A certifying extraction with time bounds from Coq to call-by-value λ-calculus

Yannick Forster Fabian Kunze

Germany

ITP 2019
September 12

Motivation

Formalising computability theory:

- Functions definable in constructive type theory are computable, but this is not provable *inside* the theory
- Explicit model of computation needed for negative results or complexity theory
- We use: call-by-value λ-calculus "L"
- Computability provable for every *concrete* function defined in Coq
- But computability proofs tedious and repetitive!

We provide a framework that automates[1] this *extraction*

[1] for ML-like subset of Coq

Idea

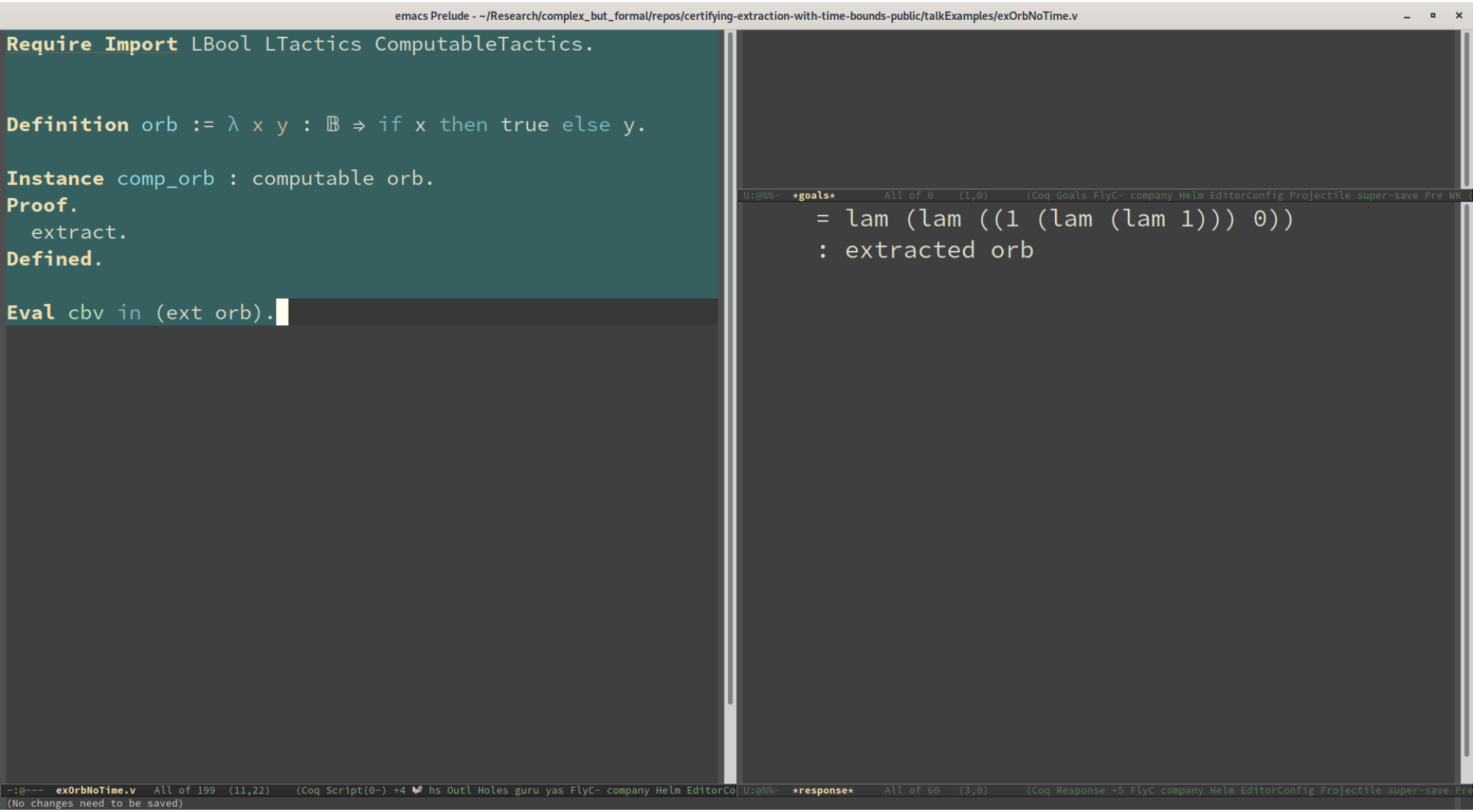
Call-by-Value lambda calculus (Plotkin [1975], Forster and Smolka [2017]):

$$s, t \quad ::= \quad x \quad | \quad \lambda s \quad | \quad st$$

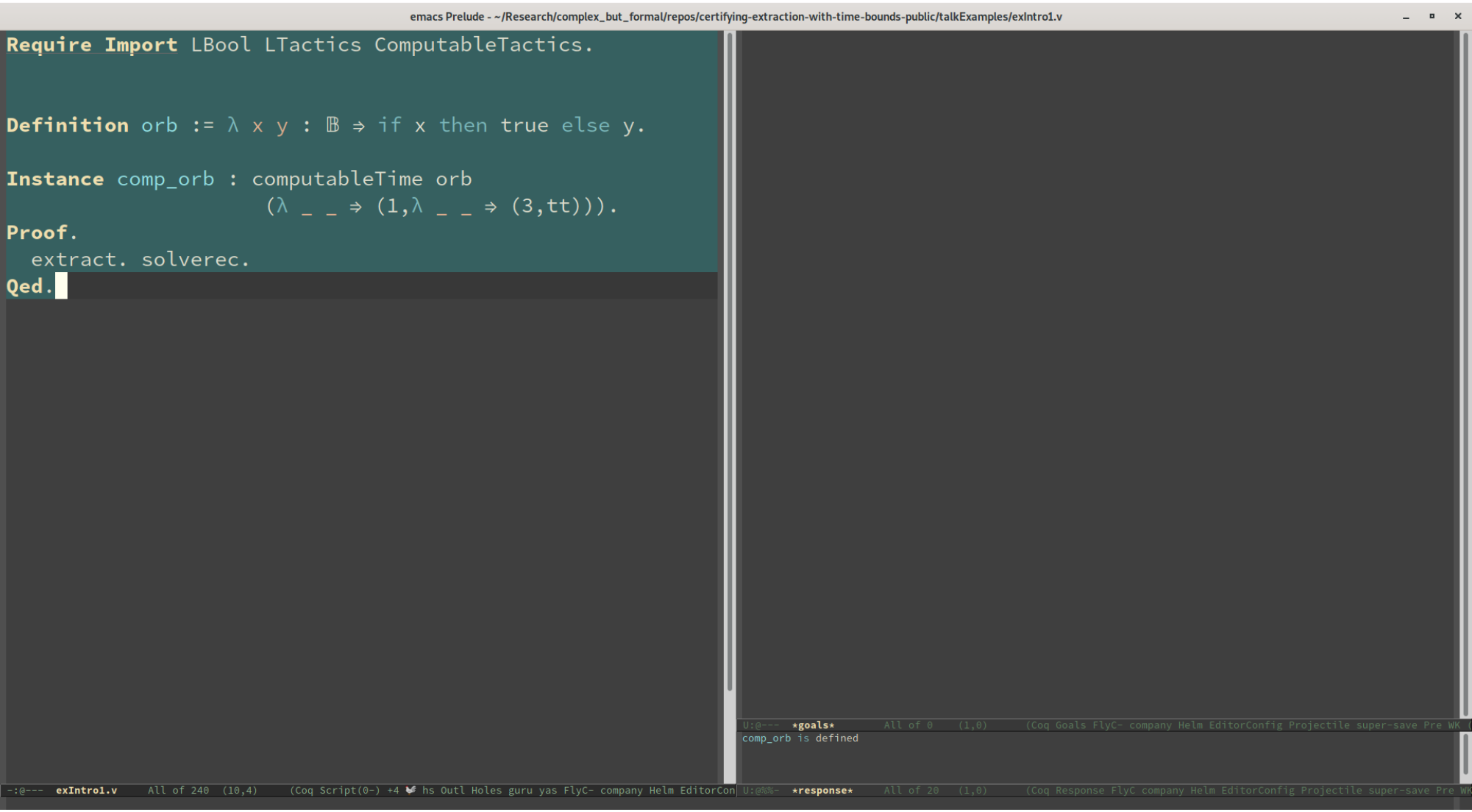
- For each concrete function $f : X_1 \to \ldots \to X_n$ over ML-like data types,
- find a λ-term t_f and a time complexity τ_f such that
- for all $\vec{x}$, t_f $(\mathtt{enc}\,\vec{x})$ reduces to $\mathtt{enc}(f\,\vec{x})$
- within $\tau_f\,\vec{x}$ many beta-reduction steps (Accattoli and Lago [2016]).

Simple, syntax directed extraction process.

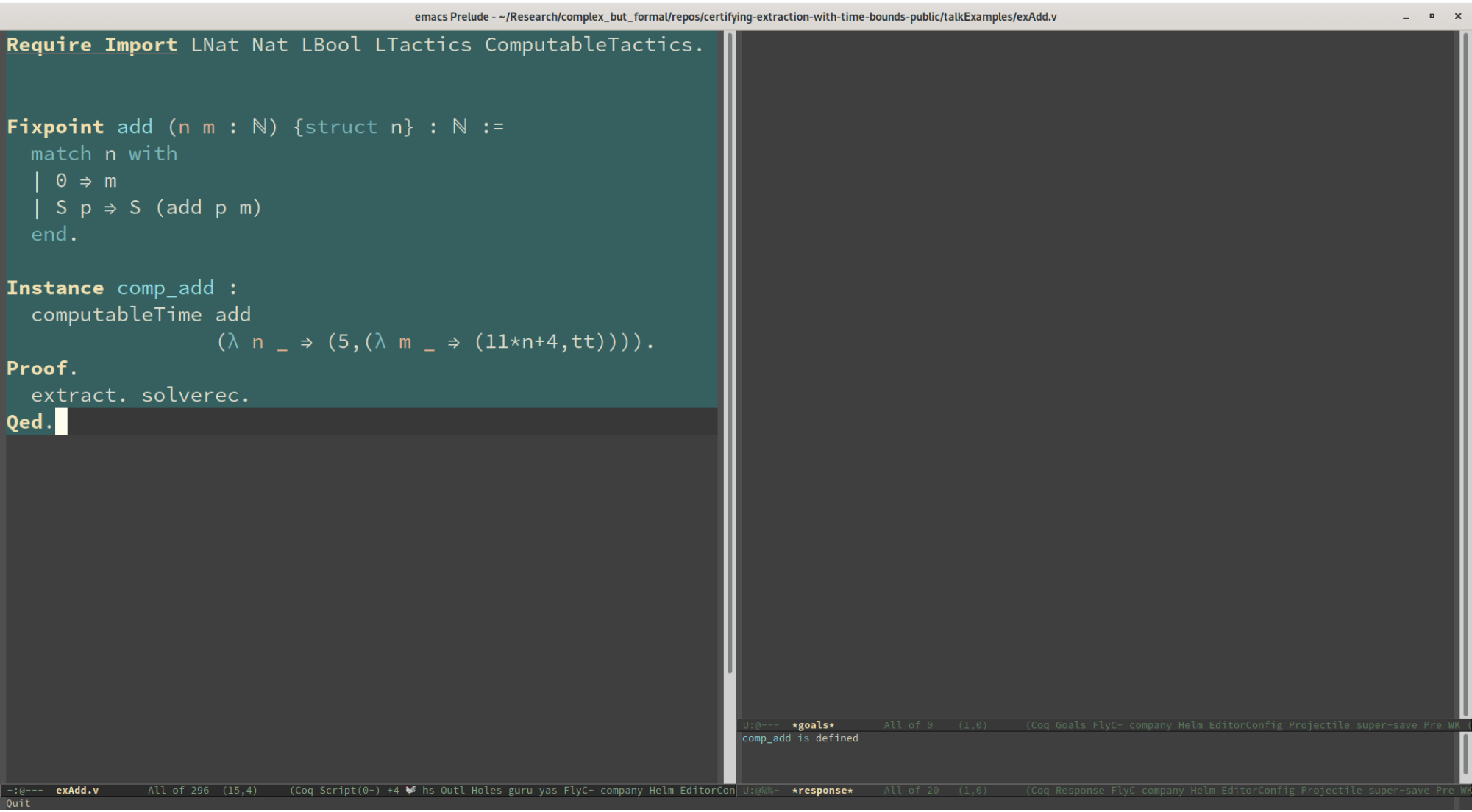
Example


emacs Prelude - ~/Research/complex_but_formal/repos/certifying-extraction-with-time-bounds-public/talkExamples/exOrbNoTime.v
Require Import LBool LTactics ComputableTactics.
Definition orb := λ x y : 𝔹 ⇒ if x then true else y.
Instance comp_orb : computable orb.
Proof.
extract.
Defined.
Eval cbv in (ext orb).
goals
= lam (lam ((1 (lam (lam 1))) 0))
: extracted orb
exOrbNoTime.v
response
(No changes need to be saved)

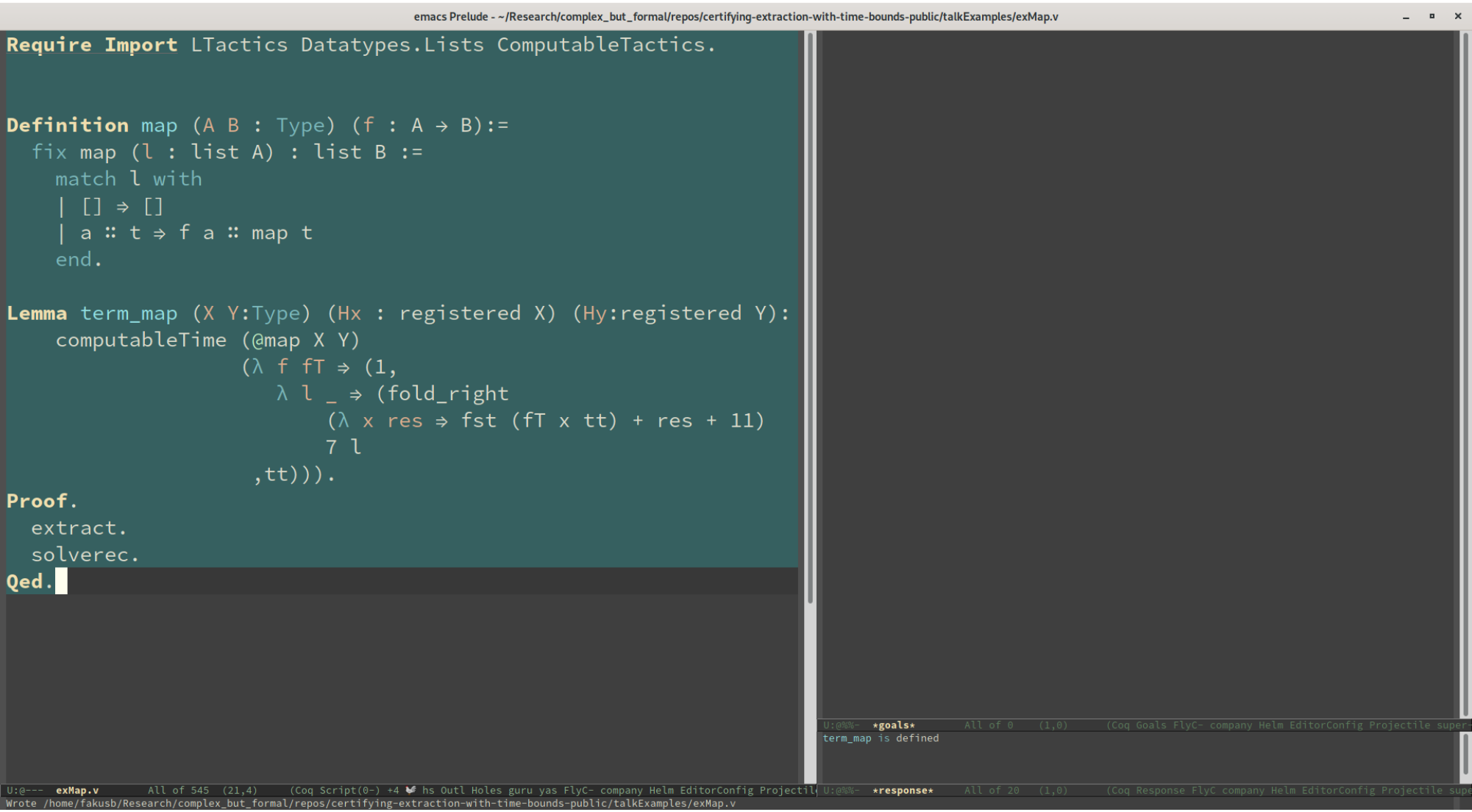
Example with time



Example with recursive function



Example with higher order function



Outline

- What *is* correctness?
- How to extract terms?
- How to prove those extractions correct?
- Case studies:
  - Self-interpreter for L
  - Enumerator for diophantine equations ($\rightarrow$ Reduction to L)
  - Turing machine interpreter

What is Correctness?

Encoding values: Scott encoding

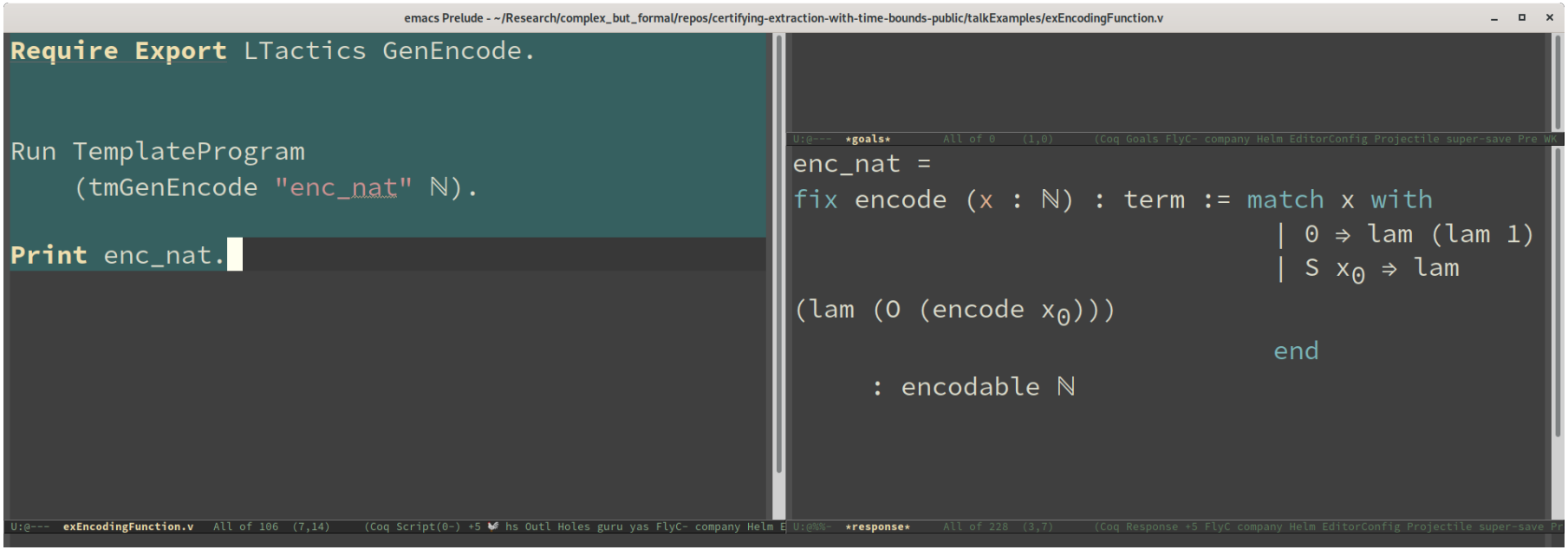
Inductive values encoded as functions (corresponding to their `match`):

`true`	$\lambda tf.\ t$
`false`	$\lambda tf.\ f$
0	$\lambda sz.z$
1	$\lambda sz.s(\lambda sz.z)$
S	$\lambda asz.sa$

$\rightarrow$ Encoding follows from definition of inductive data type

Generating encoding functions

Mechanically derivable[2] using Template Coq, part of MetaCoq (Sozeau et al. [2019]):



Stored in Typeclass, with properties (e.g. injectivity) proven by `Ltac`.

[2] for all ML-like (non-dependent, non-mutual) inductive types

Correctness predicate

When does a (λ-)term t_f compute a (Coq-)function f?
Example: $t_{\texttt{orb}} \sim \texttt{orb}$:

$$\forall xy : \mathbb{B}, t_{\texttt{orb}}\ (\texttt{enc}\, x)\ (\texttt{enc}\, y) \succ^* \texttt{enc}(\texttt{orb}\ x\ y)$$

Tool: Logical relation!

Correctness predicate

When does a (λ-)term t_f compute a (Coq-)function f?
Example: $t_{\texttt{orb}} \sim \texttt{orb}$:

$$\forall xy : \mathbb{B}, t_{\texttt{orb}}\ (\texttt{enc}x)\ (\texttt{enc}y) \succ^* \texttt{enc}(\texttt{orb}\ x\ y)$$

Tool: Logical relation!

$$\frac{}{\texttt{enc}_A a \sim a} \quad (\text{for } a : A)$$

$$\frac{\forall (a : A)(t_a : \mathbf{T}).\ t_a \sim a \to \quad t_f\, t_a \quad \sim fa}{t_f \sim f} \quad (\text{for } f : A \to B)$$

Correctness predicate

When does a (λ-)term t_f compute a (Coq-)function f?
Example: $t_{\texttt{orb}} \sim \texttt{orb}$:

$$\forall xy : \mathbb{B}, t_{\texttt{orb}}\ (\texttt{enc}\, x)\ (\texttt{enc}\, y) \succ^* \texttt{enc}(\texttt{orb}\ x\ y)$$

Tool: Logical relation!

$$\frac{}{\texttt{enc}_A a \sim a} \quad (\text{for } a : A)$$

$$\frac{\begin{array}{c} t_f \text{ is closed value} \wedge \\ \forall (a : A)(t_a : \mathbf{T}).\ t_a \sim a \to \Sigma (v : \mathbf{T}).\ t_f\, t_a \succ^* v \wedge v \sim fa \end{array}}{t_f \sim f} \quad (\text{for } f : A \to B)$$

Correctness predicate

When does a (λ-)term t_f compute a (Coq-)function f?
Example: $t_{\texttt{orb}} \sim \texttt{orb}$:

$$\forall xy : \mathbb{B}, t_{\texttt{orb}}\ (\texttt{enc}x)\ (\texttt{enc}y) \succ^* \texttt{enc}(\texttt{orb}\ x\ y)$$

Tool: Logical relation!

$$\frac{}{\texttt{enc}_A a \sim a} \quad (\text{for } a : A)$$

$$\frac{t_f \text{ is closed value} \wedge \quad \forall (a : A)(t_a : \mathbf{T}).\ t_a \sim a \to \Sigma(v : \mathbf{T}).\ t_f\, t_a \succ^* v \wedge v \sim fa}{t_f \sim f} \quad (\text{for } f : A \to B)$$

Problem: Not strictly positive.
Solution: Recursion on Type...?

Correctness predicate (2)

```
Inductive 𝒯 : Type → Type :=
  𝒯_base A `{registered A} : 𝒯 A
| 𝒯_arr A B (τ_1 : 𝒯 A) (τ_2 : 𝒯 B) : 𝒯 (A → B).

Fixpoint computes {A} (τ : 𝒯 A): A → T → Type :=
  match τ with
    𝒯_base ⇒ fun x xInt ⇒ (xInt = enc x)
  | @𝒯_arr A B τ_1 τ_2 ⇒
    fun f t_f ⇒
      proc t_f * ∀ (a : A) t_a,
        computes τ_1 a t_a
        → {v : term & (t_f t_a >* v) * computes τ_2 (f a) v}
  end%type.
```

Complexity functions

Describe number of steps dependent on input.
Example:

```
                                                orb true false
                  (fun x y : bool ⇒ if x then true else y) true false
                     (fun y : bool ⇒ if true then true else y) false
                                    (if true then true else false)
                                                             true
```

⇒ Time described by function

$$\tau_{\texttt{orb}} : \mathbb{B} \to \mathbb{N} \times (\mathbb{B} \to \mathbb{N}) := \lambda x.(1, \lambda y.2)$$

Complexity functions (2)

- Recursive functions:

$$\tau_+ : \mathbb{N} \to \mathbb{N} \times (\mathbb{N} \to \mathbb{N}) := \lambda m.(c_1, \lambda n.m \cdot c_2 + c_3)$$

- Higher-order functions: Type for time-complexity functions actually more involved, see paper for full truth.
- Correctness predicate easily extended with time-complexity functions:

$$t_f \sim^{\tau_f} f$$

Extracting Functions

Extracting functions

- Variables translate to variables according to environment
- Abstraction translates to lambda
- Fixpoints translate to fixed-point combinator ρ

```
Fixpoint extract (env : nat → nat) (s : Ast.term) (fuel : nat) :
    TemplateMonad T :=
  match fuel with 0 ⇒ tmFail "out of fuel" | S fuel ⇒
  match s with
    Ast.tRel n ⇒ ret (var (env n))
  | Ast.tLambda _ _ s ⇒
    t ← extract (↑ env) s fuel ;;
      ret (lam t)
  | Ast.tFix [Ast.mkdef _ _ _ s _] _ ⇒
    t ← extract (⇑ env) s fuel ;;
      ret (ρ (lam t))
```

Extracting functions (2)

- Applications are application
- Extractions of constants get reused using `Typeclasses`

```
| Ast.tApp s R ⇒
   params ← tmDependentArgs s;;
   if params =? 0 then
     t ← extract env s fuel;;
     monad_fold_left (fun t1 s2 ⇒ t2 ← extract env s2 fuel ;; ret (app t1
      t2)) R t
   else
     let (P, L) := (firstn params R,skipn params R)  in
     s' ← tmEval cbv (Ast.tApp s P);;
     a ← tmUnquote s' ;;
     a' ← tmEval cbn (my_projT2 a);;
     nm ← (tmEval cbv (String.append (name_of s) "_term")≫=tmFreshName);;
     i ← tmTryInfer nm (Some cbn) (extracted a') ;;
     let t := (@int_ext _ _ i) in
     monad_fold_left (fun t1 s2 ⇒ t2 ← extract env s2 fuel ;; ret (app t1
      t2)) L t
```

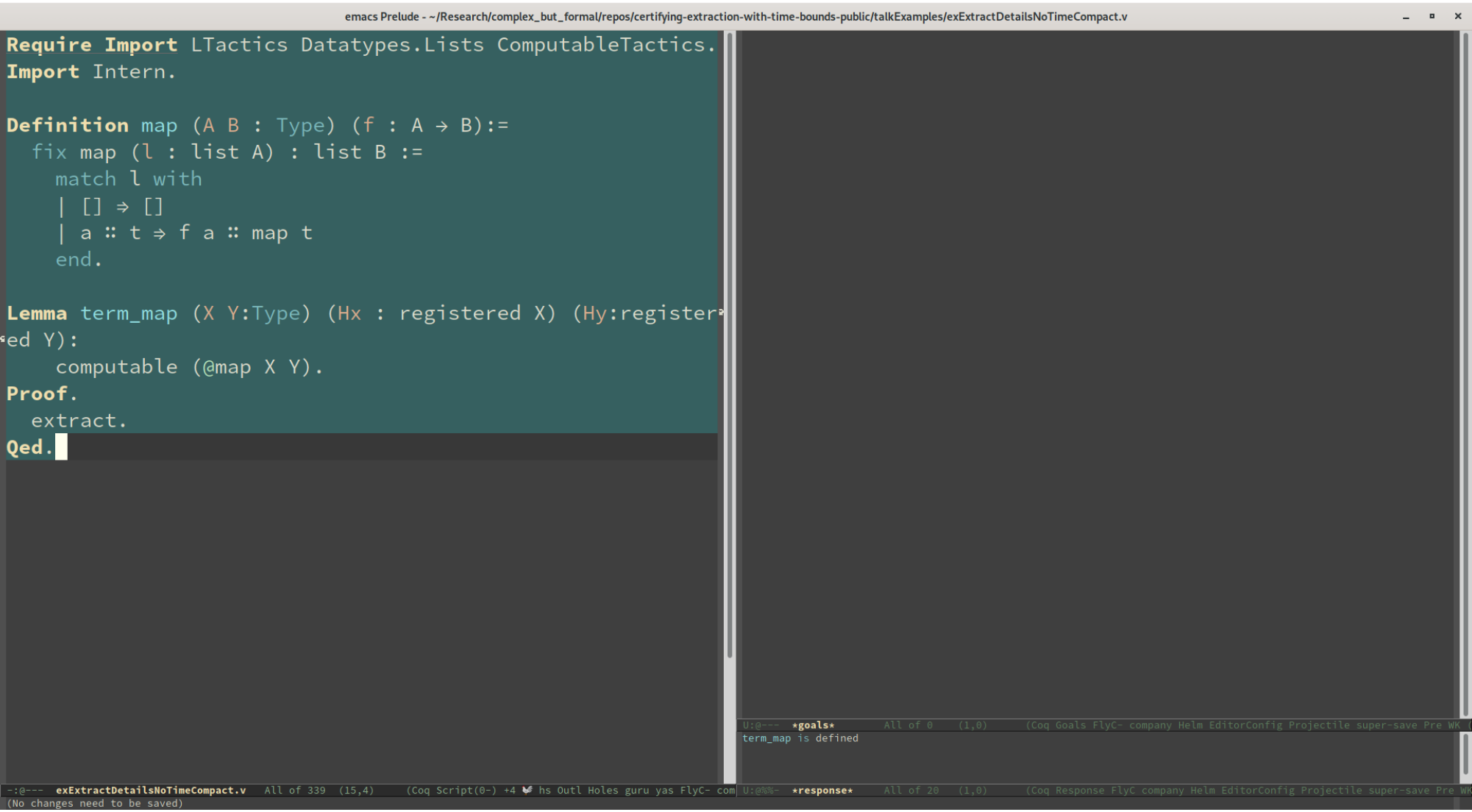
Proving correctness

Proving correctness

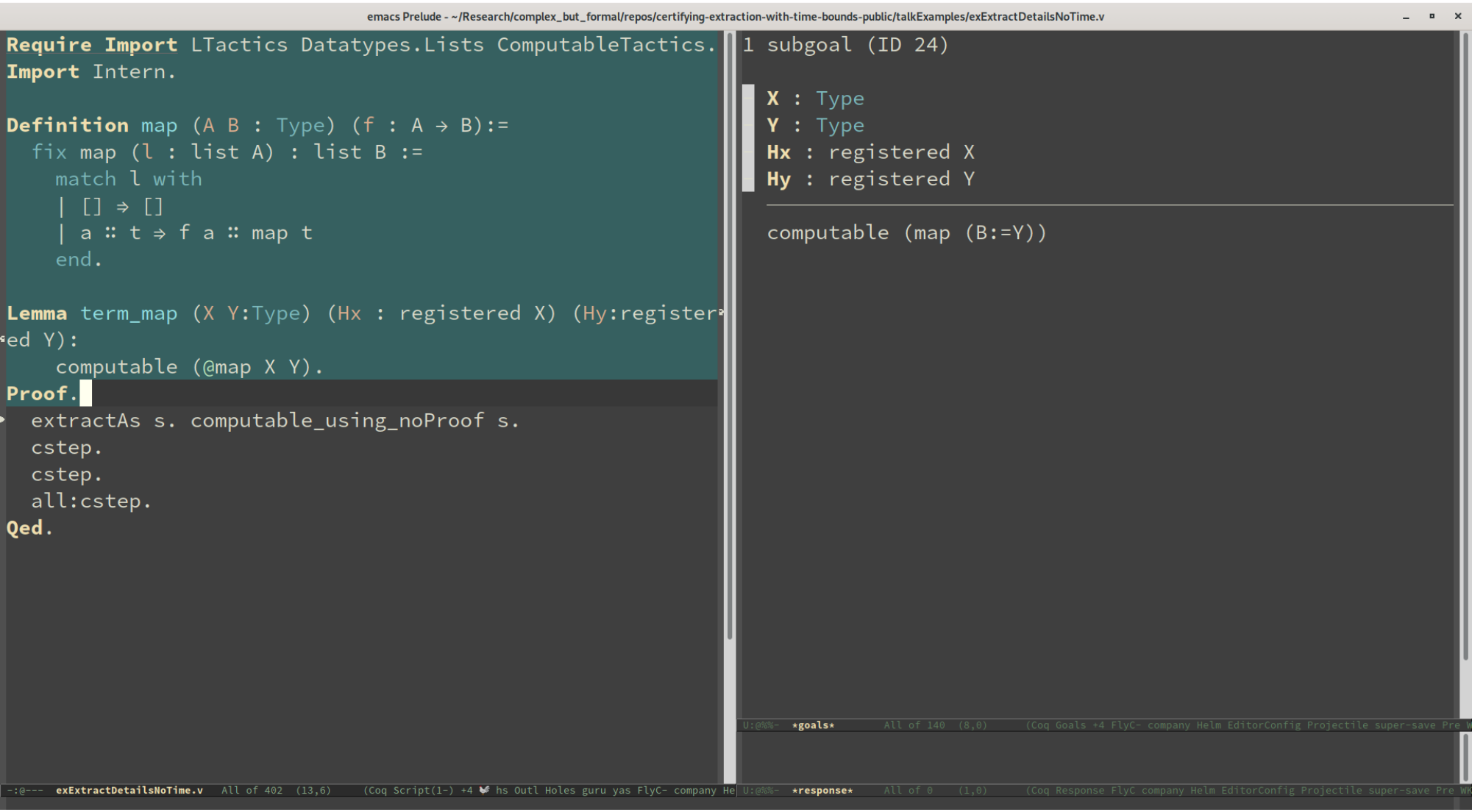
Each extracted term is certified using `Ltac`:

- Tactic reducing λ-terms keeping track of number of steps
- Show correctness by following same case distinctions/recursions as used in the extracted function

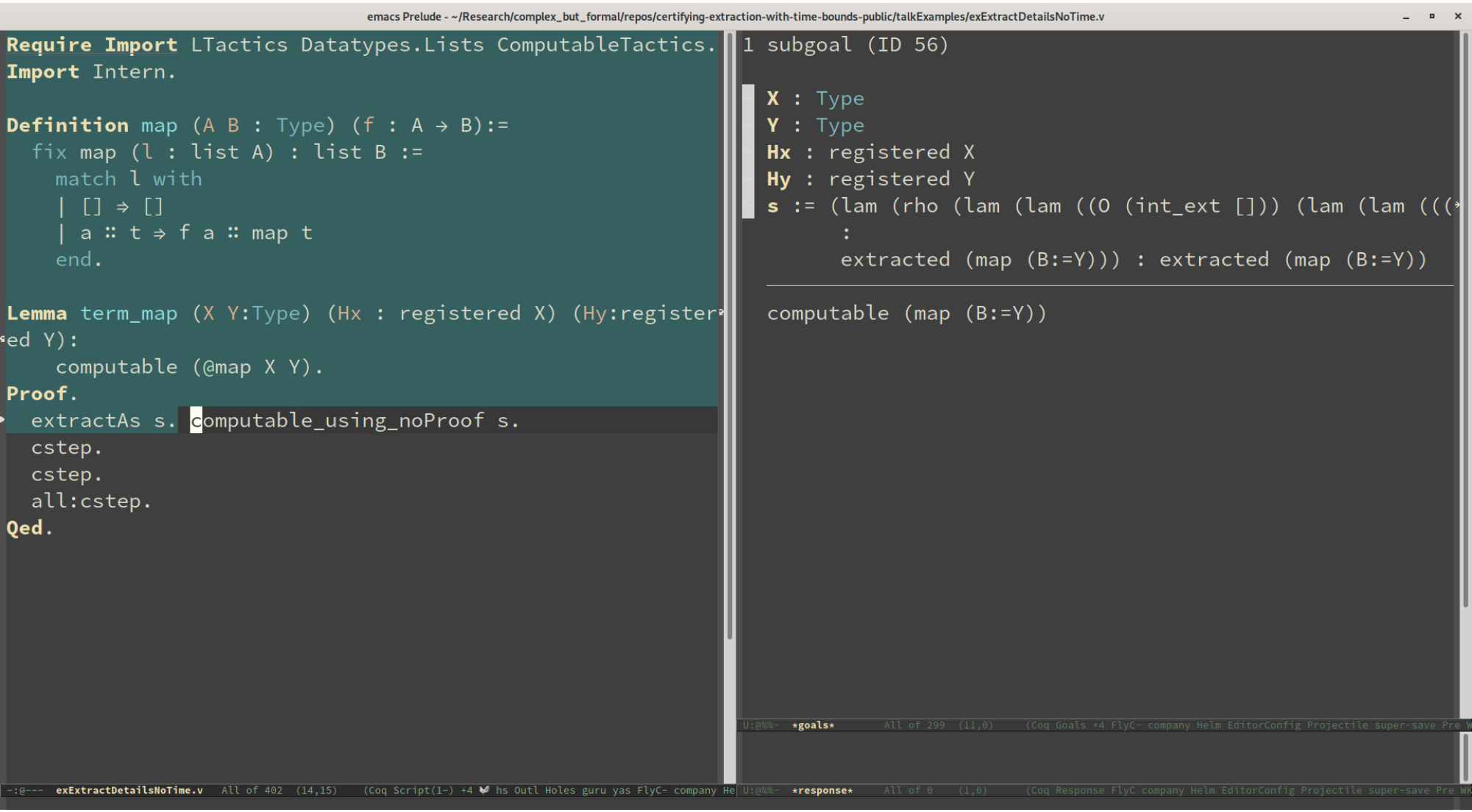
Proving correctness, example



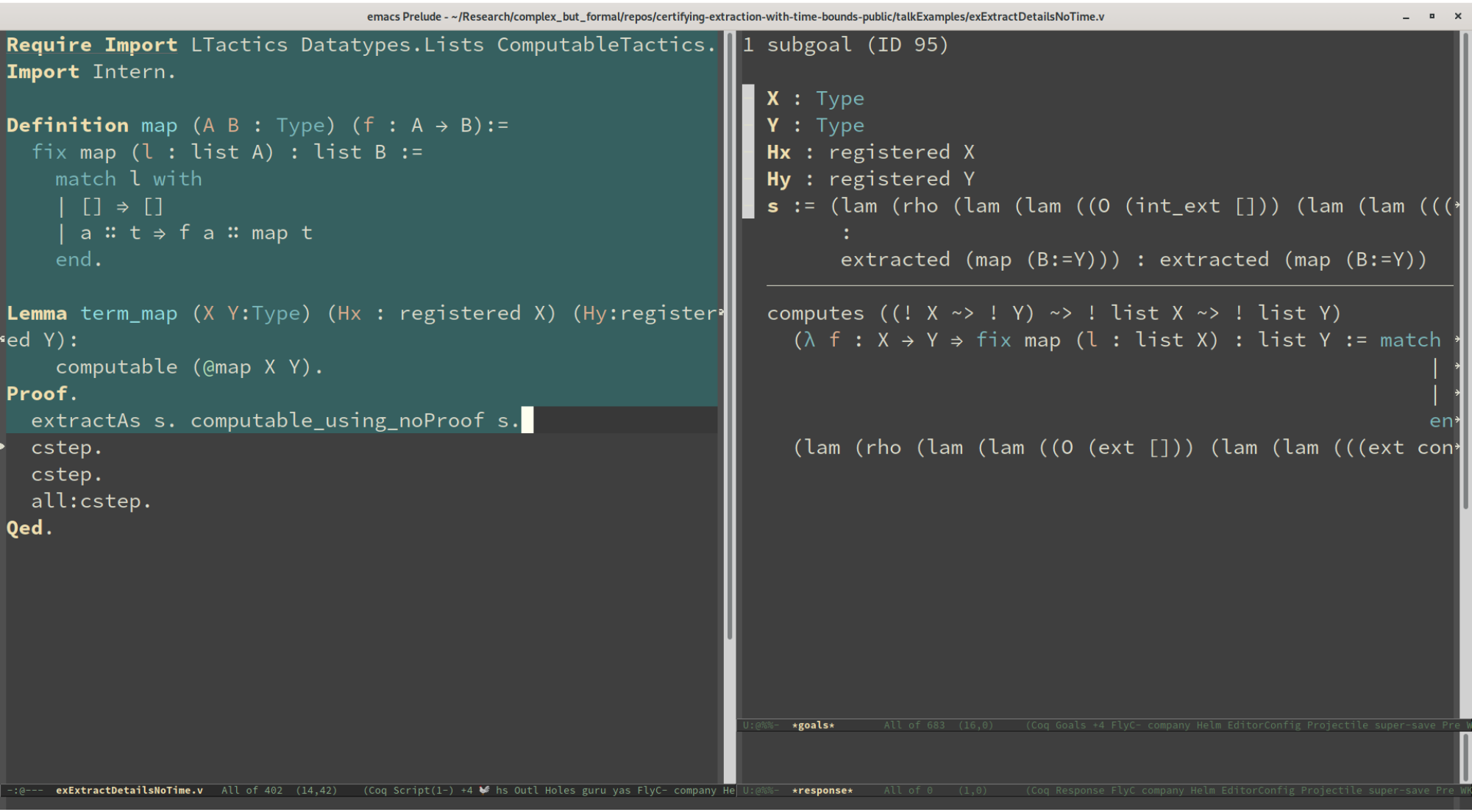
Proving correctness, example



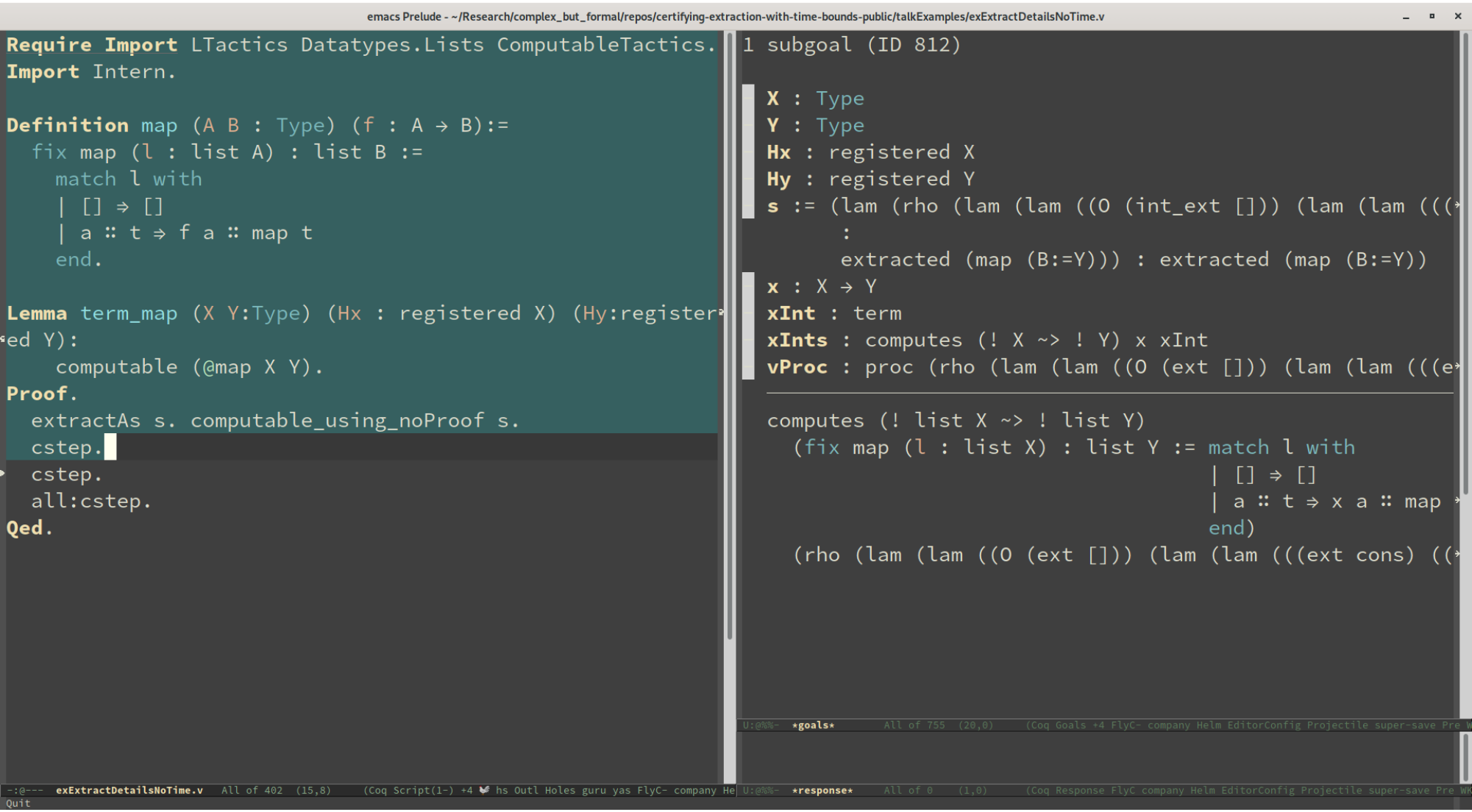
Proving correctness, example



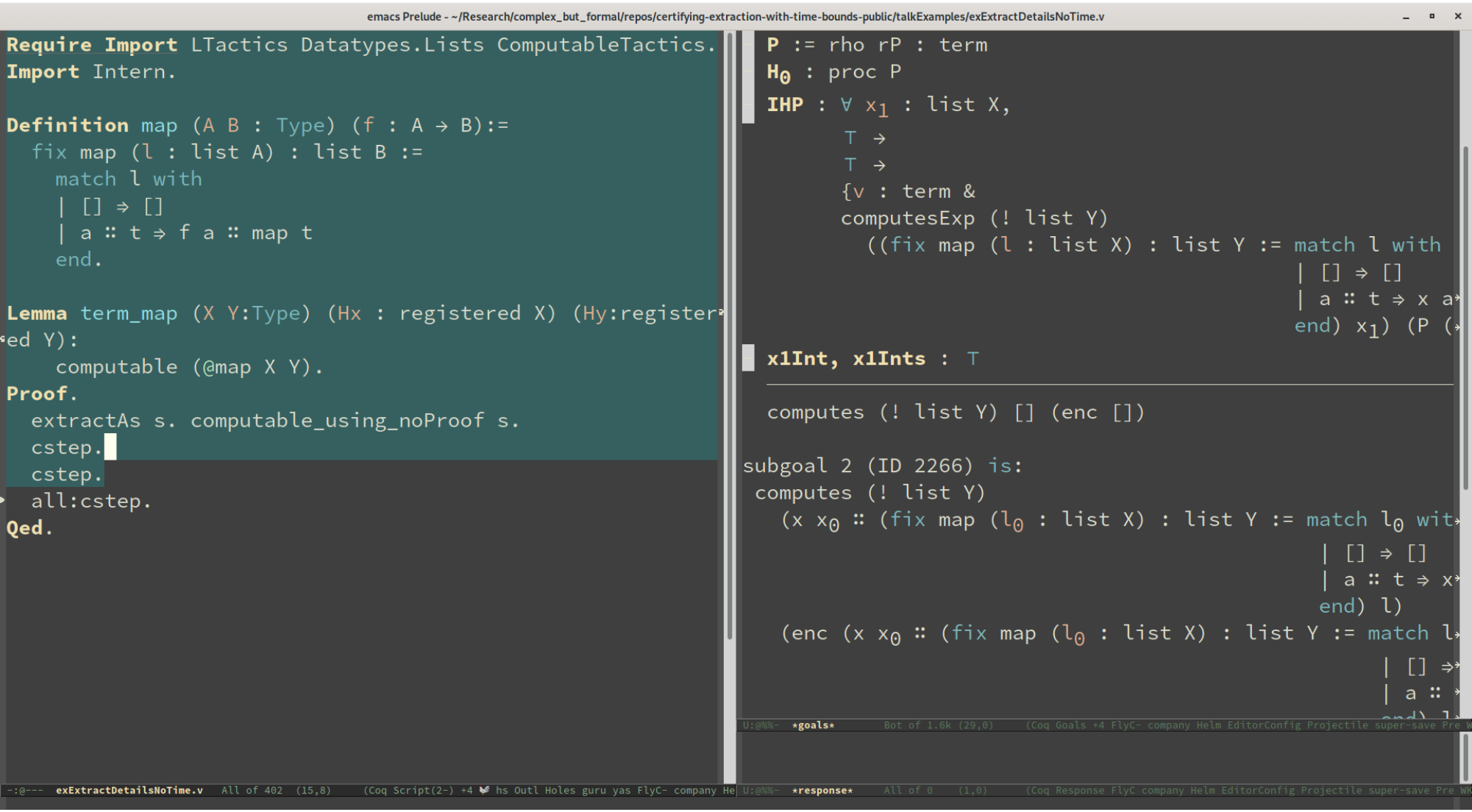
Proving correctness, example



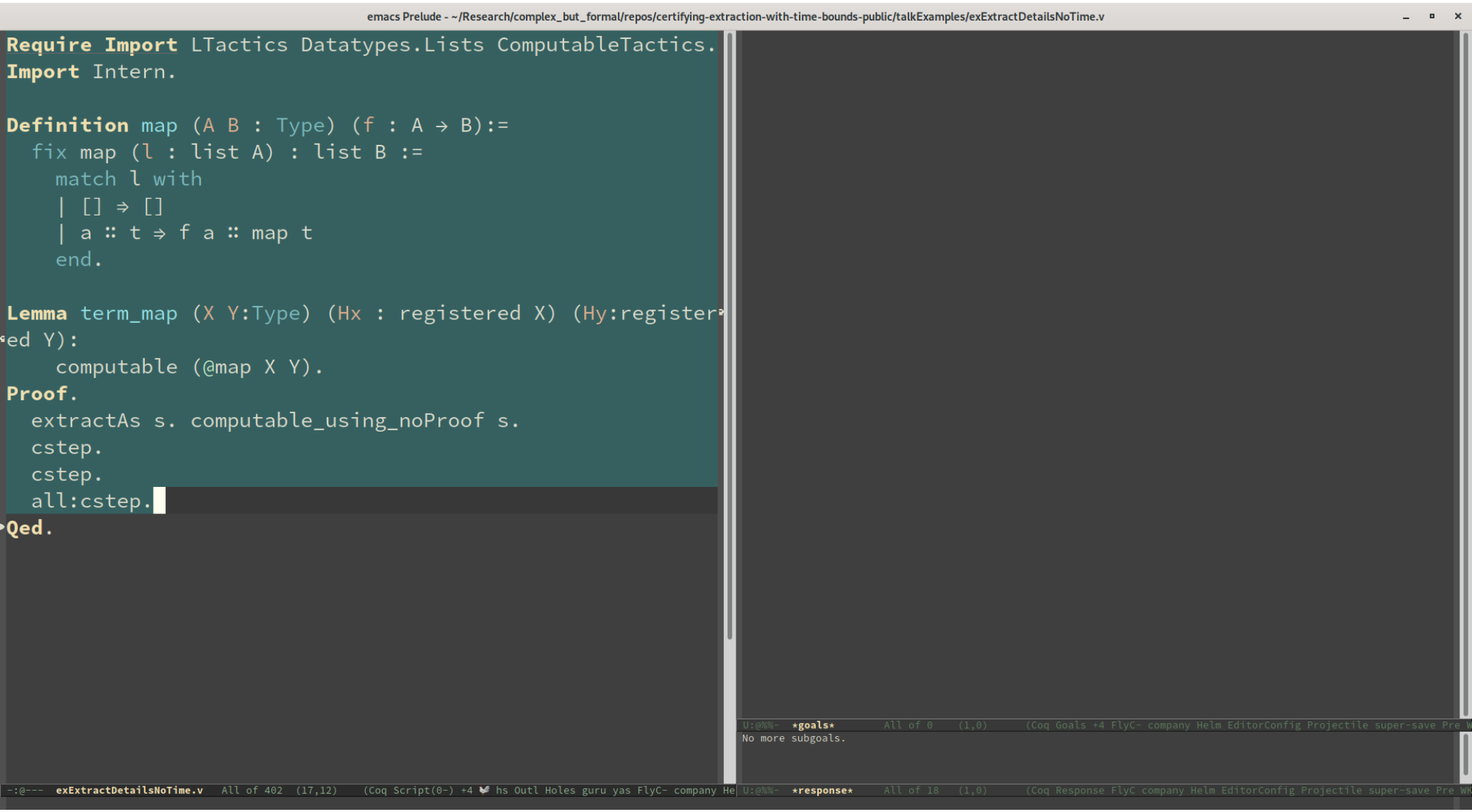
Proving correctness, example



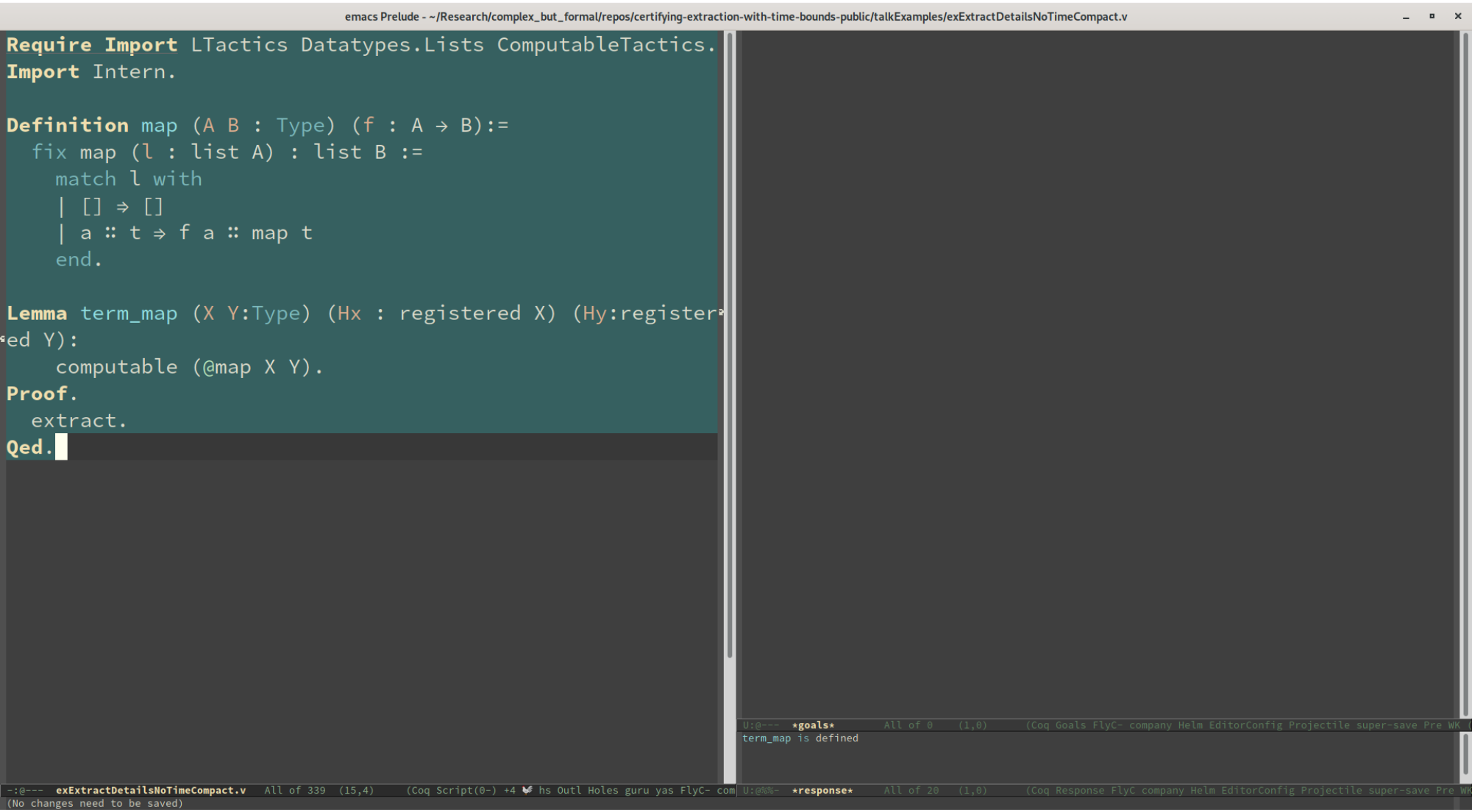
Proving correctness, example



Proving correctness, example

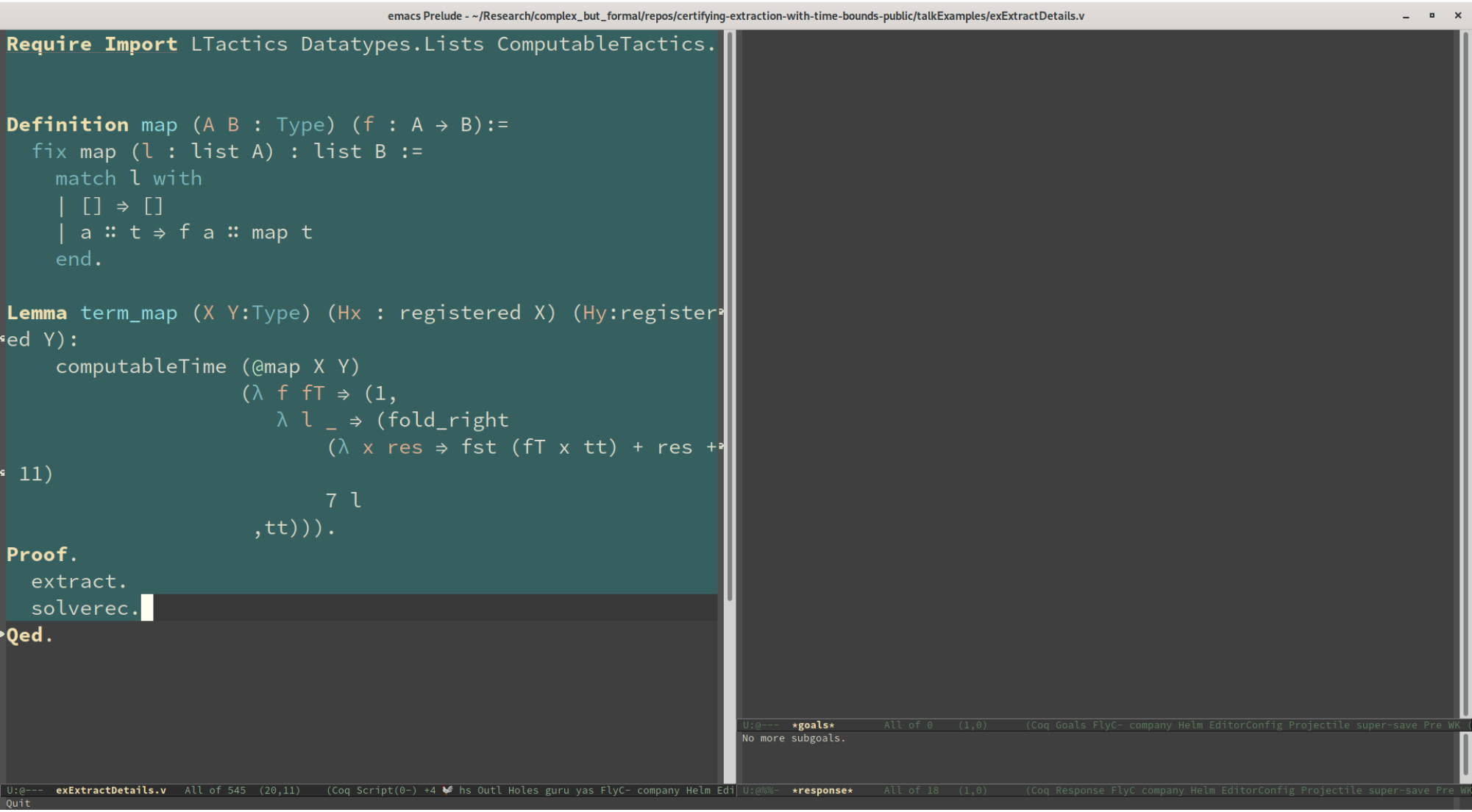


Proving correctness, example



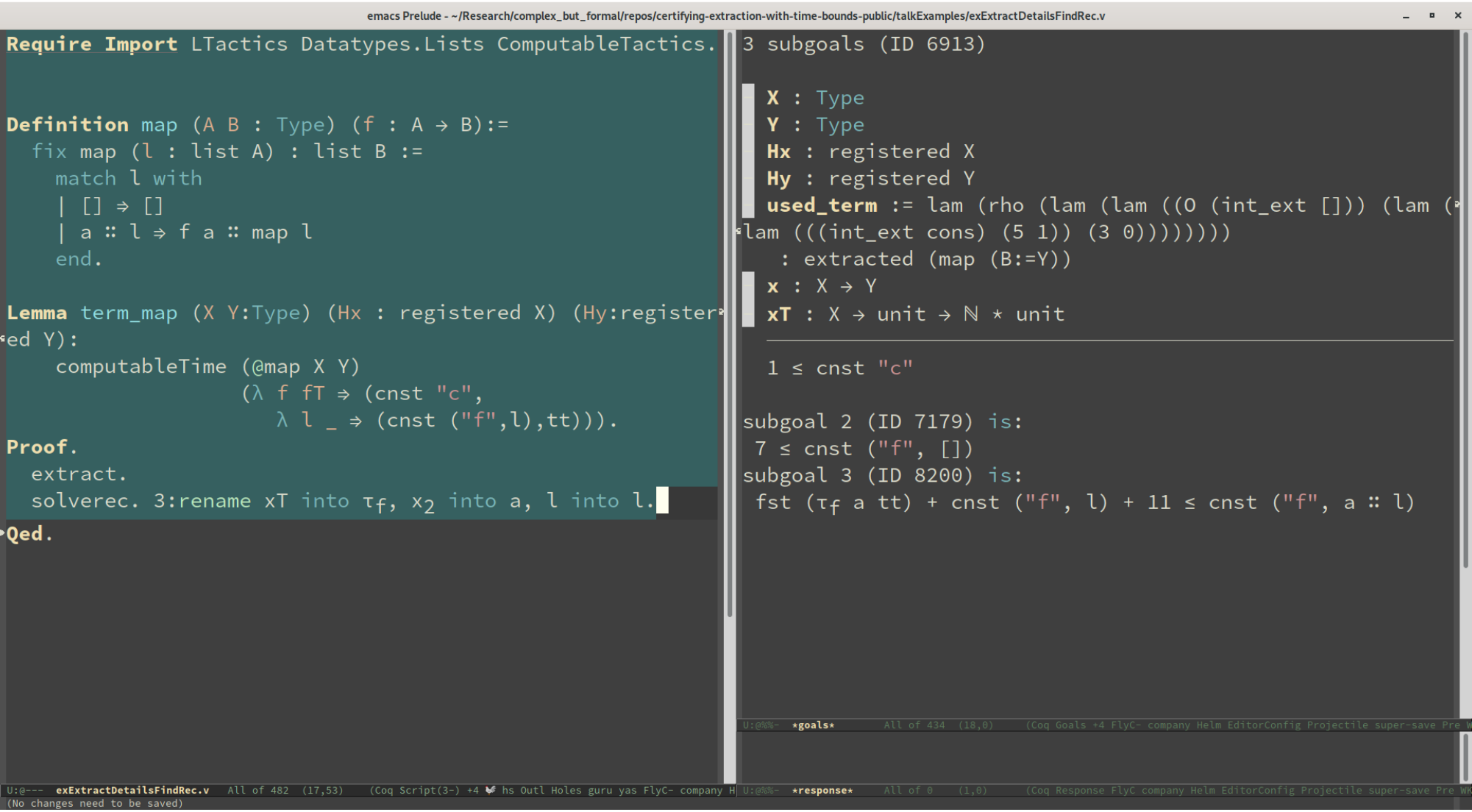
Deriving time complexity

Solving recurrence relations interactively



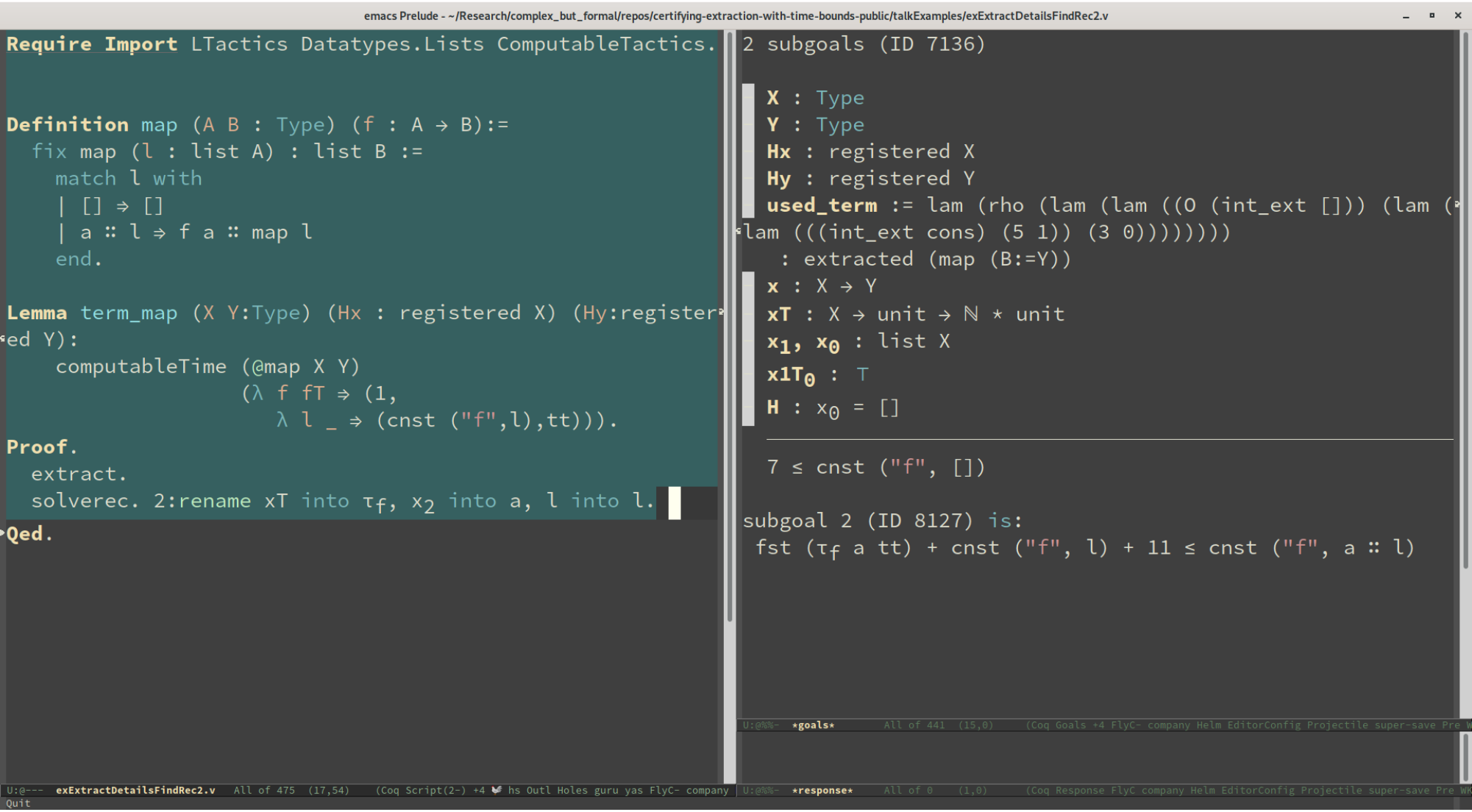
Deriving time complexity

Solving recurrence relations interactively



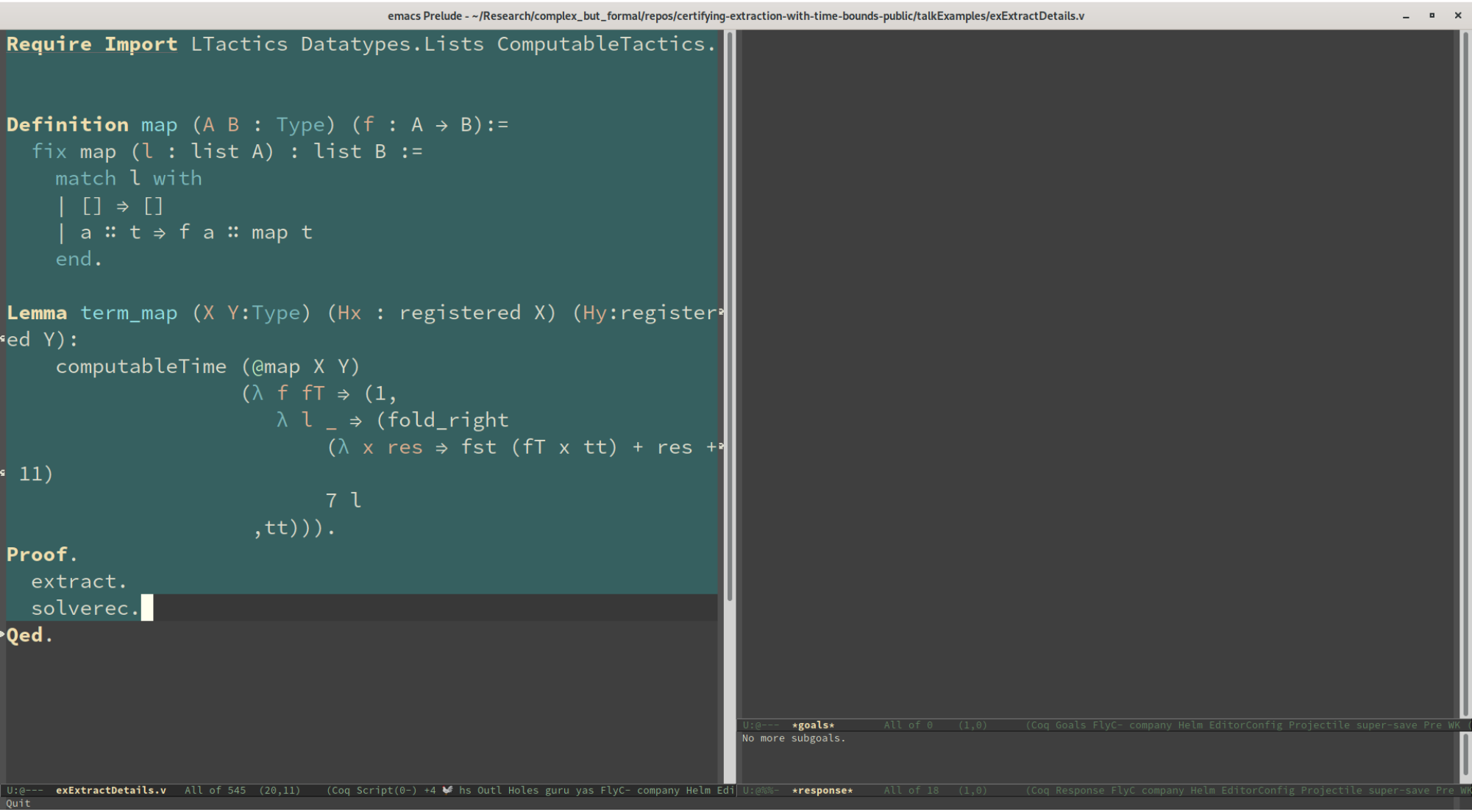
Deriving time complexity

Solving recurrence relations interactively



Deriving time complexity

Solving recurrence relations interactively



Case Study

Library of datatypes and functions

Build upon shared extraction of:

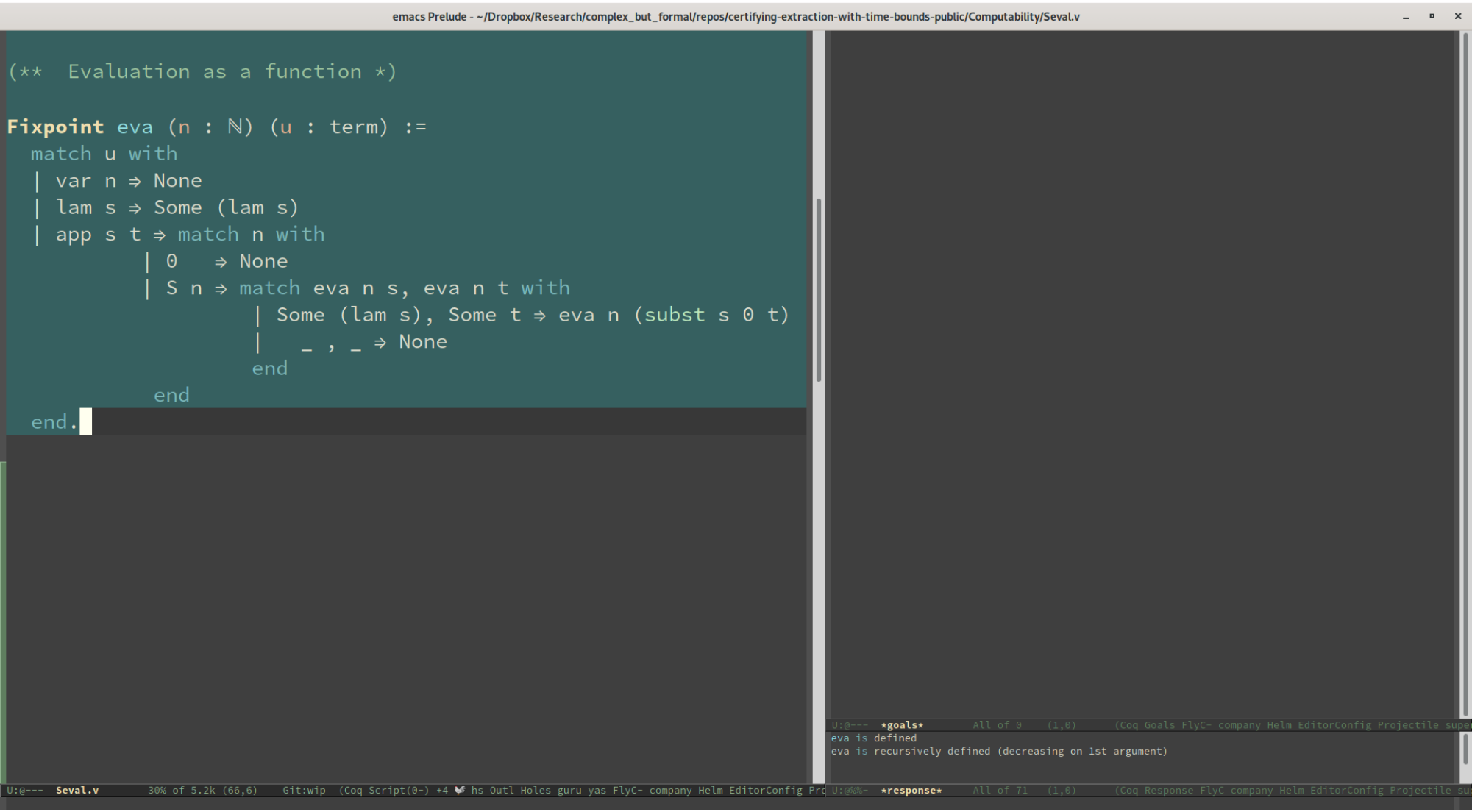
- $\mathbb{B}$, options, pairs ...
- $\mathbb{N}$: addition, multiplication, equality ...
- Lists: map, filter, ...
- $\approx$ 360 lines

Self interpreter for L

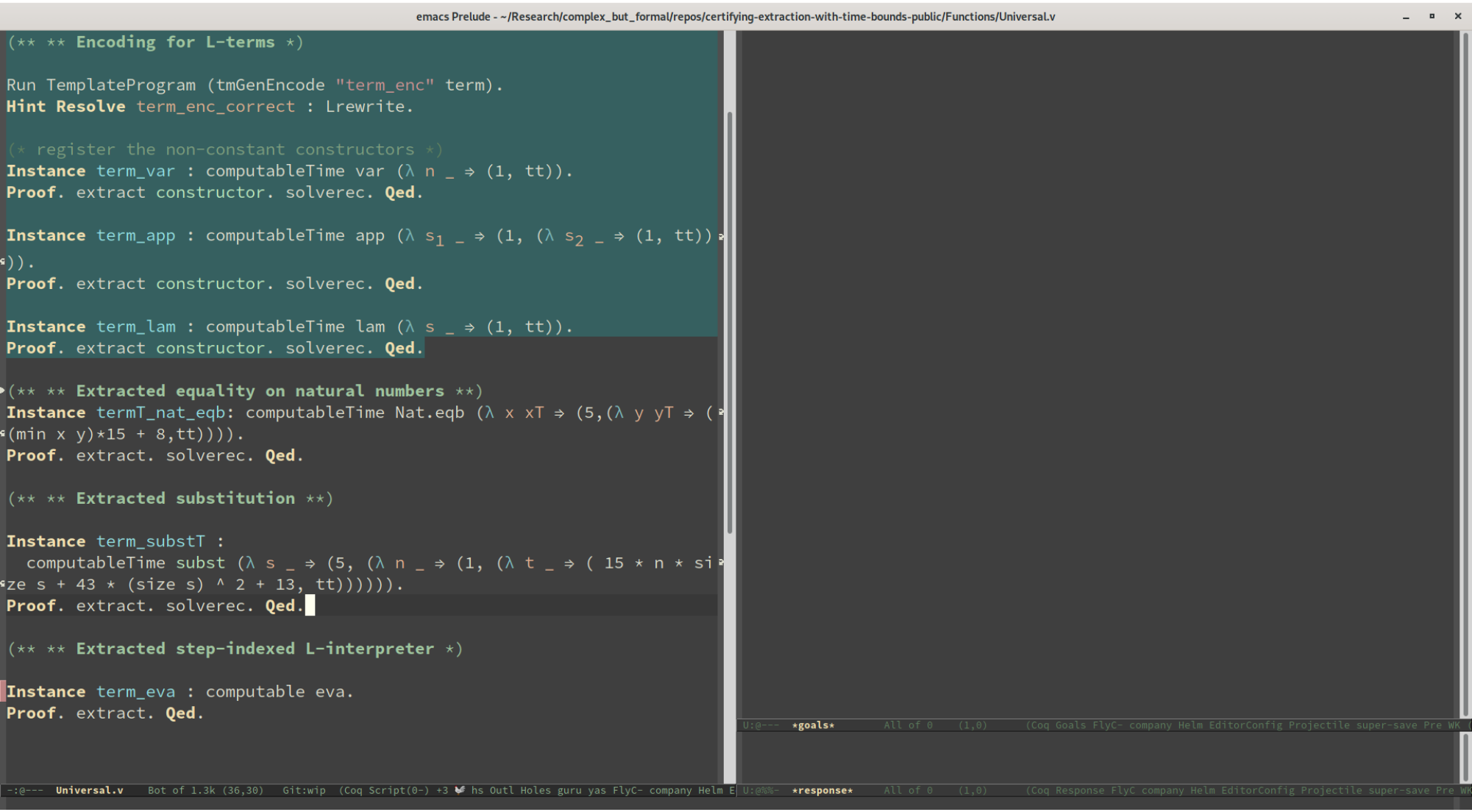
Universal L-term:

- Function $\texttt{eva} : \mathbb{N} \to \mathbf{T} \to \mathbf{T}_\bot$ with $s \succ^* t \Leftrightarrow \exists n, \texttt{eva}\ n\ s = t$
- Base for many results in computability theory
- 20 lines from specification in Coq to correct extraction in L

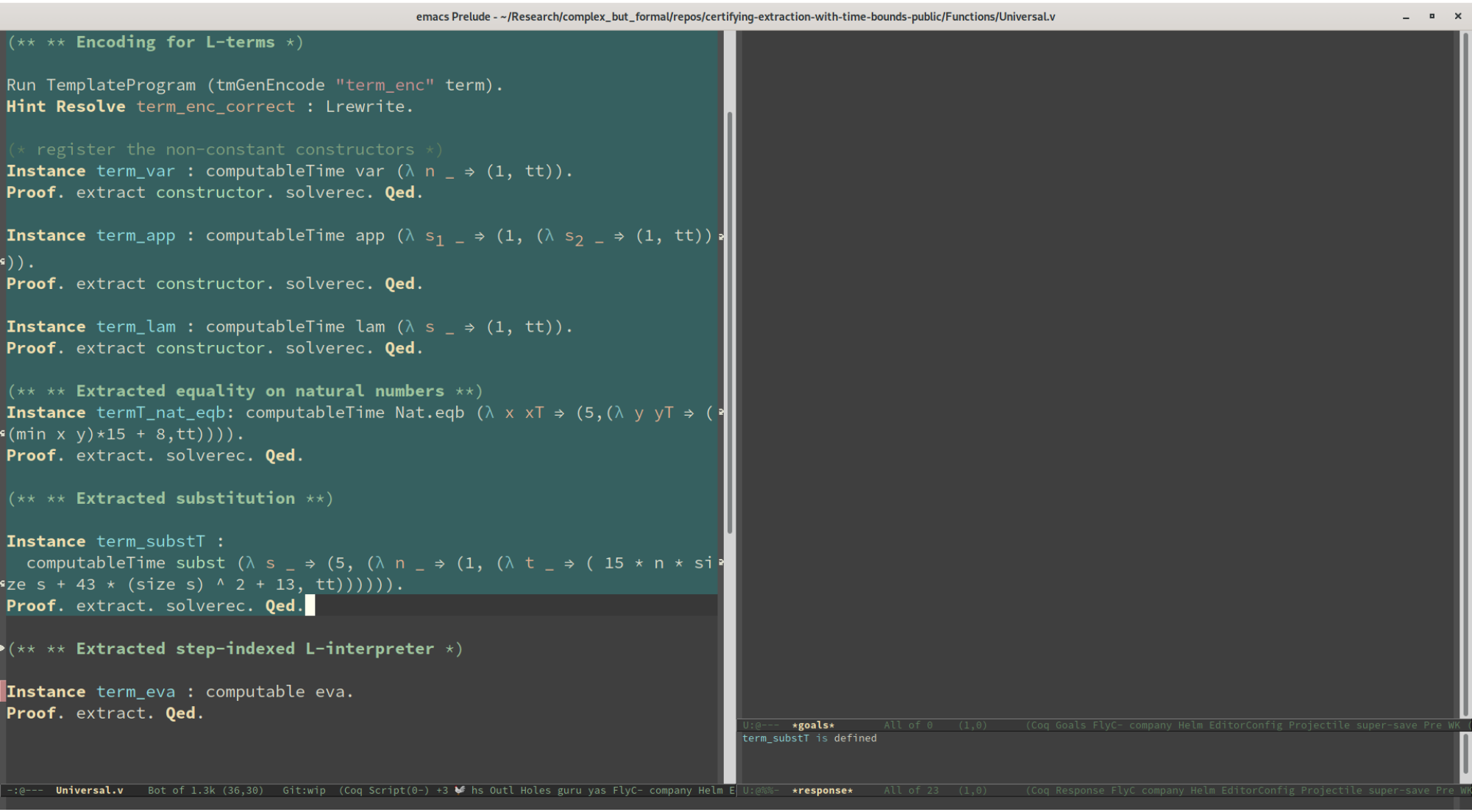
Self interpreter for L



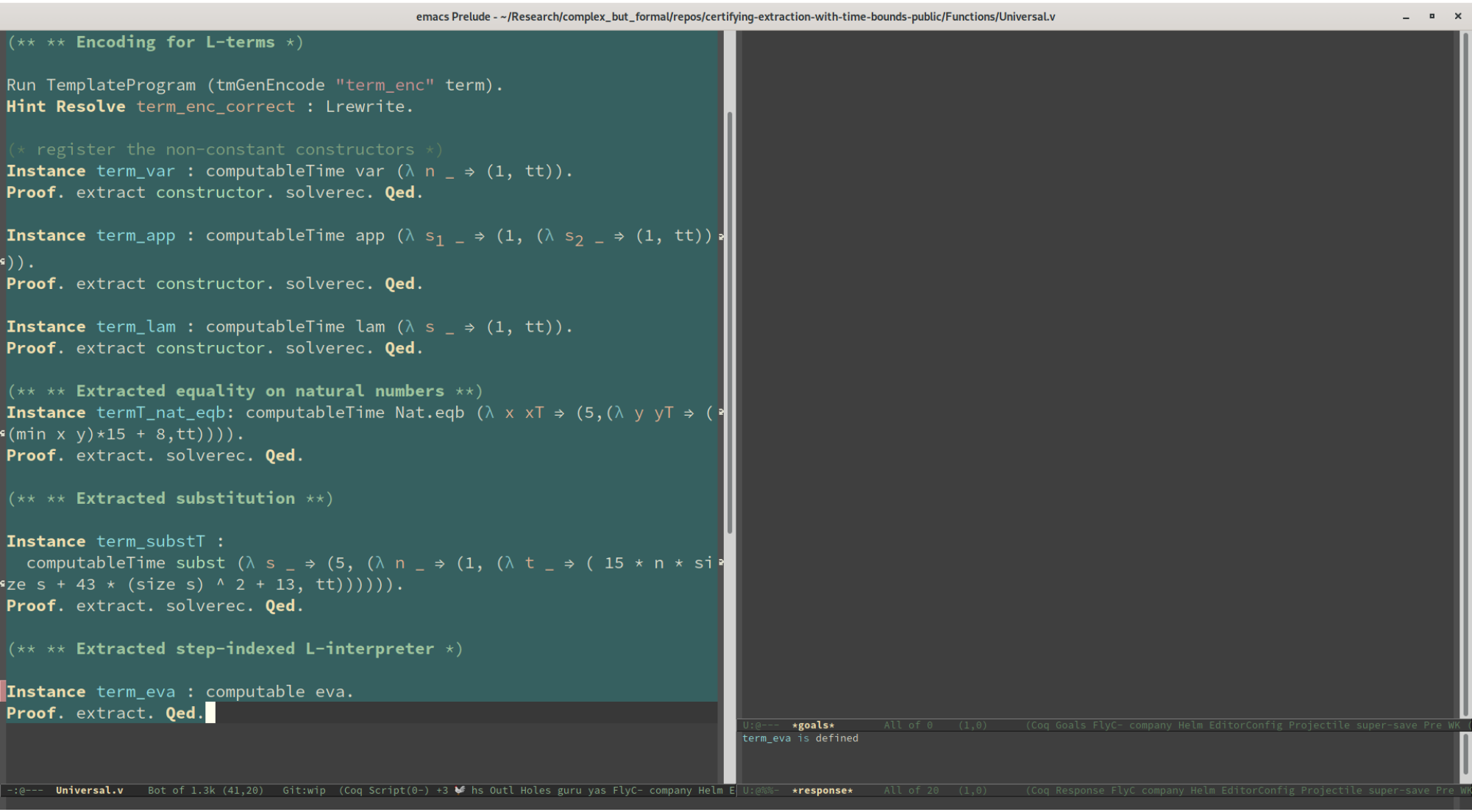
Self interpreter for L



Self interpreter for L



Self interpreter for L



Diophantine equations

- Diophantine sets are recursively enumerable
- $\Rightarrow$ Many-one reduction from diophantine equations to L-halting problem
- $\approx$ 200 lines

Turing machine interpreter

- Each step of a TM can be computed in constant time in L:

```
Global Instance term_loopM :
  let c1 := (haltTime + n*121 + transTime + 76) in
  let c2 := 13 + haltTime in
  computableTime (loopM (M:=M))
                 (fun _ _ ⇒ (5,fun k _ ⇒ (c1 * k + c2,tt))).
```

- Many-one reduction from TM-halting to L-halting problem
- $\approx$ 400 lines

Future Work

- Formalise complexity theory:
  - Extract *efficient* self interpreter
  - Time Hierarchy Theorem
  - NP-Completeness
- Include space analysis in framework
- Verify extraction process using MetaCoq
- Better treatment of dependent functions (realisability model for Coq)

Related Work

- Myreen and Owens: HOL4 to CakeML
- Hupel and Nipkow: Isabelle/HOL to CakeML
- Köpp: Minlog to λ-Calculus
- Œuf project: Verified compiler from Coq-Subset to Assembly
- CertiCoq project: Verified extraction from Coq to Clight.

Contribution

- A plugin extracting Coq functions of simple polymorphic types to cbv λ-calculus L
- Logical relations connecting Coq functions with correct extractions and time bounds
- Automated correctness and semi-automated time analysis for extracted terms
- Three case studies and library including $\mathbb{N}$ and lists:
  - Universal L-term
  - Reduction from Diophantine equations to L-halting problem
  - Polynomial-time simulation of Turing machines in L
- Contributed to library of undecidable problems in Coq: `github.com/uds-psl/coq-library-undecidability`

Thank you!

References

Beniamino Accattoli and Ugo Dal Lago. (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. *Logical Methods in Computer Science*, 12 (1), 2016. doi: 10.2168/LMCS-12(1:4)2016.

Yannick Forster and Gert Smolka. Weak call-by-value lambda calculus as a model of computation in Coq. In *ITP 2017*, pages 189–206. Springer, 2017.

Gordon D. Plotkin. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi: 10.1016/0304-3975(75)90017-1.

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. working paper or preprint, June 2019. URL `https://hal.inria.fr/hal-02167423`.

LoC

Framework:

	loc
Correctness predicate	370
Extraction	380
Simplification Tactics	950
Verifying Tactics	420
total	2100

Case Studies:

	spec	proof
Library	355	282
Universal Term	13	7
H10	79	124
Universal TM	243	151

Correctness with time bounds

$$\frac{}{\mathtt{enc}_A\, a \sim^{\tau} a} \quad (\text{for } a : A)$$

$$\frac{t_f \text{ is a procedure} \;\wedge\; \forall a\, t_a\, \tau_a.\; t_a \sim^{\tau_a} a \to \Sigma v : \mathbf{T}.\; t_f\, t_a \succ^{\leq n} v \wedge v \sim^{\tau} f a \text{ where } \tau_f\, a\, \tau_a = (n, \tau)}{t_f \sim^{\tau_a} f} \quad (\text{for } f : A \to B)$$