the pr	and in-distribution. The Barry was a second or before the second or bef	at 1 and (cc) conformation which is not found by the proposed for the second state of a seal to propose the two second state of a set is a found by the proposed for the second state of	In anti-represe. Find address, among marine (Inst. address, among marine)).	Set 1, 1 for 1, 1 feat 1, 2 m. ("In (Profit) a count many that the stant is empty") If you (A_AAAA) = 1 Army (A_AAAA) = 1 Army (A_AAAA) = 1 Army (A_AAAA) = 1 Army (A_AAAAA) = 1 Army (A_AAAAA) = 1 Army (A_AAAAAA) = 1 Army (A_AAAAAA) = 1 Army (A_AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA	of Tragit-Long Lipsonia town in the Silve Tragit-Long Lipsonia town in which Tragit-London Lipsonia Lipsonia town in which Lipsonia Lipson

The Guard Checker of Coq

- · About 1,000 lines of **unspecified**, **unexplained** OCaml code
- Iterated by different authors over 30 years
- Multiple dimensions of complexity

Contribution

Two main contributions of this project:

Implementation

A full implementation of Coq's Guard Checker in Coq, using the MetaCoq project.

Extending previous work by Lennard Gäher [2].

Documentation

In the report: examples (Chapter 2, Appendix) and explanations (Chapter 3).

MetaCoq project

- Formalises Coq's type theory in Coq (faithful) [3]
- A verified implementation of type checker [4]
- A verified extraction function to OCaml [5]

Proved:

- Subject Reduction and Canonicity,
- parameterised by a guard checker
- assumed Normalisation

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.
(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | 0 => n
  S m' \Rightarrow add m' (S n)
  end.
MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.

(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

```
MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.
(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | 0 => n
  S m' \Rightarrow add m' (S n)
  end.
MetaCoq Quote add_syntax := add.
Check check fix.
Compute (check_fix add_syntax).
```

```
add syntax : Ast.term :=
  (Ast.tFix [{|
    dname := {| binder_name := nNamed "add" |};
    dtype := Ast.tProd { | binder_name := nNamed "m" | }
      (Ast.tInd { | inductive mind := "nat" | } [])
     (\ldots):
    dbody :=
      Ast.tLambda
       {| binder name := nNamed "m" |}
        (Ast.tInd {| inductive_mind := "nat"; inductive_ind := 0 |} [])
        (Ast.tLambda
          {| binder_name := nNamed "n" |}
          (Ast.tInd {...} [])
          (Ast.tCase
             {| ci_ind := {| inductive_mind := "nat" |}; |}
             {| Ast.pcontext := [{| binder_name := nNamed "m"; |}];
                 Ast.preturn := Ast.tInd {| inductive_mind := "nat" |} []
              1}
             (Ast.tRel 1)
             [ { | Ast.bcontext := []; Ast.bbody := Ast.tRel 0 | };
                {| Ast.bcontext := [{| binder_name := nNamed "m'"; |}];
                   Ast.bbody :=
                    Ast.tApp (Ast.tRel 3)
                       [Ast.tRel 0;
                        Ast.tApp
                          (Ast.tConstruct { | inductive_mind := "nat"; inductive_ind := 0 |})
                          [Ast.tRel 1]]
                 |}]));
          rarq := 0
      [{]
```

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.
(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

check_fix : Ast.term -> bool

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.
(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

= true : bool

History of the Guard Checker

Phase 1: Beginnings

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]

Phase 1: Beginnings

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]

Phase 1: Beginnings

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]

Phase 2: Specifications

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]
- · Codifying Recursive Definition with Recursive Schemes (Eduardo Gimenez, 1994) [8]
- Inductive Definitions for Type Theory (Christine Paulin-Mohring, 1996) [9]
- Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants (Eduardo Gimenez, 1996) [10]

Phase 3: Big Changes

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]
- · Codifying Recursive Definition with Recursive Schemes (Eduardo Gimenez, 1994) [8]
- Inductive Definitions for Type Theory (Christine Paulin-Mohring, 1996) [9]
- Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants (Eduardo Gimenez, 1996) [10]
- β - ι commutative cuts subterm rule (Pierre Boutillier, 2010) [11]

```
match v2 in with
| nil => (fun _ => nil C)
| cons h2 t2 => (fun t1' => cons (f h1 h2) (map2 f t1' t2))
end t1
```

Two Weeks before Christmas, 2013

```
From: Daniel Schepler <dschepler AT qmail.com>
To: Coq Club <coq-club AT inria.fr>
Subject: [Coq-Club] bijective function implies equal types is provably inconsistent with functional extensionality in
Coq
Date: Thu, 12 Dec 2013 11:02:00 -0800
Section bijective_impl_eq.
Hypothesis functional_extensionality :
  forall (A B:Type) (f q:A->B),
  (forall x:A, f x = q x) -> f = q.
. . .
Definition not_bijective_impl_eq : False := func_unit_discr unit_eq_False_False_funs.
End bijective_impl_eq.
Daniel Schepler
```

Phase 3: Big Changes

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]
- · Codifying Recursive Definition with Recursive Schemes (Eduardo Gimenez, 1994) [8]
- Inductive Definitions for Type Theory (Christine Paulin-Mohring, 1996) [9]
- Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants (Eduardo Gimenez, 1996) [10]
- β - ι commutative cuts subterm rule (Pierre Boutillier, 2010) [11]
- · Restore compatibility with Propositional Extensionality (Maxime Dénès, 2014) [12]
- Restore strong normalisation (Hugo Herbelin, 2022) [13]
- Extrude uniform parameters (Hugo Herbelin, 2024) [14]

Phase 3: Big Changes

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]
- · Codifying Recursive Definition with Recursive Schemes (Eduardo Gimenez, 1994) [8]
- Inductive Definitions for Type Theory (Christine Paulin-Mohring, 1996) [9]
- Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants (Eduardo Gimenez, 1996) [10]
- β - ι commutative cuts subterm rule (Pierre Boutillier, 2010) [11]
- · Restore compatibility with Propositional Extensionality (Maxime Dénès, 2014) [12]
- Restore strong normalisation (Hugo Herbelin, 2022) [13]
- Extrude uniform parameters (Hugo Herbelin, 2024) [14]

A Taste of the Guard Checker

Example: add

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Goal: check that add is guarded.

Guarded: *All* recursive calls have a **strict subterm** as the **recursive argument**.

Example: add

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Subterm Specification

With respect to the **recursive parameter** m, terms can be a

- Large Subterm (e.g. m)
- •
- •

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Subterm Specification

With respect to the **recursive parameter** m, terms can be a

- Large Subterm (e.g. m)
- Strict Subterm (e.g. m¹)

•

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Subterm Specification

With respect to the **recursive parameter** m, terms can be a

- Large Subterm (e.g. m)
- Strict Subterm (e.g. m¹)
- · Not Subterm (e.g. n)

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Guard Env : [n:Bound{1}|m:Large|add]

Guard Environment

Subterm specifications of terms in the local context are stored.

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Guard Env : [...] Stack : [Closure m'|Closure(S n)]

Stack of subterm specifications

The subterm information of arguments are stored on a stack when checking the head of an application.

```
Guard env: [m:Large|add]
Stack: []
```

Initial state. Parameters after the recursive parameter are turned into lambdas.

```
Guard env: [m:Large|add]
Stack: []
```

For a lambda to be guarded, its

- · binder type must be guarded, and
- · body must be guarded.

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | 0 => n
  | S m' => add m' (S n)
  end.

Guard env: [m:Large|add]
Stack: []
```

Binder type is guarded.

Guard env: [n:Bound{1}|m:Large|add]
Stack: []

The body is checked with a updated guard environment.

```
Guard env: [n:Bound{1}|m:Large|add]
```

Stack: []

For a match to be guarded,

- · discriminant,
- return type, and
- every branch

must be guarded.

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Guard env: [n:Bound{1}|m:Large|add]

Stack: []

Discriminant (m) and the return type (nat) are guarded.

```
Guard env: [n:Bound{1}|m:Large|add]
```

Stack: []

To check a branch:

- expand into a lambda
- specify parameters
- · check the lambda

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | 0 => n
  | S m' => add m' (S n)
  end.
```

Guard env: [n:Bound{1}|m:Large|add]

Stack: []

0-th branch has no parameter.

- expand into a lambda
- specify parameters
- · check the "lambda": guarded.

Stack:

[m':Strict]

```
Guard env: [n:Bound{1}|m:Large|add]
```

Stack: [m':Strict]

1-st branch:

- expand into a lambda
- specify parameters: strict!

•

```
Guard env: [n:Bound{1}|m:Large|add]
```

Stack: [m':Strict]

1-st branch:

- · expand into a lambda
- specify parameters: strict!
- · check the lambda

1-st branch:

- · expand into a lambda
- specify parameters: strict!
- · check the lambda

Application with the recursive call is guarded if

- arguments are all guarded, and
- key case: the recursive argument is a strict subterm (on the stack)

Arguments are checked from right to left: both guarded.

Stack is populated with closures.

Since the recursive parameter of add is at position 0, specify the 0-th element of the stack.

m' is a **strict** subterm according to the Guard Environment

Done!

Since the recursive parameter of add is at position 0, specify the 0-th element of the stack.

m' is a **strict** subterm according to the Guard Environment.

Done!

Since the recursive parameter of add is at position 0, specify the 0-th element of the stack.

m' is a **strict** subterm according to the Guard Environment.

Done!

What if...

Delayed? Answer: Stack handles this well.

```
Fixpoint add (m n : nat) {struct m} : nat :=
  (fun k => match k with
  | 0 => n
  | S m' => add m' (S n)
  end) m.
```

What if...

Obfuscated? Answer: weak-head reduction only when checking subterm specification.

```
Fixpoint add (m n : nat) {struct m} : nat :=
  (fun k => match (id k) with
  | 0 => n
  | S m' => add (pred (S m')) (S n)
  end) m.
```

```
What if...
```

Not guarded in erasable subterms?

Answer: strong normalisation (reduction only when needed).

```
Fixpoint add (m n : nat) {struct m} : nat :=
  let _ := add m (add m m) in
  (fun k => match (id k) with
  | 0 => n
  | S m' => add (pred (S m')) (S n)
  end) m.
```

What if...

Not guarded in erasable subterms?

Answer: strong normalisation (reduction only when needed).

```
Fixpoint add (m n : nat) {struct m} : nat :=
  let _ := add m (add m m) in
  (fun k => match (id k) with
  | 0 => n
  | S m' => add (id m') (S n)
  end) m.
```

Not covered in example: β - ι cuts, redex stack, nested fix, ...

The (at least) 4 Dimensions of Complexity

Dimensions of Complexity

- The stack of subterm specifications for β - ι commutative cuts
- Strong normalisation:
 - a redex stack
 - only reduce terms to weak-head normal form when needed
- Support for mutual and nested fixpoints
 - regular trees
- OCaml lazy for efficiency

Resulting in 1,000 lines of OCaml code.

Full Implementation in Coq

- · Complete, available as a MetaCoq (TemplateCoq) plugin.
- Feature parity with the kernel
- Test parity* with the kernel
- · Intentionally kept as close as possible to Coq's guard checker
- Available at: https://github.com/inria-cambium/m1-tan/tree/v1.0.0

Conclusion and Future Work

Summary

Conclusion

- implemented the Guard Checker in Coq
- documented its features
- · gave examples of its behaviour

Future Work

- verify that the guard checker itself is a terminating program
- specification of an abstract guard condition of the checker
- verify that the guard checker implements the guard condition
- relative consistency proofs for its soundness

Well-Founded Recursion

- An alternative to structural recursion
- · Coq: structural by default; well-founded using Program Fixpoint Or Equations

Lean: structural by default; well-founded attempted otherwise (termination_by)

Agda: structural by default; well-founded using Induction. WellFounded

Agda: Semantic Termination Checking

	Syntactic	Semantic
Example	Coq	Agda
Reduction	Minimal	Full
Mechanism	Guard	Sized Types
Advantage	Fast	Accurate

- Chan, Li, and Bowman [15] attempted Sized Types in Coq in 2019, compilation time increased as much as 5-15x on the Coq Standard Library.
- New algorithm in Agda by Nisht and Abel [16] is linear on input, but not yet proven complete.

Lean: Native Eliminators

- Lean is the opposite of Coq: eliminators are native in the kernel, recursive functions only exist in the surface syntax
- · Type Checking:
 - 1. Eliminators are generated for Inductive Types
 - 2. A strong (aka course-of-values) induction principle is defined using the said eliminators
 - 3. Recursive functions are translated into an encoding by the strong induction principle
- Extraction (Code Generation/Compilation) to C: the syntax gets extracted as-is
- Advantage: eliminators are simpler for the theory
 Disadvantage: hard to prove extraction correct, possible suprising behaviour

- F. Pfenning and C. Paulin-Mohring, "Inductively Defined Types in the Calculus of Constructions," in Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 April 1, 1989, Proceedings, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, Eds., in Lecture Notes in Computer Science, vol. 442. Springer, 1989, pp. 209–228. doi: 10.1007/BFB0040259.
- [2] L. Gäher, "Guard Checker in MetaCoq." GitHub, 2021.
- [3] M. Sozeau *et al.*, "The MetaCoq Project," *J. Autom. Reason.*, vol. 64, no. 5, pp. 947–999, 2020, doi: 10.1007/S10817-019-09540-0.
- M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter, "Coq Coq correct! verification of type checking and erasure for Coq, in Coq," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 1–28, 2020, doi: 10.1145/3371076.

- [5] Y. Forster, M. Sozeau, and N. Tabareau, "Verified Extraction from Coq to OCaml," Proc. ACM Program. Lang., vol. 8, no. PLDI, Jun. 2024, doi: 10.1145/3656379.
- [6] T. Coquand, "Pattern matching with dependent types," in *Informal proceedings* of Logical Frameworks, 1992, pp. 66–79.
- [7] C. Cornes *et al.*, "The Coq Proof Assistant-Reference Manual," *INRIA Rocquencourt* and ENS Lyon, version, vol. 5, 1996.
 - E. Giménez, "Codifying Guarded Definitions with Recursive Schemes," in Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June
- [8] 6-10, 1994, Selected Papers, P. Dybjer, B. Nordström, and J. M. Smith, Eds., in Lecture Notes in Computer Science, vol. 996. Springer, 1994, pp. 39–59. doi: 10.1007/3-540-60579-7_3.

- C. Paulin-Mohring, Définitions Inductives en Théorie des Types. (Inductive Definitions in Type Theory). 1996. [Online]. Available: https://tel.archivesouvertes.fr/tel-00431817
- [10] E. Giménez, "Un Calcul de Constructions Infinies et son application a la vérification de systemes communicants," 1996.
- P. Boutillier, "A relaxation of Coq's guard condition," in *JFLA Journées* [11] *Francophones des langages applicatifs 2012*, Feb. 2012, pp. 1–14. [Online]. Available: https://hal.science/hal-00651780
 - M. Dénès, "Tentative fix for the commutative cut subterm rule." [Online].
- [12] Available: https://github.com/coq/coq/commit/9b272a861bc3263c69b699cd2ac 40ab2606543fa
- [13] H. Herbelin, "Check guardedness of fixpoints also in erasable subterms." [Online]. Available: https://github.com/coq/coq/pull/15434

- [14] H. Herbelin, "Extrude uniform parameters of inner fixpoints in guard condition check." [Online]. Available: https://github.com/coq/coq/pull/17986
- [15] J. Chan, Y. Li, and W. J. Bowman, "Is sized typing for Coq practical?," J. Funct. Program., vol. 33, p. e1, 2023, doi: 10.1017/S0956796822000120.
 - K. Nisht and A. Abel, "Type-Based Termination Checking in Agda," 30th
- [16] International Conference on Types for Proofs, Programs, TYPES 2024, pp. 32–33, 2024. [Online]. Available: https://types2024.itu.dk/abstracts.pdf