Towards Formalising the Guard Checker of Coq

<u>Yee-Jian Tan</u>, Yannick Forster

14 September, 2024

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

Unrestricted fixpoints can be non-terminating...

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

and break consistency!

```
Check (boom 0). (* False *)
```

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!

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.
```

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 = renv.rel minen:	match kind c' with	let () = accort (Lint.is emoty large) in	(match mult with	elt :: stack' ->			check rec call state reny needreduce fix non absorbed stack rs (fun	match stack with
(* Check if t is a subterm of Rel n, and given its specification,	rel_min = renv.rel_min=n; genv = iterate (fun ge -> lary Not_mubters::ge) n renv.genv }	Prod(ra.a.b) →	<pre>let () = assert (List.is.empty large) in let recarg = build_recarge ienv (List.bd trees) b in let ienv = ienv_pub_var ienv (ra_b_ak_norec) in</pre>	(match mubt with Subterm (_, _s, wf) →	(** YJ: run below until stack is empty. First who [ar] into [t], *)		() →		elt :: stack →
assuming let already gives index of subteres with corresponding specifications of recursive arguments *)	let push_fix_renv renv (_,v,_ as recdef) =	let isno' = isno_push_var isno (na,a,sk_norec) in isno_decompose_prod ?evers isno' (n-1) b	recargs_constr_rec ienv' (List.tl trees) (recarg::lrec) d	(" We take the subterm space of the constructor of the record ") let wf_args = (dest_subterms wf).(8) in	let t = whd_mll Tevers env ar in match kind t with	given [recpos], the decreasing arguments of each mutually defined fispoint. *)		(" we try hard to reduce the fix away by looking for a constructor in [decrârg] (we unfold definitions too) ")	let rs = check_inert_subterm_rec_call renv rs a in let renv', stack', body' = pop_argument NoWeedReduce renv elt
(* A powerful notion of subterm *)	let n = Array.length v in { env = push_rec_types recdef renv.env;	→ assert false	_hd → List.rev lrec	(" We extract the tree of the projected argument ") let n = Projection.arg p in	(** Y3: There are only 2 cases for the type of $[\rho]$ here: Prod or anything else.	let check_one_fix Tevars remy recpos trees def = let efi = Array.length recpos in		match List.nth stack decrarp with	tack x m body in check_mested_fix_body illformed renv' (decr-1) stack' rs bod
(* To each inductive definition corresponds an array describing the	rel_min = renv.rel_min=n; genv = iterate (fun ge -> lary Not_mubters::ge) n renv.genv }	(* Dis removes global parameters of the inductive types in lc (for nested inductive types only) *) let dammy_univ = Lavel.(make (Ullobal.make (BirPath.make [Id.of_string	in recargs_constr_rec ienv trees [] c	spec_of_tree (List.nth wf_args n) Dead_code → Dead_code	We only care if type of [p] is a Prod. Otherwise we map the stack into SArgs. *)	(* Checks if [t] only make valid recursive calls		Strg _ + (* A match on the way *) None Stlomare (_,_,e,reckrg) +>	<pre> [] -> let renv' = push_var_renv renv (redex_level rs) (x,a) in</pre>
structure of recursive arguments for each constructor, we call it the recursive spec of the type (it has type recards weet). For	time the chart result a	"implicit")) == 80)	in (* starting with re env = [] seems safe because any unbounded Rel will be	Not_subters -> Not_subters Internally bound subters s -> Internally bound subters s)	Prod (n,a,d) -> ("" Y2: open the binder (ctx : a) into ctx ")	[stack] is the list of constructor's argument specification and arguments that will be smalled after reduction		let c = whd_all ?evars renv.env (lift n recArg) in let hd, _ = decompose_app_list c in	check_mested_fis_body illformed renv' (decr-1) [] rs body
the recursive spec of the type (it has type recarge weet). For checking the guard, we start from the decreasing argument (Rel n) with its recursive spec. During checking the guardness condition,	Needleduce of erv * fix guard error NeWeedleduce	let dammy_implicit_mort = mkType (Universe_make dammy_univ) let lambde_implicit n m =	satigned Norec *) build_recarge_nested (env,[]) tree (ind, args)	Count c ->	(** Y2: open the binder [ctx : a] into ctx *) let d = Localksum (n,a) in let ctx, a = whd_decompose_prod_decls ?evers env a in	example u in t where we have (match with + t end) u; [rs] is the stack of redexes traversed w/o having been triggered *)		match kind hd with Construct → Somm (contract_fix fix, stack)	_ → illformed ()
we collect patterns variables corresponding to subterms of n, each of them with its recursive spec. They are organized in a list lat	(* Definition and manipulation of the stack *)	let mon = Context.make_annot Anonymous Sorts.Relevant in let lambda implicit a = mkiambda (anon, dummy implicit mort, a) in	(* [restrict spec env spec o] restricts the size information in spec to what is	begin try let "Environ.comstant value in renv.env c in Not subters	let env = push_rel_context ctx env in (** Yl- who [a] which should have from of an application [tv area] *)	let rec check_rec_call_stack renw stack rs t = match kind t with		Cofix Ind Lambda Prod LetIn 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 * recargs) list which is sorted with respect to the first argument.	type stack_element = (" arguments in the evaluation stack ")	iterate lambda_implicit n m	allowed to flow through a match with predicate p in environment env. ") (** TODO %); what is this predicate thing about? check the usage to find out.	with NotEvaluableCorat (ExPrinitive (_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) →		Rel War Const App Case Fix Proj Cast Mets Evar + None)	meeds the state to be reduced before continuing checking *) satch meedreduce_of_head meedreduce_of_stack_stack_with
	(* [constr] is typed in [guard_env] and [int] is the number of binders added in the current env on too of [suard env.env] *)	let shatract_mind_ic ntype spars mind ic = let ic = Arraw.mag (fun (cts. c) -> Yerm.it skProd or Letin c cts) ic in	*) (** YJ: if mec is [Not mobered etc then no restriction to it. Otherwise.	Orienties specif Swarz reny on 1	****	let rs, stack = Array.fold right (fun a (rs.stack) →		ent (in. u an cu) ->	Noted Reduce → rs Need Reduce as e →
	SClosure of fix_check_result * guard_env * int * constr	let rec realism (of k c s	unions [n] as such as resulting a forall. Instella, if the head of smallesting	NotivelumbleCount - + Not_nubters	(** 11: it can account type (a) is some inductive applies with (args), then it could be a subter. ") (** 11: If the spec of [alt] is subters, intersect and return subters. Orthorise return spec. *) [Ind ind ->	lat contactor or a deat on all one or to		rbank ran rall state rany Solice/Dature stank ra (for () is	(* Expand if possible, otherwise, last chance, propagate need
* Environment annotated with marks on recursive arguments *)	Science of fix_check_result " guard_env " int " constr (" arguments applied to a "match": only their spec traverse the match ") Sarg of subterm_spec Lazy.t	let hd, args = decompose_app_list c in match kind hd with Ind ((mind',1),_) when MutInd.ConOrd.equal mind mind' ->	of [p] is an inductive, then return the intersection of spec with spec of [unwrapped p]. *)	end	("" MJ: If the spec of [elt] is subterm, intersect and return subterm. otherwise return spec. ")	let stack = push_stack_closure renv needreduce a stack in (rs.stack)) args (rs.stack)	renv.env cu	if evaluable_constant kn renv.env then Some (constant_value_in []) else None)	for expansion, in the hope to be eventually erassed *) match expand_head () with
(* tells whether it is a strict or loose subterm *) type size = Large Strict	let () x y = metch x with	let rec drop garges n = function	[unerapped p]. *) (** NOTE XJ: this is to fix propest after beta-ints cuts. Lennard's code has more info about the cuts and some comments about the logic, but it is	Var _ Sort _ Cost _ Prod _ Letin _ App _ Ind _ Construct _ Cofix _ Int _ Float _ Array> Not_subterm		in check_rec_call_stack renv stack rs f			None -> e :: List.tl rs Some (c, stack') -> check_rec_call_stack rerw (stack'@stack) rs c
(* merging information *)	NeedReduce → x NeMeedReduce → y	_ :: args when n > 8 → drop_params (n-1) args args → lambds_implicit n (Term.applist (mkHzl (ntype·n·k-i),	basically translating code to english. The top of his document says he does not understand why the restriction either. Maxima Denes who implemented the		let many = lary (match Lary.force spec with Not_subters Dead_code Internally_bound_subters _ as spec ->	end		mbda (x,x,b) → begin	and check_inert_subtarm_rec_call remy rs c =
let size_glb s1 s2 = match s1,s2 with	let rec medreduce_of_stack = function	List_Smart.map (replace_ind (n-k)) args)) in	first version of this could perhaps give an explanation.") let restrict spec Yevers erw spec p =	(* Other terms are not subterms *)	spec Subters(1,s,path) ->	Rel p -> begin		let meedreduce, rs = check_rec_call remv rs a in metch 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	[] → Motembleduce Skrg _ :: 1 → needreduce_of_wtack 1	drop_params npars args _ → map_with_binders succ replace_ind k c	match spec with Not_subters Internally_bound_subters -> spec	and lary_subters_specif Tevers renv stack t = lary (subters_specif Tevers renv stack t)	(** UI: (westion: in the first intersection, the contribution of [p] boils down to [args] only. *) (** UI: Find out how get_recarge_approx uses args (which does	(* Test if [p] is a fixpoint (recursive call) *) if renv.rel_min <= p && p < renv.rel_min=nfi then		elt :: stack → let renv, stack, b = pop_argument ?evars needreduce renv elt	level of the redex stack *) let need reduce, rs = check_rec_call renv rs c in
(* possible specifications for a term:	Stlosure (needreduce, , , , ,) :: 1 \Rightarrow needreduce needreduce_of_stack 1	in Array.map (replace_ind 0) lc	1 ÷	and stack element specif Pewers - function	(** TJ: Find oud how get_recarge_approx uses args (which does not really make serve) *)	(* the position of the invoked fispoint: *) let glob = renv.rel_min*nfi-1-p in	stack x a b	in check_rec_call_stack_renv_stack_rs_b	check_rec_call_state renv meed_reduce [] rs (fum () → None)
- Not subters: when the size of a term is not related to the recursive argument of the fixedist	let redex_level rs = List.length rs		(* Optimization: if the predicate is not dependent, no restriction is needed	Silosure (_, h_renv, _, h) -> lazy_subters_specif Towars h_renv [] h Sing x -> x	let recarge = get_recarge_approx levars env path ind args in let path = inter_wf_paths path recargs in	(* the decreasion are of the rec rell: *)		file the check rec call stack (push var renv renv (redex level rs)	and check_rec_call new ra c =
- Internally_bound_subters: when the recursive call is in a subters	let push_stack_closure renv medreduce c stack =	match env.retroknowledge.Retroknowledge.retro_array with	if noccur_with_mets 1 (Context.Rel.length abacts) or then spec		Subters(1,s,path))	let np = recpos.(glob) in if List.length stack <- np then	(x,n)) [] =	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
of a redex and the recursive argument is bound to a variable which will be instantiated by reducing the redex; the integers refer to the number of redexes stacked, with 1 counting for the	(Silosure (meedreduce, renv. 0, c)) :: stack let outh stack closures renv 1 stack =	Some c' when (Constant.equal env c c' → true _ → false	let ew = push_rel_context absctx env in let arctx, x = who decompose prod decis Yevers env ar in	and extract_stack Towars = function [] → Lary_from_val Not_subters, [] elt : 1 → stack element specif Towars elt. 1	Skry marg > Skry (met iots specif or (lary Not subters))	set_med_reduce_top renv.env (NotEnoughArgumentsForFixCall glob) rs		end od (x.a.u) →	meed for reduction in the redex stack with rec calls on variables bound at higher levels of the redex stack ") List.mee first (check rec call stack reny [] (NoNescheckus::rm) c)
refer to the number of redexes stacked, with 1 counting for the variables bound at head in the body of the fix (as $s, g, [x]$ in [fix $f n := fun x \Rightarrow f x]$); there may be several such indices	List.fold_right (push_stack_closure renv NoNeedReduce) 1 stack	(* [get_recarge_approx env tree ind args] builds an approximation of the	let any a rush rel context arriv any in		in	else (* Retrieve the expected tree for the argument *) (* Deck the decreasing arg is smaller *)		od (x,s,u) → (* Note: we cannot ensure that the stack is empty because	List.map_first (check_rec_call_stack renv [] (NoNeedReduce::rs) c)
	let pub_stack_args 1 stack -	recargs tree for ind, knowing args. The argument tree is used to know when candidate	match kind i with	and primitive_specif Towars renv op args = let open Chrimitives in	<pre>elt :: filter_stack (push_rel d env) of stack' ("" YJ: if the stack element is not an inductive, it must not be a</pre>	let z = List_oth stack op in		non-accessible branches of "match" expressions can have arbitrary types (see #17873) *)	in let meed_reduce, rs = 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 -> 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	begin match spec with	match op with Arranget Arrandefault -> (* t.[i] and default t can be seen as strict subterms of t, with a	subterm. ") > List.fold_right (fun _ 1 -> SArg (set_intm_specif or (lary	match check_is_mabters (stack_element_specif Tevers z) trees (glob) with			meant (List.is_empty rs); match need_reduce with
the wf_math argument specifies which subterms are recursive; 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.map (function	not compute the number of recursive arguments. *)	Dead_code -> spac Subterm(1,st,tree) ->		Not_matters)) :: 1) stack []	(glob) with NeedleduceSubtern 1 → set_need_reduce renv.env 1 (illegal_rec_call new glob z) rs Invalidablesm → raise (finberdirec (renv.env.		check_rec_call_stack (push_var_renv renv (redex_level rs) (x,a)) []	match need_reduce with NeedReduce (env.err) -> raise (fixRunrdfrror (env.err)) NoNeedReduce -> ()
others are calls to bound variables - Dead code: when the term has been built by elimination over an	Silosure (needreduce, s,n,c) \Rightarrow Silosure (needreduce, s,n+k,c) $x \rightarrow x$)	let get_recarge_approx Severa env tree ind args = let rec build recarge (env. ra env as ienv) tree c =	let recarge = get_recarge_approx Tevars env tree i args in let recarge = inter wf maths tree recarge in	<pre>let arg = List.nth args 1 in (* the result is a strict subterm of the second argument *)</pre>		Invalidables -> raise (fishardfror (rest.est, illegal_rec_call rest glob z))	rs u		et inductive of mutfix Tevers env ((nvect.bodynum).(names.types.bodies as
empty type	let lift1_stack = lift_stack 1	let x,largs = decompose_app_list (whd_all Tewars env c) in match kind x with	Subters(1,st,recarge)	let mubt = mubters_specif Tevars renv [] arg in begin match subt with	<pre>let find_uniform_parameters recindx rangs bodies = let mbodies = Array.length bodies in</pre>	else check_rec_call_state reny NoNeedNeduce stack rs (fun () ->	1.0	Fix () () typerray better as randef() us	cdef)) = let nbfix = Array.length bodies in
	/	Prod (ma,b,d) -> numert (List.is emoty haron):	end Not subtern	Subtern (., .x, wf) → let wf arms = (dest subterns wf).(0) in	let min_inds = Array.fold_left min range recinds in (" We work only on the i-th body but are in the context of n bodies ")	match lookup_rel p renv.env with	typarray in		if Int.equal obfix 8
type mihtem_spsc = Smitzer of (Int.Set.t * mize * wf_paths) Dand_code Not_mubters	(* (6 Computing the recursive subterms of a term (propagation of size information through Cases).) *)	build_recarge (ienv_push_var ienv (ns, b, sk_norec)) tree d Rol k ->		spec_of_tree (List.nth wf_args 8) (* first and only parameter of 'array	(se work only on the 1-th cody out are in the context of h codies) Let rec aux i k muniformparams c =	LocalDef (_,c,_) -> Some (lift p c, []))		art ren' = pum_fiz_ren' ren' recom in Array.fold_left (fun rx body -> let meedreduce', rx = check_rec_call rens' rx body in check_rec_call_state ren' needreduce' stack rx (fun> None))	not (int.equal (Army.length Nuct; notis) not (int.equal (Army.length types) notis) not (int.equal (Army.length names) notis)
Not_subtern		(* free variables are allowed and assigned Norec *)	(* [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	match kind f with	and		check_rec_call_state renv needreduce' stack rs (fun> None))	bodynus < 8
Internally_bound_mubters of Int.Set.t	let lookup_matterms env ind = let (_,mip) = lookup_mind_mpecif env ind in	<pre>(try and (tist.nth ra_env (k-1)) sdth failure _ Invalid_argument _ → mk_morec)</pre>	the fispoint we are checking. [renv] collects such information shout variables.	Not_subters → Not_subters Internally_bound_subters n → Internally_bound_subters n	Rel n → (* A recursive reference to the i-th body *)	Case (ci, u, pms, ret, iv, c_0, br) -> (* iv ignored: it's just a cache *)		PE DODINE	bodynum >= nbfix then anomaly (Pp.str "Ill-formed fix term.");
let eq.wf.paths = Rtree.equal Declareops.eq.recarg	mip.mind_recarge	Ind ind kn → (* When the inferred tree allows it, we consider that we have a		end	if Int.equal m (mbodies * k - i) them List.fold_left_i (fun j muniformparams m →	let (ci, (p,_), _iv, c_8, brs) - expand_case renv.env (ci, u, pms, ret, iv, c_8, br) in	1	d _ Construct _ → check_rec_call_state new NoNeedReduce stack rs (fun () → None)	let fixenv = push_rec_types recdef env in let vdefj = judgment_ef_fixpoint recdef in
	let match_inductive ind ra - match ra with	potential nested inductive type *)	let rec subterm_specif ?evers renw stack t = (" maybe reduction is not always necessary! ")	let set_iots_specif or spec =	match kind a with Rul m when Int.equal m (k - j) ->	let needreduce_c_8, rs = check_rec_call renv rs c_8 in		oi (p c) →	let raise err env i err = error ill formed rec body env (Tupe errors.fixQuantError err) names i
let inter_wf_paths = Rtree.inter Declareops.eq_recarg inter_recarg Norec	Mrsc (RecArgled i) → Ind.Canfird.equal ind i Norsc Mrsc (RecArgfrim _) → false	begin match dest_recarg tree with Nrec (RecArgInd ind') when (End.equal env (fat ind_kn) ind' ->	let f,l = decompose_app_list (whd_all Tevars renv.env t) in match kind f with	lary (match Lary, force spec with	(* a reference to the j-th parameter *) muniforeparame	(* compute the recorp info for the arguments of each branch *) let rs' = NoNeedReducetrs in let rs = redec_level rs' in		let meedreduce', rs = check rec call reny rs c in	ixerv wdefj in (* Check the i-th definition with recess k *)
let incl_wf_paths * Rtree.incl Declareops.eq_recarg inter_recarg Norec	(* In (match c as z in ci v s return P with C i x s ⇒ t end)	build_recarge_mested denv tree (ind_kn, large)	Rel k → subterm_var k renv Case (ci. u. pez. n. iv. c. lbr) → (* iv innoved: it's fast a cache *)	Not_subters -> if nr >= 1 then Internally_bound_subters (Int.Set_singleton nr) slow Not_subters mec -> spec)	(* out a secondary this outs a board on the size of an			check_rec_call_state new needreduce' stack rs (fun () -> (* we try hard to reduce the oral may by looking for a	let find ind i k def = (* check fi does not appear in the k+1 first abstractions,
let spc_of_tree t = if eq_wf_paths t sk_norec	[branches_specif renv c_spec ci] returns an array of x_s specs knowing	end Const (c,) when is primitive positive container may c ->	let (ci, (p,_), _iv, c, lbr) - expand_case renv.env (ci, u, pms, p, iv,		estrudable prefix of uniform arguments *) min j nuniformparams) 0 nuniformparams 1	branches, specif renv (sat_iots_specif nr (lary_subters_specif ? evers renv [] c_0)) ci in let stack' = filter_stack_domain ?evers renv.env nr p stack in		constructor in c (we unfold definitions too) ") let c = whd_all ?evers renv.env c in	gives the type of the k*1-eme shatraction (must be an inductive) *) let rec check_occur env n def =
the Not_makers (Int.Set.empty, Strict, t)	c_spec. ") let branches_specif renv c_spec ci = let car = ("" YJ: an array of ints, arities of constructors of C ")	begin match dest_recarg tree with New (Beckgyin c') when (Constant.equal env c c' ->	c, sery in let stack' = push_stack_closures renv 1 stack in let cases seec =	exception fixSuardirror of env * fix pund error	else	let stack' - filter_stack_domain ?evers renv.env nr p stack in		let hd, args - decompose app c in let hd, args = match kind hd with	sate rec conce occur env n enr = sate kind (whd_sll ?evars env def) with Lambda (x_m,b) →
lat marra internal misterna 11 12 m	(* We fairly the resulter free associated to the industries of the match	build_meangy_mested_primitive innv tree (c, large)	branches specif reny Clary subterm specif Tevars reny [] c) ci	let illeral per call new for a function	> fold_constr_with_binders succ (aux i) k muniformparams c	Array.fold_left_i (fun k rs' br' →		Cefix cefix -> decommons and (with all Texars renv.env (mkAno (contract cefix	if noccur_with_mets n mbfix a then let env' = push_rel (localAssus (x,s)) env in
let marge_internal_multares 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 c_spec.	None Nne> ek_name end	let atl -	SClosure (_,arg_renv,_,arg) →	in Array.fold_left_i (fun i → mux i 0) min_indx bodies	let stack br = push_stack_args case_spec.(k) stack' in check_rec_call_stack_renv_stack_br_rs' br') rs' brs in	cofix, args	1)	if Int.equal n (k + 1) then
let inter_spec s1 s2 =	Note that c_spec might be more precise than [v] below, because of nexted inductive types. *)	per -> ek_norec	Array.mapi (fun i br' -> let stack_br = push_stack_args (cases_spec.(i)) stack' in	let le_lt_vars = lary (let (_,le_vars,lt_vars) =	(** Given a fispoint [fix $f \times y \times n := phi(f \times y \cup t, \dots, f \times y \cup^t t')$]	<pre>let needreduce_br, rs = List.xep_first rs' in check_rec_call_state renv (needreduce_br needreduce_c_0) stack</pre>		> hd, args in match kind hd with	(* get the inductive type of the fixpoint *) let (mind, _) =
match s1, s2 with Dead_code -> s1 Dead_code,> s2	(* let (_,mip) = lookup_mind_mpecif renv.env ci.ci_ind in *) (* let v = dest_mubterss mip_mind_recorps in *)	and build_recarge_nested (envrs_env as inv) tree (((sind,i),u), large) - (" If the inferred tree already disalloss recursion, no need to go further	subterm_specif ?evers renv stack_br br') lbr in	List.fold_left (fun (i,le,lt) mbt →	transfer impose $\{z_1, z_1, z_2, \ldots, z_k\}$ is the functional reservative on $[a]$, with $[z]$ set uniform we build in context $\{z, z_k, y_2 \in \{z_k, z_k \in \{z_k\}\}\}$ a term $\{f : x_i \in z_k = f_k \in \{z_k, \ldots, f_k \in \{z_k\}\}\}$, say $[g_k z_k]$, of some type $[f \text{small } (z_k \in \{x_k\}) (c_k \mid z_k, y_k, z_k)]$, to that	rs (fun () -> (* we try hard to reduce the match away by looking for a	p), [])	Construct $_ \rightarrow$ Some (args.(Projection.npars p * Projection.arg	try find inductive Tevers env a with Not_found →
	<pre>let v = dest_subterms (lookup_subterms renv.env ci.ci_ind) in Array.map List.length v in</pre>		let spec = subters_spec_glb stl in restrict_spec Tevers renv.env spec p	match Lazy, force sht with (Subters(_,Strict,_) Desd_code) -> (i-1, le, i::lt)	[fix f z n := phi(f u t,, f u' t')], may [pmi], of some type [formll $(z:C(x,y))$ $(n:I(x,y,z))$, $I(x,y,z,n)], so that$	constructor in c.g (we unfold definitions too) *) let c.g = whd_all Tevers renv.env c.g in		Cofix Ind Lambda Prod Letin Sort Int Float Array -> assert false	raise_err env i (RecursionNotUnInductiveType a) in let mib, = lookup_mind_specif env (out_punivs mind) in
, Not_subters → s2 Internally bound subters 11. Internally bound subters 12 →	(** YJ: for each constructor, generates a list of mize=[ncm]. *) (** YJ: if discriminant spec [c_spec] is the same inductive as *)	if eq.wf_paths tree ek_morec than tree	Fix ((recindes,i),(_,typerray,bodies as recdef)) →		[fun x y z \Rightarrow pxi z] is of same type as the original term *)			Sert Int Finet Army -> assert false Rel War Coret App Case Fix Proj Cast Meta Evar -> None	<pre>if mib.mind_finite \(\psi \) finite then raism_err env i (RecursionNotEnductiveType m);</pre>
Internally_bound_matters 11, Internally_bound_matters 12 → Internally_bound_matters (merge_internal_matters 11 12) Subters (11,a1,t1), Internally_bound_matters 12 → Subters	Array.mapi (fun i mcm → (* i=1-th catructor has arity nos *)	let mib = Environ.lookup_mind mind env in let mumper = mib.mind_sparams_rec in	(* when proving that the fixpoint 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,[],[]) resv.genv in (la_vara_lt_vara)) in	let drop_uniform_parameters muniformparams bodies = let mbodies = Array.length bodies in	let hd, args = metch kind hd with Cofix cofix → decompose_app_list (whd_all ?evers renv.env (ferm.applist		end	(sind, (env', b)) else check occur env' (n+1) b
(merge_internal_nubterns 11 12,st,t1) Internally bound nubtern 11 13,styten (12,s2,t2) → Subtern	let lyes : spec Lary.t = lary (match Lary.ferce c_spec with	let nonecpar = min.mind.nparams = mumpar in let (lear.) = List.chco mumcar larce in	furthernore when f is applied to a term which is strictly less than	RecursionDelliegalTere(fx,(arg_renv.env, arg),le_lt_vars)	let rec aux i & c = let f, l = decompose_app_list c in	(contract_cofix cofix, args)))	1.9	r id → check rec call state reny NoNeedReduce stack rs (fun () → n	else snowaly -label: "check_one_fix" (Pp.str "Bad occurrence of
[merge_internal_nublerns 11 12,02 22 22 22 22 22 23 24 24	Subterm (_,_,t) when match_inductive ci.ci_ind (dest_recarg t)	let mentyp = mib_did_ntypes in (* Extends the environment with a variable corresponding to	if not (check_inductive_codomain Tevers 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 *)	match kind f with	match kind bd with Construct cutr -> Somm (apply_branch cutr args ci brs, [])		let open Context Named Declaration in match lookup named id renv.env with	_ → raise_err env i NotEnoughAbstractionInfisBody
Subters (ergs_internal_subterms 11 12, size_glb at a2, inter_wf_paths t1	let vrs = Array.of_list (dest_mbterms t).(i) in spect (int.equal ncs (Array.length vrs)):	("Excess the environment with a variable corresponding to the inductive def ") let (env', so isow') = iero push inductive Severa iero ((mind.u).lter) in	Not_subterm	NotEnoughArgumentsForFisCall fx	("A recursive reference to the i-th body ") if Int.equal n (nbodies * k - i) then	Cofix Ind Lambds Prod Letin Sort Int Float Array - assert false		LocalAssum None	let ((ind, u), _) as res = check_occur fixers 1 def in let , sio = lookup mind specif env ind in
2)	accert (int.equal ncm (Array.length vrm)); Array.map spec_of_tree vrm	(* Persenters conversed to any! *)	elme let (ctxt,clfix) = whd_decompose_prod Tevars renv.env typerray.(i) in	let set_meed_reduce_one erw or err rs =	if Int.equal n (nbodies * k - i) then let new_args = List.skipn_at_best numiforeparams 1 in Term.applist (f, new_args)	Sort Int Float Array - * assert false Rel Yar Const App Case Fix Proj Cast Meta Euer - * Nora		LocalDef (_,c,_) -> Some (c, []))	<pre>let _, sip * lookup_mind_specif env ind in (* recursive sprop means non record with projections -> squashed *)</pre>
let subters_spec_glb = Array.fold_left inter_spec Desd_code	Array.map spc_of_tree vra Desd_code > Array.make nca Desd_code Internally_bound_subters _ ms x > Array.make nca x	<pre>let lpar' = List.map (lift montyp) lpar in (* In case of mutual inductive types, we use the recarge tree which was</pre>	let oind = let env' = push_rel_context ctxt renv.env in	let er = List.length rx in let rx1, rx2 = List.chop (er-nr) rx in	Term.applist (f, new_args)		L	tin (s,c,t,b) -> let medreduce_c, rs = check_rec_call remv rs c in	<pre>let () = if Ensiron.is_type_in_type env (GlobRef.IndRef ind) then ()</pre>
type guard_env =	Subters Not_matters -> Array.make nos Not_matters) in List.init nos (fun j -> lary (Lazy.force lvra).(j)))	computed statically. This is fine because nested inductive types with mutually recursive containers are not supported. *)	try Some(fst (find_inductive Texars env' clfix)) with Not_found → None in	let _, rs2 = List.sep_first rs2 in rs1 @ NeedReduce (env, err) :: rs2	c _→ map_with_binders succ (max i) k c	(* fnables to traverse Fixpoint definitions in a more intelligent way, ie, the rule :		let meedreduce_t, rs = check_rec_call renv rs t in begin	else match relevance_of_ind_body mip u with Sorts.Frelevant Sorts.RelevanceVar _ as rind →
(" dB of last fixpoint ")	car	let trees = if Int.equal accetyp 1 then [dest_subteres tree]	<pre>(match oind with</pre>	let set_med_reduce env 1 err rs -	in Array.mapi (fun i → max i 0) bodiex	<pre>if - g = fix g (y1:11)(yp:1p) (struct yp) != e &</pre>	with	match medical_of_stack stack medical_oc_c medical_oc_t	<pre>if not (Sorts.relevance_equal names.(i).Contest.binder_relevance ind)</pre>
(* dB of variables denoting subterms *)	(** YJ: checks if the codomain of predicate [p] is an inductive by wh- unwresoins. *)	<pre>else Array.map (fun mip -> dest_mubterms mip.mind_recargs) mib.mind packets</pre>	Some (ind, _) -> let obfix = Array, length typerray in	Int.Set.fold (fun n -> set_meed_reduce_one env n err) 1 rs	let filter fix stack domain or decrarp stack muniformnames -	in at as & - f is guarded with respect to the set of pattern variables S		(* Stack do not require to beta-reduce: let's look if the	then raise_err env i fixpointOnTrrelevantInductive Sorts.Relevant + ()
genv : subterm_spec Lazy.t list;	let check inductive codemain Tevars env p = let abscts, ar = whd_decompose_lambds_decls Tevars env p in let env = push_rel_context abscts env in	in let sk_irecarge j mip =	let recargs - lookup_subteres renv.env ind in	let set_med_reduce_top env err rs = set_med_reduce_com env (List.length rs) err rs	let rec mux i muniforeparams stack - match stack with	in T1 Tp & - sp is a sub-term of the formal argument of f &	body of the	let meds *) let spec = lazy_subters_specif Tevars renv [] c in	in .
let make_renv env recarg tree -	let env = push_rel_context about env in let arcts, s = whd_decompose_prod_decls ?evers env ar in	(* The nested inductive type with parameters removed *) let auxlevect = abstract_mind_bc auantyp auxopar mind mip.mind_mf_lc in	let renv' = push_fix_renv renv recdef in let renv' =	type check_subtere_result =	[] -> [] n :: ntack ->	 f is guarded with respect to the set of pattern variables S-fue) in a 		let stack = lift1_stack stack in check_rec_call_stack (push_let_renv (s,c,t,spec)) stack rs b	in
Family and	let env = push_rel_context arcks env in let i,_1' = decompose_app (whd_all havars env s) in	let pathy - Array mari	(* Why Strict here ? To be general, it could also be	Invalidations Needleduction of Int.Set.t (* easty - Newschieduce *)	let uniform, nuniformparams - if nuniformparams - 0 then false, 0 else true, nuniformparams -1 in	then f is guarded with respect to S in (g al am).		Needleduce -> check rec call stack reny stack rs (subst1 c b)	lat rv = Army.mmp2.i find_ind nvect bodies in (Army.mmp fat rv, Army.mmp and rv)
rel_min = recarge2; (" recarg = 0 -> Rel 1 -> recarg; Rel 2 -> fix ") gen = [tary.from_vel(Sabtere(Int.Set.empty, Large,tree))] }	<pre>let i,_1' = decompose_app (whd_all Tevars env s) in isind i</pre>	(fun k c -> let c' - hof_prod_applist ?evers env' c lpar' in	large") assign_var_spec renv'		let a -	fix ((recindxx,i),(_,typarray,bodies as recdef) as fix) →			(Array.map fst rv, Array.map snd rv)
let push, ver renv (x,ty,spec) = f env = push rel ((conlinum (x,ty)) renv.env:	(** TJ: I guess [ienv] means inductive environment, and this is code to suild of caths. ")	("skip non-recursive parameters ") let (ienv',c') = ienv_decompose_prod 'hwars ienv' nonrecpar c' in build recerns constructors ienv' trees.(i).(k) c')	<pre>(nbfix-1, lary (Subters(Int.Set.empty,Strict,recarge))) in let decrArg = recinder.(i) in let theBody = bodies.(i) in</pre>	(" Check term c can be applied to one of the autual fixpoints. ") let check in subterm x tree = meth lawy force x with	if uniform Int.equal i decrarg them a	let decrArg = recindex.(i) in let nbodies = Array.length bodies in		st (c,_,t) → let rs = check_inert_subters_rec_call reav rs t in let rs = check_rec_call stack reav stack rs c in	et check_fix Yevers env ((nvect,_),(names,_,bodies ax recdef) ax fix) -
rel_min = renv.rel_min=1;	(* The following functions are almost duplicated from indtypes.ml, except	build_recargs_constructors isnv' trees.(j).(k) c') muslcvect	let mb0fAbst = decrArg-1 in	Subters (need_reduce,Strict,tree') ->	(* descrivate the status of non-uniform parameters since we cannot guarantee that they are preserve in the recursive	let honders = frequency() length hodies in let ref = frequency() length hodies in let ref = frequency() length (check_inert_mubters_rec_call remv) (billeesBedocs:rb_typerrep; let remv' = push_fix_remv remv recdef in		Set rs = check_rec_call_stack renv stack rs c in rs	let (minds, rdef) = inductive of mutfix Nevers env fix in let flags = Environ.typing flags env in
genv = spec:: renv.genv }	that they carry here a poorer environment (containing less information). *) let ienv_push_var (env, lrs) (s,s,rs) =	in sk.paths (Nroc (Neckrylini (mind,j))) paths	let sign,strippedBody = whd_decompose_lambda_n_assum ?evers renv.env rbDFAbst theBody in	if incl_wf_paths tree tree' then NeedReduceSubters meed_reduce else InvalidSubters	calls ") SArg (set_iots_specif or (lazy Not_subters)) in		1.5	rt Int float →	if flags.check_guarded then let pet tree (kn.i) =
	(push_rel (LocalAssum (x,a)) env, (Norec,ra)::lra)	in let irecarge - Array.mapi sk_irecarge mib.mind_packets in	(* pushing the fix parameters *) let stack' * push_stack_closures renv l stack in	Dead_code -> NeedleduceSubtern Int.Set.empty Not middless Subtern (James) ab Inval of Subtern	s :: aux (i*1) numiformparams stack in aux @ numiformparams stack	stack) bodies in let bodies - drop_unifore_parameters nuniforeparaes bodies in		(* See [Prod]: we cannot ensure that the stack is empty *)	let mib = Environ.lookup_mind kn env in mib.mind_packetx.(i).mind_recarge
let push_let renv (s,c,ty,spec) = { env = sush rel (locallet (s,c,ty)) renv.env:			let reny' - push_ctxt_reny reny' sign in	Not_mattern Sabtern (_,large) -> InvalidSabtern Internally_bound_mattern 1 -> NeedReducSabtern 1		let fis_stack = filter_fis_stack_domain (redes_level rs) decrArg	1.4	ray (_u,t,def,ty) →	in let trees - Array.map (fun (mind,_) -> get_tree mind) minds in
at push_let renv (x,c,ty,upec) = { env = push_rel ((ocallet (x,c,ty)) renv.env; rel_uin = renv.rel_uin+1; onv = nect; renv.ceu, uin+1;	let inv_push_inductive Tevers (env, ra_env) ((mind,u),lpar) - let mib = Environ.lookus mind mind env in	(Rtree.ek_rec irecargs).(i)	let repy" -		match needreduce, elt with	let fix stack - if List.length stack > decrare then List.firstn			for i = 8 to Array.length bodies - 1 do
rel_min = renv.rel_min=1; genv = mpec:: renv.genv }	let mib = Environ.lookup_mind mind env in let ntures = mib mind ntures in	(Atrea.mk_rec irecargs).(1) and build_recargs_mested_primitive (env, ra_env) tree (c, large) = (f an of raths tree at rooms than tree.	let rem'' = if List.length stack' < nbOfRbst then rems''	(** HIGH LEVEL: [stack] is a stack of organizate to a term? [p]. this function "filters" (actually mans) [stack] by refining the	Midwelleton Classes (Midwelleton n r) is	(dendrost) fix stack also fix stack to		(" See [Prod]: we cannot ensure that the stack is empty ") but on a forms fold laft (check inset witters per call result on t	had (flows hards) a solud (d) da
rel_min = recv.rel_min=1; genv = mpec::renv.genv }	let mib = Environ.Lookup.mind mired env in let nitypem = mib.mind.ptypem in let punh.ind mip env = let = relevance.of.ind.body mip u in	and build_recarge_rested_printitive (env, ra_env) tree (c, large) = if eq_wf_paths tree sk_rorec then tree	else let decrârg = List.eth stack' decrârg in	this function "filters" (actually maps) [stack] by refining the [Silosure]s in the stack into [Skrgs].	Notembergoon, all but Notembergoon Stream (Notembergoon, , , , ,) → (* Neither function nor args have rec calls on internally bound variables	(decrarge1) fix_stack else fix_stack in let stack_this = lift_stack mbodies fix_stack in	in	(* See [Prod]: we cannot ensure that the stack is empty *) let rs = Array.fold_left (check_inert_subterm_rec_call renv) rs t	had (flows hooks) a mobil (4) do
rel_min = revv.rel_min=1; gme = spec: revv.gme st maxign_nov_upuc revv (i,spec) = { resv with genv = list_maxign revv.genv (i-1) spec }	let sit = findrem.Lookup.mind mind env in let chypes = sith.inni, chypes in let push_ind sip env = let r = nishunce, of_ind hody mip u in let men = Condent.make_unent homoproum r in let deal = localismum (enve. held conduction) enver env (tope of inductive let deal = localismum (enve. held conductive fivers env (tope of inductive let deal = localismum (enve. held conductive fivers env (tope of inductive let deal = localismum (enve. held conductive fivers env (tope of inductive fivers)	and build recorpt rested printitive (now, ra_eme) tree (c, large) = if eq. of paths tree sk_nerse then tree also late trapes = 1 in (* Printitive types are modelled by non-emissi inductive trees *)	elm	this function "filters" (actually maps) [stack] by refining the [Silosure]s in the stack into [Sirgs]. The way is an follows: ") (file of District of Interface of Sirgs).	Noteethedoo, Stleame (Noteethedoo, _, o, c) -> (* Nother function nor args have rec calls on internally bound variables *) let moc - whack element specif fewers elt in	(decring*) fix_stack else fix_stack in let stack_this = lift_stack obsoles fix_stack in let stack_others = lift_stack rhodies (List_firstn numiforeparaes fix_stack) in	in	(* See [Prod]: we cannot ensure that the stack is empty *)	<pre>Let (femv_body) = rdef.(i) in let renv = make_renv femv muect.(i) treex.(i) in try check_one_fix ?evers renv nuect treex body with finbardfrenv (fixenv_err) -></pre>
rel_min = revv.rel_min=1; gme = spec: revv.gme st maxign_nov_upuc revv (i,spec) = { resv with genv = list_maxign revv.genv (i-1) spec }	let with "Environ.Lookup sind wird onv in let obysa" - with sind of types in let push, ind wip onv " let r = relevance of ind body wip u in let non - Context.asks, smoot. Annymous r in	and build_recorps_rested_primitive (eav, $ra_n = 0$) tree (c, large) = if eq.wf_paths tree sk_rerec then tree also hat rhypes = 1 in (* Primitive types are modalled by non-matual inductive a types *) hat $ra_n = 0$ = iint range (for $(r,t) \rightarrow (r,t)$ res. lift stypes t)) $ra_n = 0$ int $ra_n = 0$.	elme let decray = List.nth stack' decrary in let arg spec = stack_element_specif Yewers decrary in	this function "filters" (actually maps) [stack] by refining the [Silosure]s in the stack into [Sirgs]. The way is an follows: ") (file of District of Interface of Sirgs).	NoNeedReduce, Stlasser (NoNeedReduce, _, n, c) → (* Neither function nor args have rec calls on internally bound variables *)	(decring=1) fix_stack also fix_stack in let stack_this = lift_stack rhodies fix_stack in let stack_others = lift_stack rhodies (isst.firstn numiforsparams fix_stack) in ("Once, guard in the expanded fix ")		(* See [Proof]: we cannot measure that the state is empty ") but re * Army-fold_left (check_inert_multane_roc_call renv) rs t but rs * check_inert_multane_roc_call renv rs def in rs * check_inert_multane_roc_call renv rs ty in rs * check_inert_multane_roc_call renv rs ty in rs * check_inert_multane_roc_call renv rs ty in	let (few.yoop) = refet (i) in let reve = max_reve few revet.(i) trees.(i) in try check_one_fix Power reve out: trees body with foldendfrow (fixer_ver) -> erer_ill_formed_rec_body fixer (Type_erers.fickwedfrow ere) as
$m_{ij}^2 \sin r \cdot mean m_{ij}^2 \sin r^2 \\ = m_{ij}^2 \sin r \cdot mean m_{ij}^2 \sin r^2 \\ = \sin r \cdot m_{ij} \cos r \cdot m_{ij} \cos r \cdot m_{ij}^2 \cos r^2 \\ = m_{ij}^2 \sin r \cdot m_{ij}^2 \cos r \cdot m_{ij}^2 \cos$	Let she "Environ. Doning since fared me in Let types "eliabority-types in Let types "eliabority-types in Let myon." Confidence and the special since Let more "Confidence forces, brid-pred, applied Powers ow (type_af_inductiv (confidence forces, brid-pred, applied Powers ow (type_af_inductiv (confidence)), applied the confidence (confidence), applied the confidence powd, red died ow powd, red died ow Let owe "Environ-field, right push, led shi and product over in	and build, energy, nested printine (ear, $r_{\rm A}$, and) true $(r_{\rm c}$, $\log p)$ of eq. $r_{\rm c}$ and $r_{\rm c}$ are distribute as a present han true in the property of the p	elum lit dering - List.nth stak' dering in lit un_uper = luts, elemen_uperi Twars dering in setipn_var_uper renn'' (1, an_uper) in subtun_uperi Twars renn'' [] strippedody) [Listis (x,x,b) >=	this function "filter" (schally maps) [stack] by refraing the [Climaryle in the stack into [Sire]. The way is an fallow: " ("[n] * [blankt.x. n] when [blankt.x. n] -poor [blankt.x. lone. cd] -when(n) [blankt.k.blankt] way [w. n] when [blankt.k.blankt.k.] (""12 [n] blankt is a predicate, an it is a bynd") leff filter_stack_demain Pawars may no patakt ""12 [n] blankt into of [www. pain], and [n] the showly of the lambdis. ")	bitschedutes, Simum (GebesShetz, , , n, q,) = (* bittler function nor any bars was calls on intervally bound variables ") let spec * taked_slemet.pperf* Twears alt in (* but, sept of not a priori reagine to be rechected, so we pash a let *) (* major to be of the let will have to be locally aquends though, see fall cass *) pask_let reaw (s,lift n c,aspec), lift!_stack stack, b	(descript) fix that also fix that is bit that, that is lift that is bit that, that is lift, that is desire fix that is bit that, the is lift, that is desire fix that is bit that, the is lift, that is desired in the symbol is that the is lift, that is lift, that is lift, that is lift, that is in the lift lift lift is lift. If it lift lift lift lift lift lift lift		(* See [Prod]: we cannot assure that the stack is ampty *) let rs = Array.fold.left (check_inert_subters_rec_call renv) rs t let rs = check_inert_subters_rec_call renv rs def in	<pre>Let (femv_body) = rdef.(i) in let renv = make_renv femv muect.(i) treex.(i) in try check_one_fix ?evers renv nuect treex body with finbardfrenv (fixenv_err) -></pre>
rel_ufin - removal_pinet; par - speci - removal_pinet; it satisfy, we give rem (t, spec) - { [rew oilt per - litherality rem_spec (t-1) spec] [rew oilt per - litherality rem_spec (t-1) spec] [st per - lay r_remove rem $\sigma(x, b_1) = 1$ [st per - lay r_remove (t r_k - 1) then intervally_boad_painters [rew oilt per - litherality boad_painters [rew oilt per - litherality boad_painters [rew oilthing - litherality	Let the 'foreign Linding paid and we is the Linding Linding Linding with the Linding	and build_recorp_needed_printine (exp. re.gen) tree (c, large) = interp_needed_prints tree adjunct to the tree the temperature of the relative tree that the record of the relative tree tree trees are united by non-stand induction forms ? **The record of the record of	elm lbt decoding - List.abt stack' decoding in lbt mr_nume - stack_alement_nputf Towars decoding in muting_war_space_rest'(", may nume) in mathers_nputf Towars_rest'(") strippedbody) Limints_(a_pa_k) ->> limints_(a_pa_k) -> limints_(a_pa_k)	this function "filters" (actually maps) (stack) by refining the [Stimure] in the stack into [Strps]. The way in a fallous: ") ("[p] "[bhests. w] who [bhests. 1] word [bhests. [ws. cd] who(s) [bhests. [wc] ind seps]. cd]") ("12: [p] bints to a predicate, us it is a type? ("12: [p] bints to a predicate, us it is a type? ("12: [p] bints to a predicate, us it is a type?)	Industriates, "Linear (Cadendinates,, $x_i \in J \to \infty$)" (* Stiffer increase are sign to receil as noticeally insert sprainform or sign have receil as noticeally insert sprainform of the control of t	(mesteps) fix_taked size fix_taked in Let take_the lift_taked reduced fix_taked in Let take_there ifft_taked modifies fix_taked) in (*Dack gard in the squaded fix *) Let liftered() * error_lift_frend_frend_taked reduced fix_frend_taked bitformyblathetical fix_taked).	(°.	(* See [Proof) we convent somes that the state is empty ") but n = *kerps_Aris_Brit (*seek_steet_stater_rec_coll remon) or t but n = *kerps_Aris_Brit (*seek_steet_stater_rec_coll remon) or t but n = *check_steet_rect_stater_rec_coll remon rs_dris_ ts_n = *check_steet_rect_stater_rec_coll remon rs_dris_ rs_ rs_ rs_ rs_ rs_ rs_ rs_ rs_ rs_ r	let (fmw.yoop) = refet(i) in let renv = maku_renv fenv ovect.(i) trem.(i) in try chek_om_fix /bear renv nect trems body with foldendfrer (fixen_ren) -> erer_ill_formed_rec_body fixenv (type_erers.fixdureffrer er) an
ord_max = removal_state; gover = spec : removal_state; gover = spe	Let the foreign Lindon part of our in Let the foreign Lindon part in Let you following of the last part in Let you following of the last part in Let the relations from the Journal applicat Newer was (type, of interior (subschool), land) in Let the last include from the Journal applicat Newer was (type, of interior (subschool), land) in Let you for the last part of the last part of the last part of the Lindon part of the last part of the last part of the last part of the Lindon year (foreign part of the last part of the last part of the Common of the last part of th	and half energy, contad printing (see, π_1 , m_1) true $\{x_1, \log x_2, \dots, x_m\}$ for $\{x_1, \log x_2, \dots, x_m\}$ for the actual ten true is the size of th	elum lit dering - List.nth stak' dering in lit un_uper = luts, elemen_uperi Twars dering in setipn_var_uper renn'' (1, an_uper) in subtun_uperi Twars renn'' [] strippedody) [Listis (x,x,b) >=	Change of the profile	Numberdame, Elevane (Numberdame, $x_i, x_i \in 0 \Rightarrow (1 + in)$ where $(1 + in)$ the formula nor way have no cally no internally bound variables $(1 + in)$ into $(1 + in)$ the	(pectype) [ratation in the flat plant in let though a lift plant in lift plant in dending flat plant in let though a lift plant in lift plant in coloring (left friend in multirequirum flat in lift in the sequential lift in lift plant in the sequential lift in lift remain (figureren. lifthandlirum lett lift in lift in lift in lift remain (figureren. lifthandlirum lett lift in li	(* I ((* See [Proof]) we coment ensure that the state is only ") if it is "when pick [and (mock jourt, journ, good) remo)" is if it is "when jourt, journ, good) remo "re def is it is "when jourt, journ, good) remo "re def is it is not desched because it is considered as the methy's context ") if you will be a state of the state of the state's context ") or " [Note] - "	let (fmw.yoop) = refet(i) in let renv = make_renv fenv ovect.(i) trems.(i) in try check_one_fix /bower renv ovect trees body with foldenefirer (fixen_ren) -> erer_ill_formed_ren_body fixenv (%pm_erers.fixdureffrer er) an
Let unify ω_i years are $(L_i g_{ij})$: $(m_i \otimes 1)^2 = (-1) \log (1 + 1) \log $	Let us be forever, being pair and one in the lamb of the foreverse being pair and one in the lamb of side pair. The lamb of side pair is pair in the vertices d_{ij} and using a real pair in the contract of the lamb of t	and making company and another processing of the $(x_1, x_2, x_3, x_4, x_5, x_5, x_5, x_5, x_5, x_5, x_5, x_5$	elm lbt decoding - List.abt stack' decoding in lbt mr_nume - stack_alement_nputf Towars decoding in muting_war_space_rest'(", may nume) in mathers_nputf Towars_rest'(") strippedbody) Limints_(a_pa_k) ->> limints_(a_pa_k) -> limints_(a_pa_k)	this forction "filter" (factally pan) (steel) by refusing the [Billmark): in the steel in [Suph.]. [The planet, but the steel in [Suph.]. [The planet, but the planet, but t	In Bellenforce, Ellison (Medwellens, $x_i, x_i > 0$) we (whether leavis one up has we call in sixternil) hand workship and $x_i = x_i + x$	(sectory) Tick, take day Tick, take to the Tick, take done for, take to the total collection of the dependent of the total collection of the dependent of the total collection of the dependent of the dependent of the total collection of the dependent of the dep	and check if Int. check	(* in [red)) as creat mome that is state it and re? If a [red red) [red red red red red Int a red red red red Int a	let (fmw,loop) = rdef.(1) in let renv = make_renv fers reset.(1) trest.(1) in try check_one_fix Nower renv next tress body with fishandlerne (fixen.per) :> erer_ill_formed_ren_body fixenv (Nym_erers.fixkandlerne err) make
milgate removal, amilgate in a complex of the compl	Let sub - Notices. Loons guide state are in a series support and Loon for the Loon guide state are in the Loon for the Loo	undulinterrous postularization (e.e., r.um) two (r. large) - dread point two deposits to the control to the set of the deposits to the control to the set of the set	the before "list.th stack" decring in let my uper - their, dismit, quant' hower decring in let my uper - their, dismit, quant' hower decring in letter quant my uper and "(", "quant') and letter quant' hower men'" (") thingstony) list() (", "lessent (list.is, upply 1) in list() (", "lessent (list.is, upply 1) in list (", "lessent (list.is, upply 1) in list (", "lessent (list.is, upply 1) in list uper, task" "emer(stack hower stack in letter, uper l'hower (lash, lawr men (sta, uper)) stack" is letter uper (", "list.is and user, uperstance (", ") in "the list of user uperstance (") in "the list o	us is contine "tiller" (statis) yan (stat) by artising the [Discrept in the state in (Suppl.) [Internet, in	In Bellenforce, Ellison (Medwellens, $x_i, x_i > 0$) we (whether leavis one up has we call in sixternil) hand workship and $x_i = x_i + x$	(metropy) (lagation dain fragation in list study, large = 1/18, table doubles fragation in list study, large = 1/18, table doubles fragation in list study, large = 1/18, table doubles (last. fraction maniferrage and fragation). Once a general in the magnoist fragation in large = 1/18, framed, res. John person and (fragativerse fragation) (lagativerse fragativerse fragativ	ord check if Int. check else match	(* See [Proof]) we consist mome that the state is only ") If it is "not ", bryght (found, burt, buffers, good) in may "it belt in " then (buff, buff, buffers, me, good) may "it did in If it is " then (buff, buff, buffers, me, good) may "it did in It is not showful because it is considered as the well's context ") or ". If the (buff, buff, buffers, buff,	let (fmw,loop) = rdef.(1) in let renv = make_renv fers reset.(1) trest.(1) in try check_one_fix Nower renv next tress body with fishandlerne (fixen.per) :> erer_ill_formed_ren_body fixenv (Nym_erers.fixkandlerne err) make

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

Consistency Proofs

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.

Consistency Proofs

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.

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.

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)

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

Guard env: [m':Strict|n :Both
```

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).

```
Fail 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 [9] 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