Typing Total Recursive Functions in Coq

Dominique Larchey-Wendling
TYPES team, ANR TICAMORE

LORIA – CNRS
Nancy, France

`https://github.com/DmxLarchey/Coq-is-total`

Interactive Theorem Proving
ITP 2017, Brazilia, Sept. 2017

Turing completeness for (axiom-free) Coq

- Does Coq contain any μ-recursive function as a term $\texttt{nat}^k \rightarrow \texttt{nat}$?
- Axiom free Coq defines only *total* functions
  - meta-level (strong/weak) normalization
- The Kleene T predicate method (Bove&Capretta 2005):
  - T is cumbersome = small-step semantics
  - primitive recursive schemes hard to program with
- Lambda calculus method (LW 2017, big dev. of 25k lines):
  - left-most and head normalization (so also small-step sem.)
  - intersection type systems (solvability)

How avoid small-step semantics?

The content of the function type `nat` → `nat`

- What is contained in the type `nat` → `nat` ?
  - it depends on axioms (even if only of sort `Prop`)
  - without axioms, only total recursive functions (normalization)
  - but are every total and recursive functions present ?
- What are (total) recursive functions ?
  - recursive functions are an inductive class of relations
  - but totality depends on meta-theory:
    * Goodstein sequence (Kirby&Paris)
    * Finite Ramsey theorem (Paris&Harrington)
- *Turing completeness* for Coq-provably total recursive functions
  - with (short ?) Coq-implementation of this claim

Our method: avoid small-step semantics

- Bove&Capretta's hint (Kleene's normal form theorem):
  - μ-recursive fun. = minimization of primitive recursive fun.
  - Kleene's T pred. relates prog. and computations (prim. rec.)
  - primitive rec. fun. are (trivially) Coq-definable terms
  - unbounded minimization of these terms (mutual recursion ?)
- Kleene's T predicate = *small-step semantics*
  - implement as primitive recursive = awfully complicated
  - a provably correct compiler with prim. rec. schemes
- We avoid small-step semantics
  - *unbounded minimization* of decidable (& inhabited) predicates
  - *cost-aware big-step semantics* as Coq decidable predicate

Coq-provably total & computable relations

- To shorten notations, $\mathcal{N}$ denotes the type `nat`
- μ-recursive function $\mathbb{N}^k \longrightarrow \mathbb{N}$ = func. relation $\mathcal{N}^k \to \mathcal{N} \to$ `Prop`
  - an *inductive class* of functional/deterministic relations
  - constants, successor, zero, projections, composition, primitive recursion and *unbounded minimization*
  - each μ-recursive function is described by an *algorithm*
  - algorithm must be given, it *cannot* be extracted
- μ-recursive $R : \mathcal{N}^k \to \mathcal{N} \to$ `Prop` is total if $\forall \vec{v} : \mathcal{N}^k, \exists n : \mathcal{N}, R\ \vec{v}\ n$
- R is Coq-computable if $\forall \vec{v} : \mathcal{N}^k, \{n : \mathcal{N} \mid R\ \vec{v}\ n\}$
- Transforming $(\exists n, R\ \vec{v}\ n)$ into $\{n \mid R\ \vec{v}\ n\}$ called *reification*

Specificity of Coq existential quantifiers

- Three type of existential quantifiers (Σ-types)
  - for $P : X \to \texttt{Prop}$, non-informative $\exists x : X,\ P\ x$ of type $\texttt{Prop}$
  - for $P : X \to \texttt{Prop}$, partially info. $\{x : X \mid P\ x\}$ of type $\texttt{Type}$
  - for $P : X \to \texttt{Type}$, fully info. $\{x : X \ \&\ P\ x\}$ of type $\texttt{Type}$
- Reification is a map $(\exists x : X,\ P\ x) \to \{x : X \mid P\ x\}$
  - axiom called *Constructive Indefinite Description*
  - alternativelly, it is a map $\texttt{inhabited}\, X \to X$
- Reification can be implemented *without axioms*:
  - when X is an *enumerable* type (like $\mathcal{N}$)
  - when $P : X \to \{\texttt{Prop}, \texttt{Type}\}$ is *decidable*
  - implementation by *unbounded minimization*

Inductive definitions of Coq existential quantifiers

$\texttt{Inductive inhabited}\ (P : \texttt{Type}) : \boxed{\texttt{Prop}} :=$

$\quad \mid \texttt{inhabits} : P \to \texttt{inhabited}\ P$

$\texttt{Inductive ex}\ \{X : \texttt{Type}\}\ (P : X \to \texttt{Prop}) : \boxed{\texttt{Prop}} :=$

$\quad \mid \texttt{ex\_intro} : \forall x : X, P\ x \to \texttt{ex}\ P$ (also denoted $\exists x : X, P\ x$)

$\texttt{Inductive sig}\ \{X : \texttt{Type}\}\ (P : X \to \texttt{Prop}) : \boxed{\texttt{Type}} :=$

$\quad \mid \texttt{exist} : \forall x : X, P\ x \to \texttt{sig}\ P$ (also denoted $\{x : X \mid P\ x\}$)

$\texttt{Inductive sigT}\ \{X : \texttt{Type}\}\ (P : X \to \texttt{Type}) : \boxed{\texttt{Type}} :=$

$\quad \mid \texttt{existT} : \forall x : X, P\ x \to \texttt{sigT}\ P$ (also denoted $\{x : X\ \&\ P\ x\}$)

$\exists x : X, P\ x$ equivalent to $\texttt{inhabited}\{x : X \mid P\ x\}$

Unbounded minimization (sample OCaml code)

- Minimization of Boolean function f : `int` $\rightarrow$ `bool`
  - try f 0, f 1, f 2, ... until f n outputs `true`
  - if e.g. f 0 does not terminate, then minimization loops as well
- Implemented by this sample code:

```
let rec minimize_rec f n =
  match f n with
    | true  -> n
    | false -> minimize_rec f (1 + n)

let minimize f = minimize_rec f 0
```

- This codes *does not always* terminate, but Coq code must...

Terminating unbounded minimization (OCaml)

- How to ensure termination: decorate with a decreasing argument

  `let rec minimize_rec` $f\ n\ \boxed{H_n} =$

  `match` $f\ n$ `with`

  $|$ `true` $\rightarrow n$

  $|$ `false` $\rightarrow$ `minimize_rec` $f\ (1+n)\ \boxed{H_{1+n}}$

  `let minimize` $f =$ `minimize_rec` $f\ 0\ \boxed{H_0}$

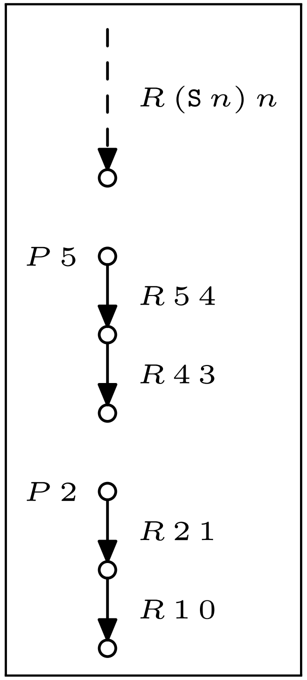
- Problems:
  - Termination input: non-informative proof $H_f : \exists n, f\ n = \texttt{true}$
  - How to obtain H_{1+n} from H_n s.t. H_{1+n} is *simpler* than H_n ?
  - How to build H_0 from $H_f : \exists n, f\ n = \texttt{true}$?
- Solution: H_n is $\boxed{\texttt{Acc}\ R\ n}$ for some rel. $R : \mathcal{N} \rightarrow \mathcal{N} \rightarrow \texttt{Prop}$

Non-informative existence as accessibility

$\texttt{Inductive Acc}\ \{X : \texttt{Type}\}\ (R : X \to X \to \texttt{Prop})\ (x : X) :=$

$\quad |\ \texttt{Acc\_intro} : (\forall y : X, R\ y\ x \to \texttt{Acc}\ R\ y) \to \texttt{Acc}\ R\ x$

- Accessibility $\texttt{Acc}\ R$ is the least R-hereditary predicate



- Let e.g. $P\ x$ iff $x = 2$ or $x = 5$
- Let $R\ n\ m$ iff $(n = \texttt{S}\ m) \wedge \neg P\ m$
- Then $\texttt{Acc}\ R\ 5$ because 5 has no R-antecedent
- $\texttt{Acc}\ R\ 4$ as $\texttt{Acc}\ R\ 5$ (5 only antecedent of 4)
- $\texttt{Acc}\ R\ 3$ as $\texttt{Acc}\ R\ 4$ (4 only antecedent of 3)
- $\texttt{Acc}\ R\ 2$ because 2 has no antecedent
- Then $\texttt{Acc}\ R\ 1$ and $\texttt{Acc}\ R\ 0$
- But $\neg\texttt{Acc}\ R\ i$ for $i \geq 6$

Well-founded unbounded minimization (1)

$\texttt{Variables}\quad (P : \mathcal{N} \to \texttt{Prop})\ \big(H_P : \forall n : \mathcal{N},\ \{P\ n\} + \{\neg P\ n\}\big)$

$\texttt{Let}\ \ R\ (n\ m : \mathcal{N}) \quad := \quad (n = 1 + m) \wedge \neg P\ m$

$\texttt{Let}\ \ \texttt{P\_Acc\_R} \quad : \quad \forall n : \mathcal{N},\ P\ n \to \texttt{Acc}\ R\ n$

$\texttt{Let}\ \ \texttt{Acc\_R\_dec} \quad : \quad \forall n : \mathcal{N},\ \texttt{Acc}\ R\ (1 + n) \to \texttt{Acc}\ R\ n$

$\texttt{Let}\ \ \texttt{Acc\_R\_zero} \quad : \quad \forall n : \mathcal{N},\ \texttt{Acc}\ R\ n \to \texttt{Acc}\ R\ 0$

$\texttt{Let}\ \ \texttt{ex\_P\_Acc\_R\_zero} \quad : \quad (\exists n : \mathcal{N}, P\ n) \to \texttt{Acc}\ R\ 0$

$\texttt{Let}\ \ \texttt{Acc\_R\_eq} \quad : \quad \forall n : \mathcal{N},\ \texttt{Acc}\ R\ n \iff \exists i : \mathcal{N}, n \leqslant i \wedge P\ i$

Well-founded unbounded minimization (2)

$\texttt{Let}\ \ R\ n\ m := (n = 1 + m) \wedge \neg P\ m$

$\texttt{Let}\ \ \texttt{Acc\_inv}\ (n : \mathcal{N})\ (H_n : \texttt{Acc}\ R\ n)\ (F : \neg P\ n) : \texttt{Acc}\ R\ (1 + n) :=$

$\quad \texttt{let}\ F' := \texttt{conj eq\_refl}\ F$

$\quad \texttt{in match}\ H_n\ \texttt{with Acc\_intro}\ \_\ H \mapsto H\ \_\ F'\ \texttt{end}$

$\texttt{Fixpoint}\ \ \texttt{Acc\_P}\ (n : \mathcal{N})\ (H_n : \texttt{Acc}\ R\ n) : \{x : \mathcal{N} \mid P\ x\} :=$

$\quad \texttt{match}\ H_P\ n\ \texttt{with}$

$\qquad \mid \texttt{left}\ T\ \ \mapsto \texttt{exist}\ \_\ n\ T$

$\qquad \mid \texttt{right}\ F \mapsto \texttt{Acc\_P}\ (1 + n)\ (\texttt{Acc\_inv}\ \_\ H_n\ F)$

$\quad \texttt{end}.$

Reification of decidable predicates

- For $P : \mathcal{N} \rightarrow \texttt{Prop}$ and $H_P : \forall n, \{P\,n\} + \{\neg P\,n\}$

$\texttt{Theorem nat\_reify} : (\exists n : \mathcal{N}, P\ n) \rightarrow \{n : \mathcal{N} \mid P\ n\}$

- Proof:
  - $\texttt{intros}\ H : \exists n, P\ n$, goal is now $\{n : \mathcal{N} \mid P\ n\}$
  - $\texttt{apply Acc\_P with}\ (n := 0)$, goal is now $\texttt{Acc}\ R\ 0 : \texttt{Prop}$
  - $\texttt{apply ex\_P\_Acc\_R\_zero}$, goal is now $\exists n : \mathcal{N}, P\ n$
  - $\texttt{assumption}$, goal solved by hypothesis H
- We also get the fully specified:

$\texttt{Theorem minimize} : (\exists n, P\ n) \rightarrow \{n \mid P\ n \wedge \forall m, P\ m \rightarrow n \leqslant m\}$

Reification of dec. and informative predicates

- Decidability for informative predicates $\boxed{P : \texttt{Type}}$

$$\texttt{decidable\_t}\ P := P + (P \rightarrow \texttt{False})$$

- For $P : \mathcal{N} \rightarrow \texttt{Type}$ and $H_P : \forall n, (P\ n) + (P\ n \rightarrow \texttt{False})$

$\texttt{Theorem nat\_reify\_t} : \big(\exists n : \mathcal{N}, \texttt{inhabited}(P\ n)\big) \rightarrow \big\{n : \mathcal{N}\ \&\ P\ n\big\}$

- Hypothesis $\exists n : \mathcal{N}, \texttt{inhabited}(P\ n)$ has no informative content
- It computes:
  - n (minimal) such that $P\ n$ is inhabited
  - but it also computes an inhabitant of (that) $P\ n$
- The proof is very similar to that of `nat_reify`

An inductive type for recursive algorithms

- X^n is the type of vectors on $X : \texttt{Type}$ of dimension $n : \mathcal{N}$
- $\mathcal{A}_k$ is a notation for $\texttt{recalg}\ (k : \mathcal{N})$
- $\texttt{recalg} : \mathcal{N} \to \texttt{Set}$ dependently defined by inductive rules:

$$\frac{n : \mathcal{N}}{\texttt{cst}_n : \mathcal{A}_0} \qquad \frac{}{\texttt{zero} : \mathcal{A}_1} \qquad \frac{}{\texttt{succ} : \mathcal{A}_1} \qquad \frac{p : \texttt{pos}\ k}{\texttt{proj}_p : \mathcal{A}_k}$$

$$\frac{f : \mathcal{A}_k \quad \vec{g} : \mathcal{A}_i^k}{\texttt{comp}\ f\ \vec{g} : \mathcal{A}_i} \qquad \frac{f : \mathcal{A}_k \quad g : \mathcal{A}_{2+k}}{\texttt{rec}\ f\ g : \mathcal{A}_{1+k}} \qquad \frac{f : \mathcal{A}_{1+k}}{\texttt{min}\ f : \mathcal{A}_k}$$

- Working with dependent types might involve some difficulties...

Beware fixpoint definitions are not compositional

```
Variable (P : forall k, recalg k -> Type)
    (Pcst : forall n, P (cst n)) (Pzero ....

Fixpoint recalg_rect k f { struct f } : P k f :=
  match f with
   | cst n     => Pcst n
   | zero      => Pzero
   | succ      => Psucc
   | proj p    => Pproj p
   | comp f gj => Pcomp [|f|] (fun p => [|vec_pos gj p|])
   | rec f g   => Prec [|f|] [|g|]
   | min f     => Pmin [|f|]
 end where "[| f |]" := (recalg_rect _ f).
```

Dependencies might involve type castings

- `eq_rect` maps a term of type $P\ i$ into $P\ j$ using a proof $e : i = j$
- Alternatively, use heterogeneous equality `JMeq` (John Major's eq.)
- Injection lemmas involve type castings:

$$\texttt{Fact ra\_comp\_inj}\ k\ k'\ i\ (f : \mathcal{A}_k)\ (f' : \mathcal{A}_{k'})\ (\vec{g} : \mathcal{A}_i^k)\ (\vec{g}' : \mathcal{A}_i^{k'}) :$$

$$\texttt{comp}\ f\ \vec{g} = \texttt{comp}\ f'\ \vec{g}' \to \exists e : k = k', \wedge \begin{cases} \texttt{eq\_rect}\ \_\ \_\ f\ \_\ e = f' \\ \texttt{eq\_rect}\ \_\ \_\ \vec{g}\ \_\ e = \vec{g}' \end{cases}$$

- These difficulties might be frightening for casual Coq users

Relational semantics for recursive algorithms

- We denote $[\![f]\!]$ for $\texttt{ra\_rel}\ k\ (f : \mathcal{A}_k) : \mathcal{N}^k \to \mathcal{N} \to \texttt{Prop}$
- $[\![f]\!]\ \vec{v}\ x$: the computation of f on input $\vec{v}$ halts and outputs x

$$[\![\texttt{cst}_n]\!]\ \_\ x \iff n = x \qquad [\![\texttt{zero}]\!]\ \_\ x \iff 0 = x$$

$$[\![\texttt{succ}]\!]\ \vec{v}\ x \iff 1 + \vec{v}_{\texttt{fst}} = x \qquad [\![\texttt{proj}_p]\!]\ \vec{v}\ x \iff \vec{v}_p = x$$

$$\begin{aligned}
[\![\texttt{comp}\ f\ \vec{g}]\!]\ \vec{v}\ x &\iff \exists \vec{w}, [\![f]\!]\ \vec{w}\ x \wedge \forall p, [\![\vec{g}_p]\!]\ \vec{v}\ \vec{w}_p \\
[\![\texttt{rec}\ f\ g]\!]\ (0\#\vec{v})\ x &\iff [\![f]\!]\ \vec{v}\ x \\
[\![\texttt{rec}\ f\ g]\!]\ (1+n\#\vec{v})\ x &\iff \exists y, [\![\texttt{rec}\ f\ g]\!]\ (n\#\vec{v})\ y \wedge [\![g]\!]\ (n\#y\#\vec{v})\ x \\
[\![\texttt{min}\ f]\!]\ \vec{v}\ x &\iff \exists \vec{w}, [\![f]\!]\ (x\#\vec{v})\ 0 \wedge \forall p : \texttt{pos}\ x, [\![f]\!]\ (\overline{p}\#\vec{v})\ (1 + \vec{w}_p)
\end{aligned}$$

- A simple exercise (given a good recursion principle for $\mathcal{A}_k$;-)
- But $x \mapsto [\![f]\!]\ \vec{v}\ x$ is not a decidable relation.

Big-step semantics for recursive algorithms

- We denote $[f;\vec v] \leadsto x$ for $\texttt{ra\_bs}\ k\ f\ \vec v\ x : \texttt{Prop}$ (or $\texttt{Type}$...)
- Same meaning as $[\![f]\!]\ \vec v\ x$ but defined as an inductive predicate

$$\frac{}{[\texttt{cst}_n;\vec v] \leadsto n} \qquad \frac{}{[\texttt{zero};\vec v] \leadsto 0} \qquad \frac{}{[\texttt{succ};\vec v] \leadsto 1 + \vec v_{\texttt{fst}}} \qquad \frac{}{[\texttt{proj}_p;\vec v] \leadsto \vec v_p}$$

$$\frac{[\texttt{rec}\ f\ g; n\#\vec v] \leadsto y \qquad [g; n\#y\#\vec v] \leadsto x}{[\texttt{rec}\ f\ g; 1 + n\#\vec v] \leadsto x} \qquad \frac{[f;\vec w] \leadsto x \qquad \forall p, [\vec g_p;\vec v] \leadsto \vec w_p}{[\texttt{comp}\ f\ \vec g;\vec v] \leadsto x}$$

$$\frac{[f;\vec v] \leadsto x}{[\texttt{rec}\ f\ g; 0\#\vec v] \leadsto x} \qquad \frac{[f; x\#\vec v] \leadsto 0 \qquad \forall p : \texttt{pos}\ x, [f;\overline{p}\#\vec v] \leadsto 1 + \vec w_p}{[\texttt{min}\ f;\vec v] \leadsto x}$$

- Easy (intuitive ?) definition
  - $\texttt{ra\_bs} : \forall k, \mathcal{A}_k \to \mathcal{N}^k \to \mathcal{N} \to \texttt{Prop},\ [\![f]\!]\ \vec v\ x \iff [f;\vec v] \leadsto x$
  - $\texttt{ra\_bs} : \forall k, \mathcal{A}_k \to \mathcal{N}^k \to \mathcal{N} \to \texttt{Type}$ is a type of computations
- Let us transform $\texttt{ra\_bs}$ into a decidable predicate

Cost aware big-step semantics

- We denote $[f;\vec{v}] \dashv\!\!\langle \alpha \rangle\!\rangle\ x$ for $\texttt{ra\_ca}\ k\ f\ \vec{v}\ \alpha\ x : \texttt{Prop}$
- α represents the *cost* (or size) of the computation

$$\frac{}{[\texttt{cst}_n;\vec{v}] \dashv\!\langle 1 \rangle\!\rangle\ n} \qquad \frac{}{[\texttt{zero};\vec{v}] \dashv\!\langle 1 \rangle\!\rangle\ 0} \qquad \frac{}{[\texttt{succ};\vec{v}] \dashv\!\langle 1 \rangle\!\rangle\ 1 + \vec{v}_{\texttt{fst}}} \qquad \frac{}{[\texttt{proj}_p;\vec{v}] \dashv\!\langle 1 \rangle\!\rangle\ \vec{v}_p}$$

$$\frac{[\texttt{rec}\ f\ g; n\#\vec{v}] \dashv\!\langle \alpha \rangle\!\rangle\ y \qquad [g; n\#y\#\vec{v}] \dashv\!\langle \beta \rangle\!\rangle\ x}{[\texttt{rec}\ f\ g; 1+n\#\vec{v}] \dashv\!\langle 1+\alpha+\beta \rangle\!\rangle\ x} \qquad \frac{[f;\vec{w}] \dashv\!\langle \alpha \rangle\!\rangle\ x \qquad \forall p, [\vec{g}_p;\vec{v}] \dashv\!\langle \vec{\beta}_p \rangle\!\rangle\ \vec{w}_p}{[\texttt{comp}\ f\ \vec{g};\vec{v}] \dashv\!\langle 1+\alpha+\Sigma\vec{\beta} \rangle\!\rangle\ x}$$

$$\frac{[f;\vec{v}] \dashv\!\langle \alpha \rangle\!\rangle\ x}{[\texttt{rec}\ f\ g; 0\#\vec{v}] \dashv\!\langle 1+\alpha \rangle\!\rangle\ x} \qquad \frac{[f; x\#\vec{v}] \dashv\!\langle \alpha \rangle\!\rangle\ 0 \qquad \forall p : \texttt{pos}\ x, [f; \overline{p}\#\vec{v}] \dashv\!\langle \vec{\beta}_p \rangle\!\rangle\ 1+\vec{w}_p}{[\texttt{min}\ f;\vec{v}] \dashv\!\langle 1+\alpha+\Sigma\vec{\beta} \rangle\!\rangle\ x}$$

- for $\texttt{ra\_ca}$, we have $[\![f]\!]\ \vec{v}\ x \Longleftrightarrow \exists \alpha : \mathcal{N}, [f;\vec{v}] \dashv\!\langle \alpha \rangle\!\rangle\ x$
- $x \mapsto [f;\vec{v}] \dashv\!\langle \alpha \rangle\!\rangle\ x$ is a decidable predicate:
  - from α, recover comp. $[f;\vec{v}] \rightsquigarrow x : \texttt{Type}$ by *prim. rec. means*

Properties of cost aware semantics

- Inversion lemmas:

$\texttt{Lemma ra\_ca\_rec\_S\_inv}\ (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (g : \mathcal{A}_{2+k})\ \vec{v}\ n\ \gamma\ x :$

$$[\texttt{rec}\,f\,g; 1 + n\#\vec{v}] \dashv\!\![\gamma\rangle\!\rangle\ x \to \exists y\ \alpha\ \beta, \wedge \begin{cases} \gamma = 1 + \alpha + \beta \\ [\texttt{rec}\,f\,g; n\#\vec{v}] \dashv\!\![\alpha\rangle\!\rangle\ y \\ [g; n\#y\#\vec{v}] \dashv\!\![\beta\rangle\!\rangle\ x \end{cases}$$

- Functionality:

$\texttt{Theorem ra\_ca\_fun}\ (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (\vec{v} : \mathcal{N}^k)\ (\alpha\,\beta\,x\,y : \mathcal{N}) :$

$$[f; \vec{v}] \dashv\!\![\alpha\rangle\!\rangle\ x \to [f; \vec{v}] \dashv\!\![\beta\rangle\!\rangle\ y \to \alpha = \beta \wedge x = y$$

- Decidability:

$\texttt{Theorem ra\_ca\_decidable\_t}\ (k : \mathcal{N})\ (f : \mathcal{A}_k)\ (\vec{v} : \mathcal{N}^k)\ (\alpha : \mathcal{N}) :$

$$\{x \mid [f; \vec{v}] \dashv\!\![\alpha\rangle\!\rangle\ x\} + \{x \mid [f; \vec{v}] \dashv\!\![\alpha\rangle\!\rangle\ x\} \to \texttt{False}$$

Typing total recursive functions

- For $f : \mathcal{A}_k$ and $\vec{v} : \mathcal{N}^k$ fixed, f terminates on $\vec{v}$ iff:
  - $\exists x, [\![f]\!]\ \vec{v}\ x$
  - $\exists x \exists \alpha, [f;\vec{v}] \dashv\!\alpha\rangle\!\rangle\ x$
  - $\exists \alpha \exists x, [f;\vec{v}] \dashv\!\alpha\rangle\!\rangle\ x$
  - $\exists \alpha,$ `inhabited` $\{x \mid [f;\vec{v}] \dashv\!\alpha\rangle\!\rangle\ x\}$
- For any α, the type $\{x \mid [f;\vec{v}] \dashv\!\alpha\rangle\!\rangle\ x\}$ is decidable:
  - `nat_reify_t` computes $\{\alpha : \mathcal{N}\ \&\ \{x : \mathcal{N} \mid [f;\vec{v}] \dashv\!\alpha\rangle\!\rangle\ x\}\}$
  - from which we extract x s.t. $[\![f]\!]\ \vec{v}\ x$

`Theorem Coq_is_total` $(k : \mathcal{N})\ (f : \mathcal{A}_k) :$

$$\big(\forall \vec{v} : \mathcal{N}^k, \exists x : \mathcal{N}, [\![f]\!]\ \vec{v}\ x\big) \to \big\{t : \mathcal{N}^k \to \mathcal{N} \ \big|\ \forall \vec{v} : \mathcal{N}^k, [\![f]\!]\ \vec{v}\ (t\ \vec{v})\big\}$$

Other applications: reifying undecidable predicates

- Normal forms (typically λ-calculus)
  - for $T : \texttt{Type}$, $R : T \to T \to \texttt{Prop}$
  - finitary: $\forall t : T, \{l : \texttt{list}\ X \mid \forall x, R\ t\ x \iff \texttt{In}\ x\ l\}$
  - with $\texttt{normal\_form}\ t\ n := (\forall x, \neg R\ n\ x) \wedge R^\star\ t\ n$
  - we have: $\forall t, (\exists n, \texttt{normal\_form}\ t\ n) \to \{n \mid \texttt{normal\_form}\ t\ n\}$
- From cut-admissibility to cut-elimination

$$\forall s\ (p : \texttt{proof}\ s), (\exists q : \texttt{proof}\ s, \texttt{cut\_free}\ q) \to \{q : \texttt{proof}\ s \mid \texttt{cut\_free}\ q\}$$

- Recursively enumerable predicates (of the form $\vec{v} \mapsto [\![f]\!]\ \vec{v}\ 0$)

$$\forall (k : \mathcal{N})\ (f : \mathcal{A}_k),\ (\exists \vec{v} : \mathcal{N}^k, [\![f]\!]\ \vec{v}\ 0) \to \{\vec{v} : \mathcal{N}^k \mid [\![f]\!]\ \vec{v}\ 0\}$$

Conclusion

- Mechanization of the Turing completeness of Coq
  - without using any (extra) axiom
  - by implementing reification of decidable predicates over `nat`
- Coq has a kind of unbounded minimization
  - provided the predicate can be informatively decided
  - and there is a non-informative inhabitation proof
- Kleene's T predicate replaced with cost aware big-step semantics
  - avoid small-step semantics and encodings
  - avoid compiler correctedness
  - show decidability of cost aware big-step semantics
- Reification extended to some undecidable predicates as well