Proofs as programs: challenges and strategies for program synthesis

Shaowei Lin
20210422
Topos Institute Colloquium

Joint work with
Zhangsheng Lai, Liang Ze Wong,
Jin Xing Lim, Barnabe Monnot, Georgios Piliouras.

Background

Goal

- Open protocols for decentralized embodied intelligence

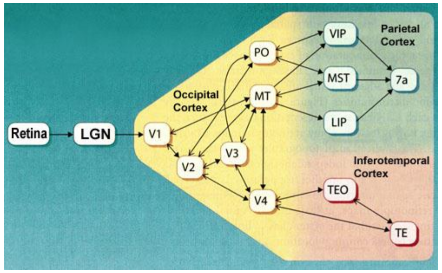
Interests

- Statistical learning theory for spiking networks
- Dependent type theory for machine reasoning
- Quantum path integrals and motivic information

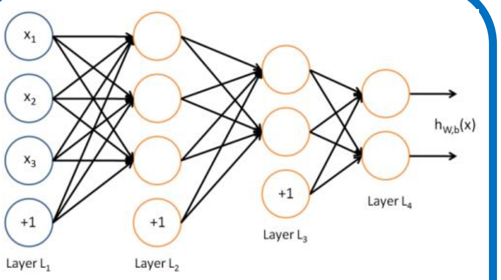
History

- Berkeley PhD Mathematics
- DARPA Math Challenges in Deep Learning
- A*STAR Urban Systems Initiative
- SUTD Assistant Professor

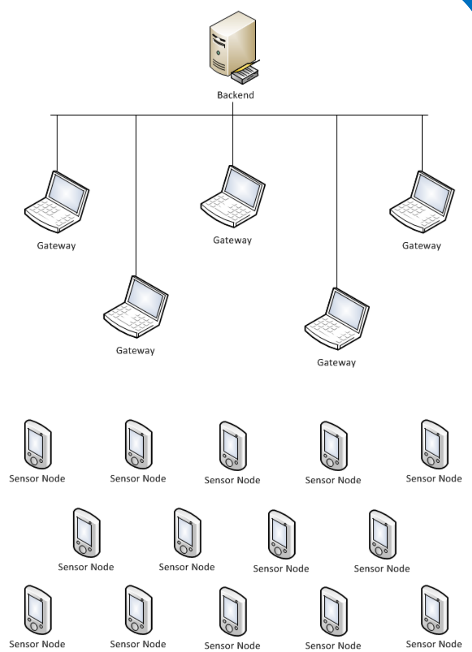
Nervous system for smart cities



Deep visual cortex



Deep learning



Sensor networks

Smart city applications

Domain	Application
unified platform	sharing of sensors and data among government agencies
urban planning	combining microclimatic models with sensing for town designs
environmental monitoring	measuring noise pollution for enforcement of noise laws
structural health monitoring	detecting faults on port cranes through sensors and analytics
infrastructural maintenance	detecting potholes, broken lights through cars with sensors
public cleanliness	sensing trash bin fill-levels to reduce cleaning workloads
high-tech farming	improving crop yields with sensors in green houses
elderly healthcare	monitoring elderly for falls, depression through home sensors
power grid security	detecting, mitigating attacks with adversarial machine learning

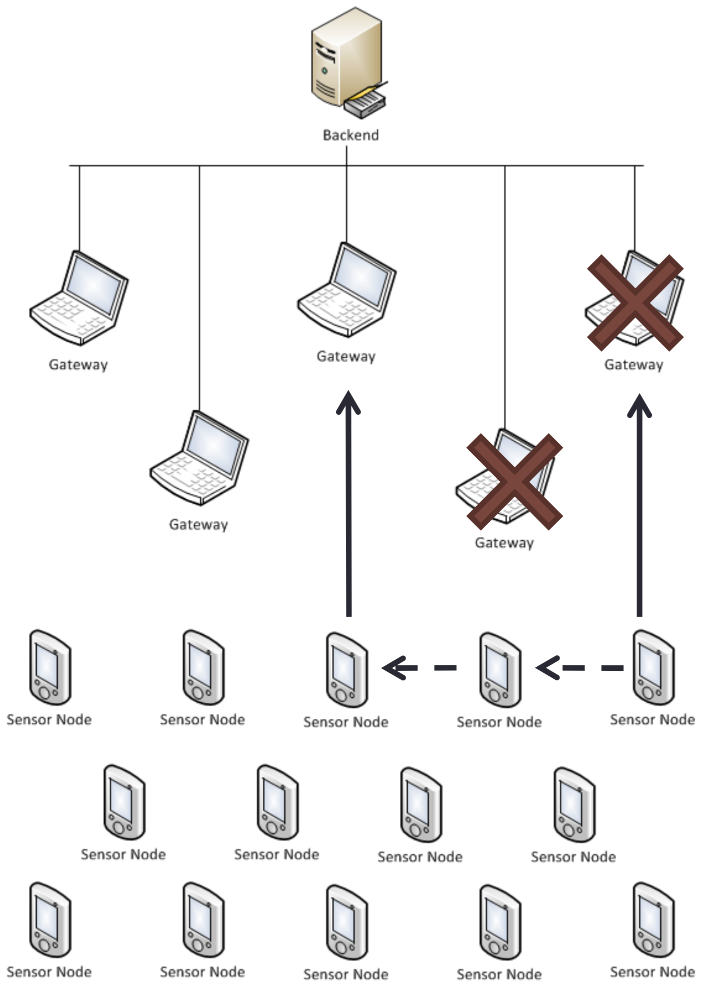
Reprogramming on the fly

Example 1

A city has camera nodes which send videos to the backend via wifi gateways. In emergencies, the nodes can be reached by 4G.

During a natural disaster, some gateways were destroyed.

How do we reprogram a camera node to stream critical videos by relaying through wifi connections to other nodes?



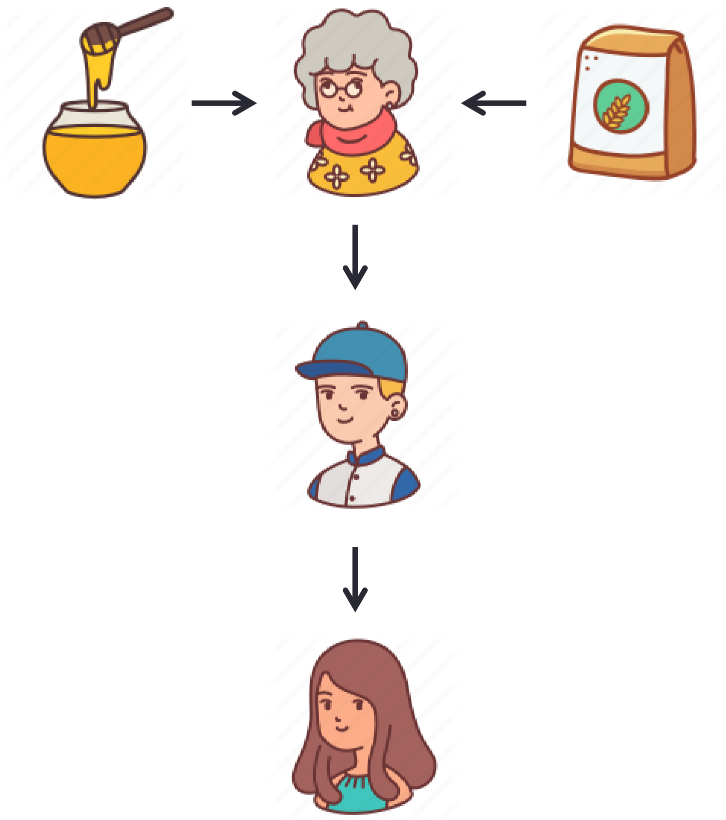
Wrapping services in code

Example 2

My grandmother wants to start a bakery. She is great at cooking but is poor with technology.

To ensure freshness, she wants to do just-in-time baking. This means that she will only order ingredients and arrange deliveries when customers put in their orders.

How can she source for ingredients and deliveries in real time while also selling to the community?



Top-down software synthesis

Example 3

The CEO of Acme Books wants a robot that takes a box of books and arranges them in alphabetical order of authors on a shelf.

His manager buys a generic android from Atoz Bots. She observes that trivially an empty shelf is sorted.

She instructs her engineer to design an algorithm to insert a new book on a sorted shelf. Her proof assistant verifies that this gets the job done.

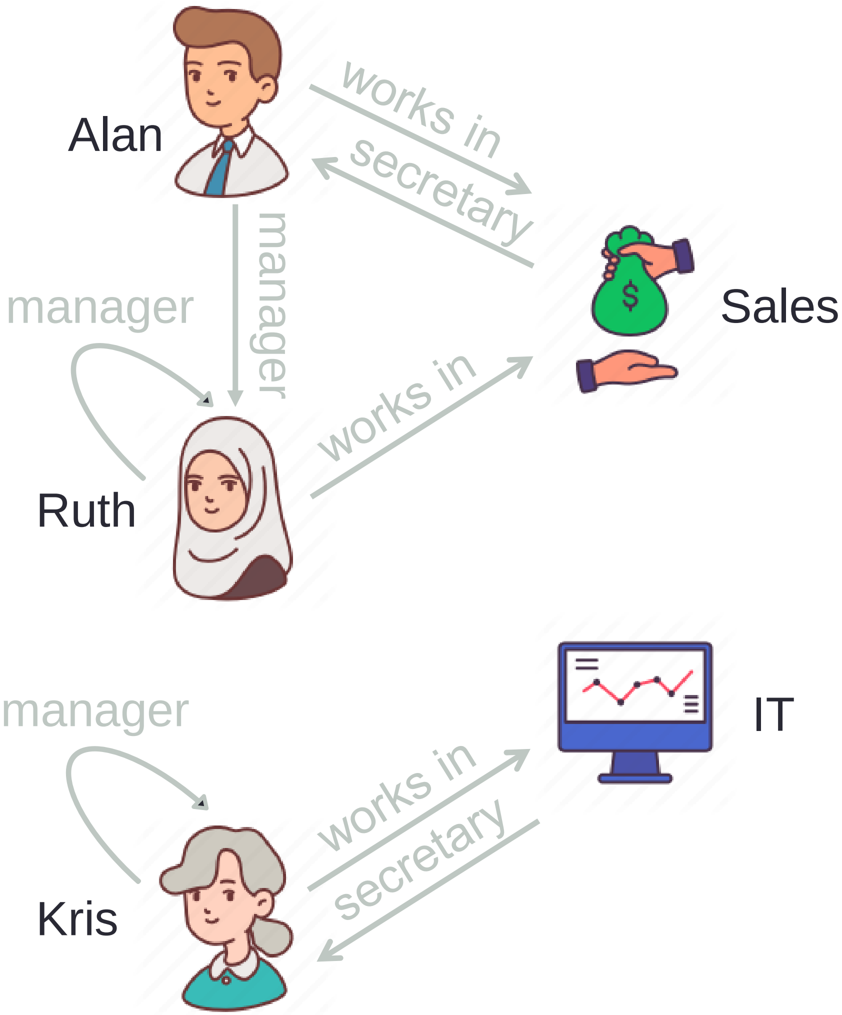
Knowledge graph queries

Example 4 [Fong & Spivak 2018]

The following knowledge graph shows the entities in a company and the relations between them.

The CEO needs to contact the secretary of Ruth's department.

An auditor wants a list of all the managers in the company. He wonders if every manager works in the department they manage.



Fong, Brendan, and David I. Spivak. "Seven sketches in compositionality: An invitation to applied category theory." *arXiv preprint arXiv:1803.05316* (2018).

Decentralized Embodied Intelligence

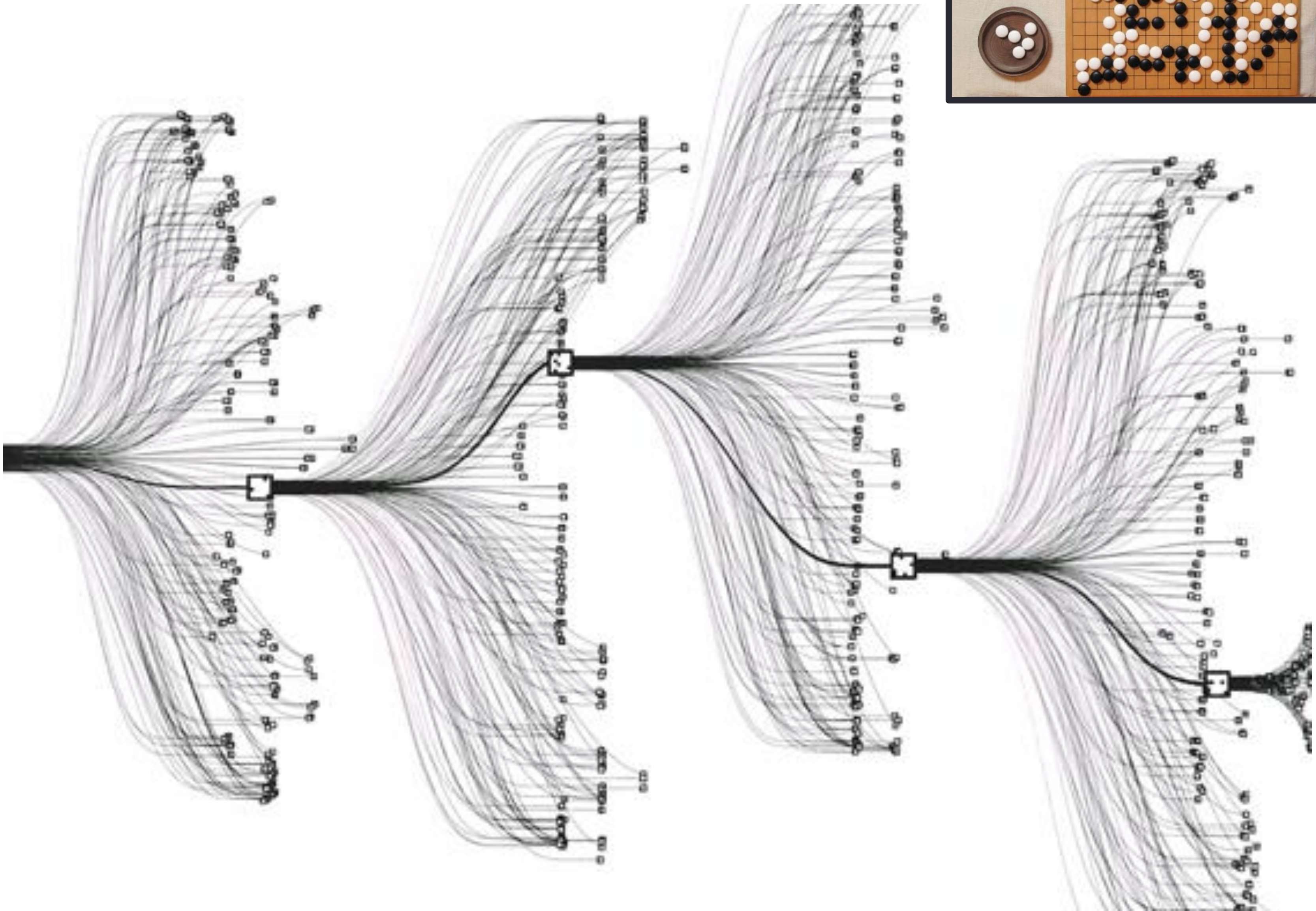
How can a network of agents accomplish given tasks by performing decentralized steps, managing embodied resources, and learning intelligent strategies?

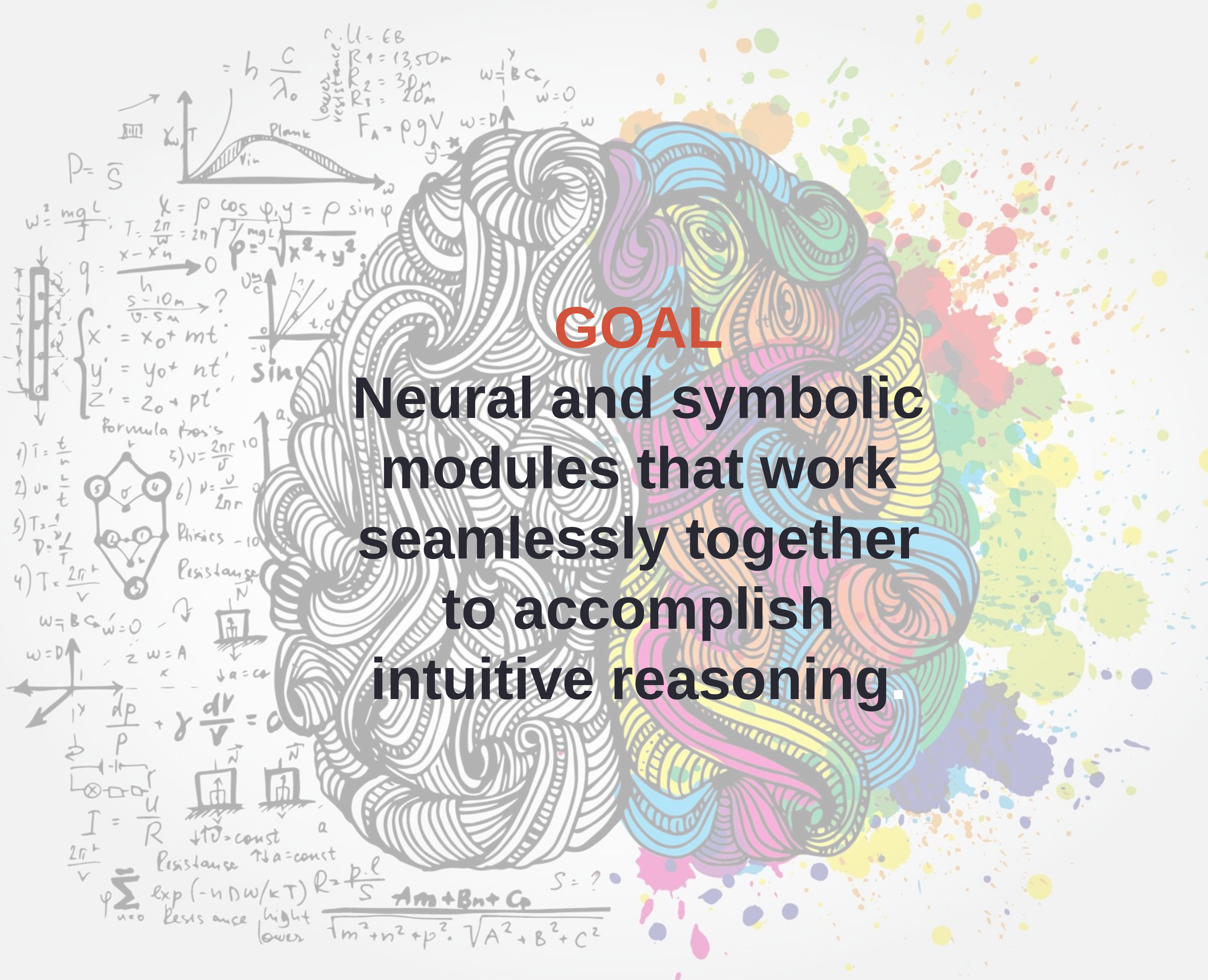
action space?

objective function?

machine reasoning?

Reinforcement learning





GOAL

Neural and symbolic modules that work seamlessly together to accomplish intuitive reasoning.

Proof assistants

Proof assistants

Axioms, definitions, theorems make up the **global context**, e.g.

Definition. A list is sorted if it is

- empty;
- a one-element list; or
- of the form $(x :: y :: l)$ where $x \leq y$ and $(y :: l)$ is sorted.

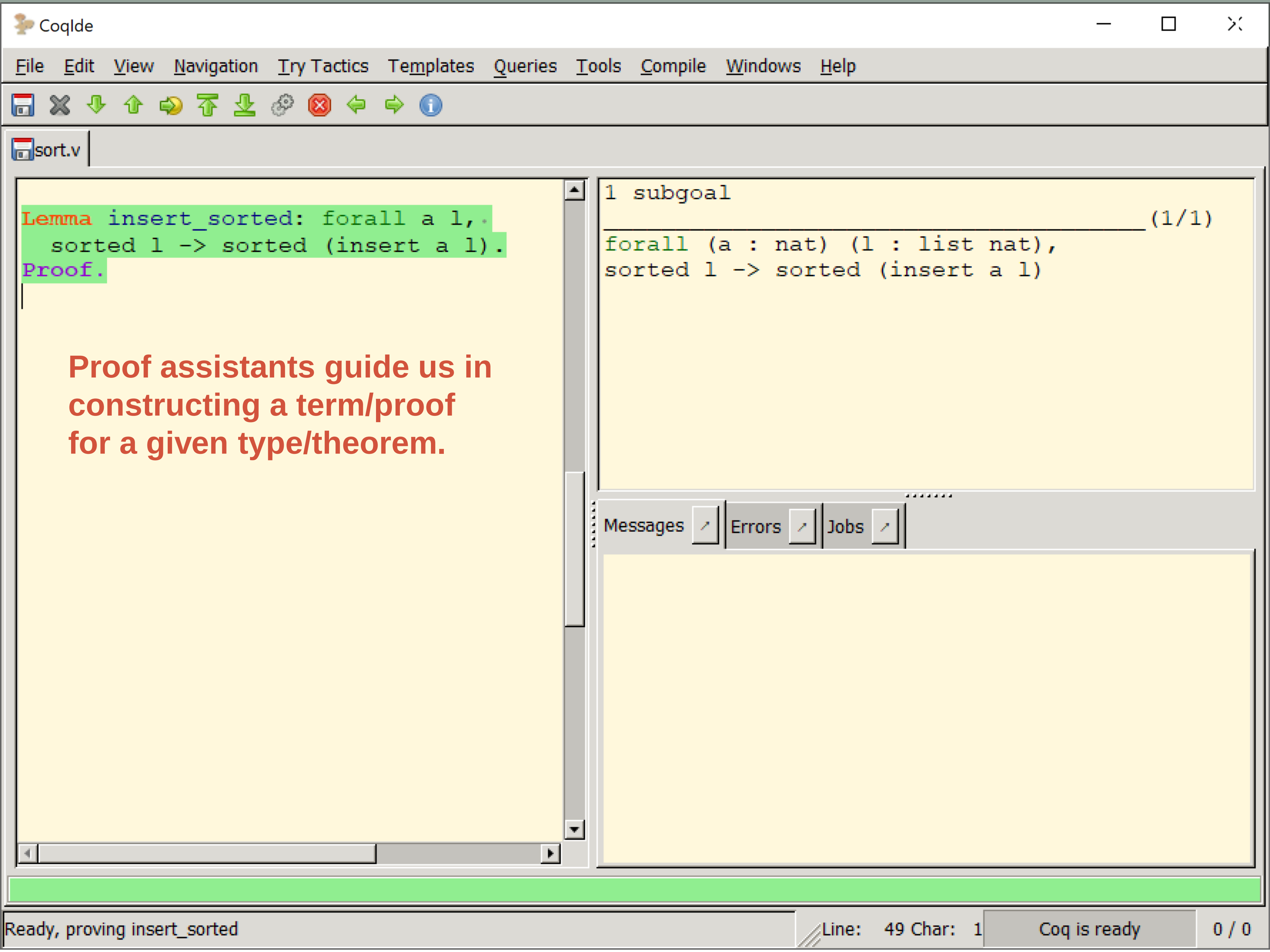
```
Inductive sorted : list nat → Prop :=
| sorted_nil  : sorted []
| sorted_one  : ∀ x, sorted [x]
| sorted_cons : ∀ x y l,
                x ≤ y → sorted (y :: l)
                → sorted (x :: y :: l).
```

Definition. The list $\mathrm{insert}(i, l)$ is

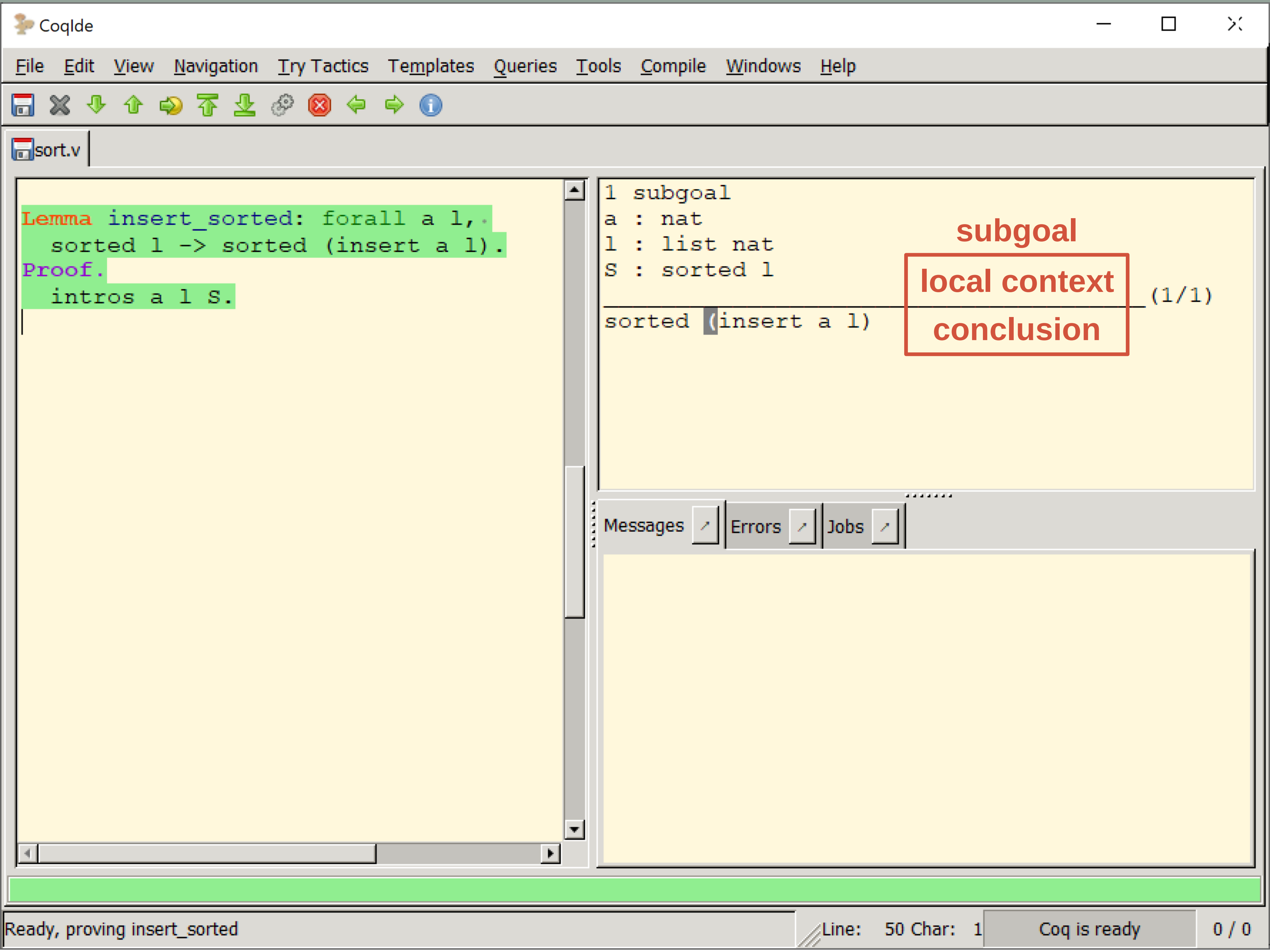
- $[i]$ if l is empty;
- $(i :: h :: t)$ if $l = (h :: t)$ and $i \leq h$;
- $\big(h :: \mathrm{insert}(i, t)\big)$ if $l = (h :: t)$ and $i > h$.

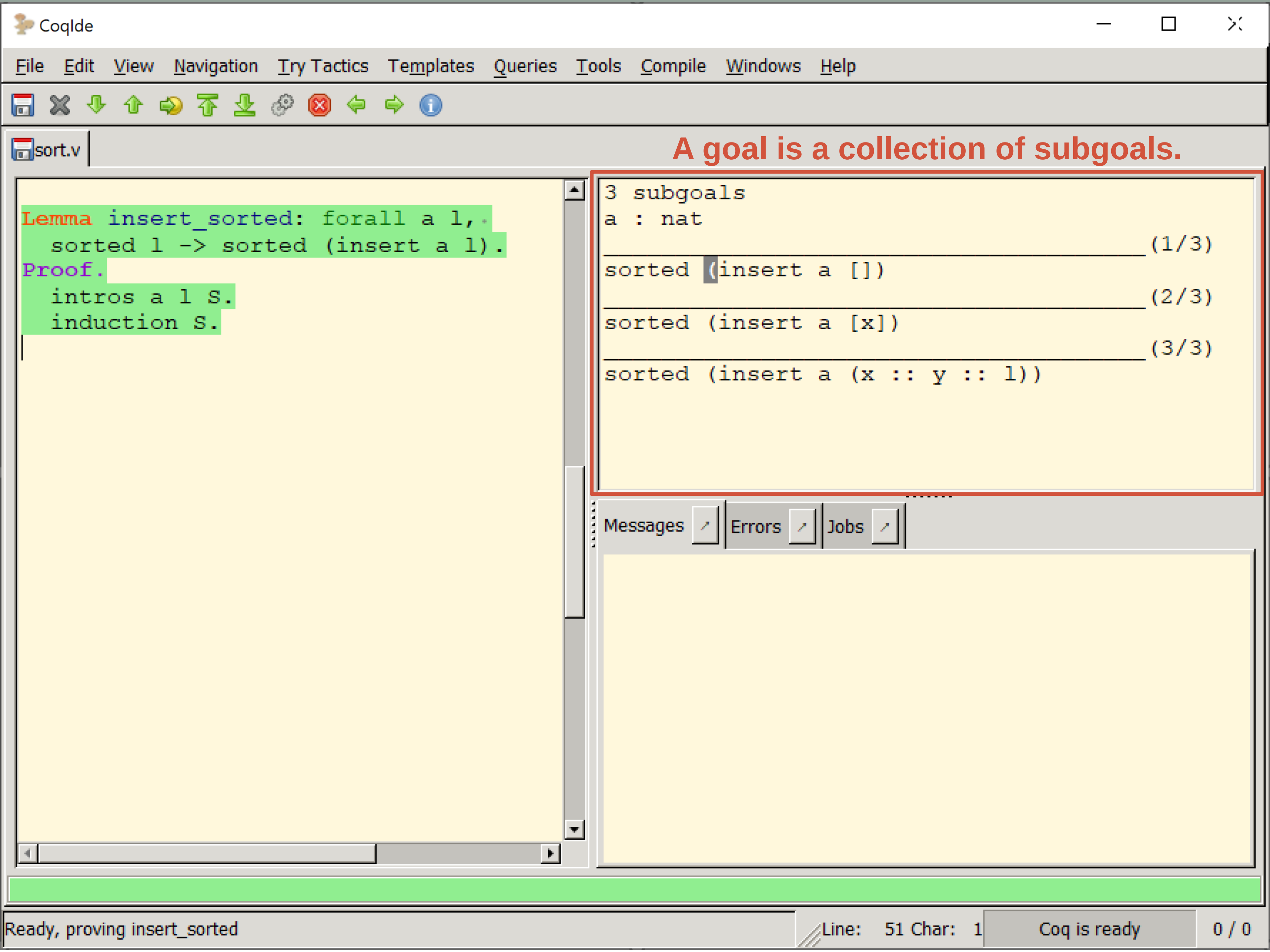
```
Fixpoint insert (i : nat) (l : list nat) :=
  match l with
  | [] ⇒ [i]
  | h :: t ⇒ if i <=? h then i :: h :: t
                        else h :: insert i t
  end.
```

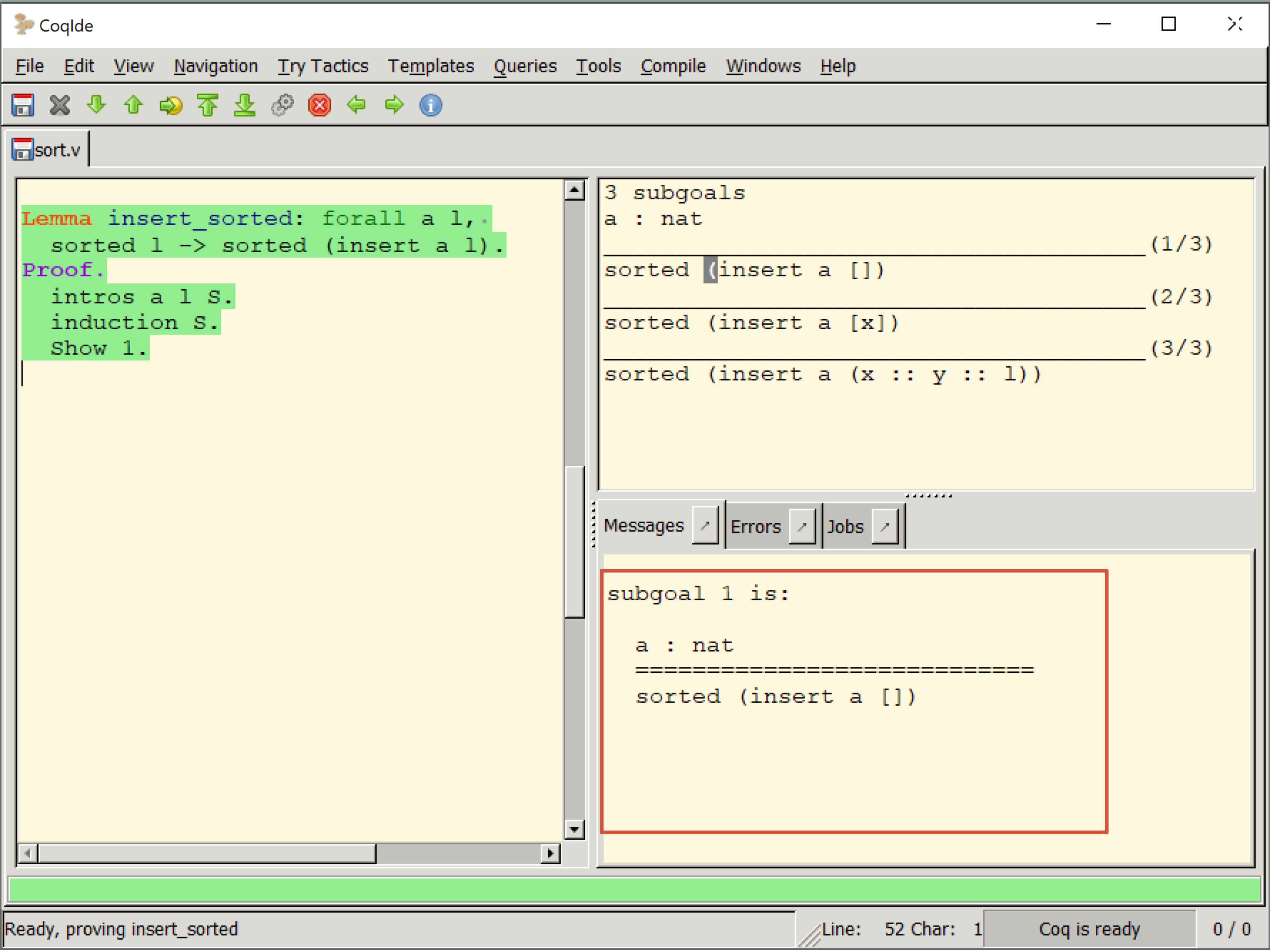
Pierce, Benjamin C., Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. "Software foundations." *Webpage: http://www.cis.upenn.edu/bcpierce/sf/current/index.html* (2010).

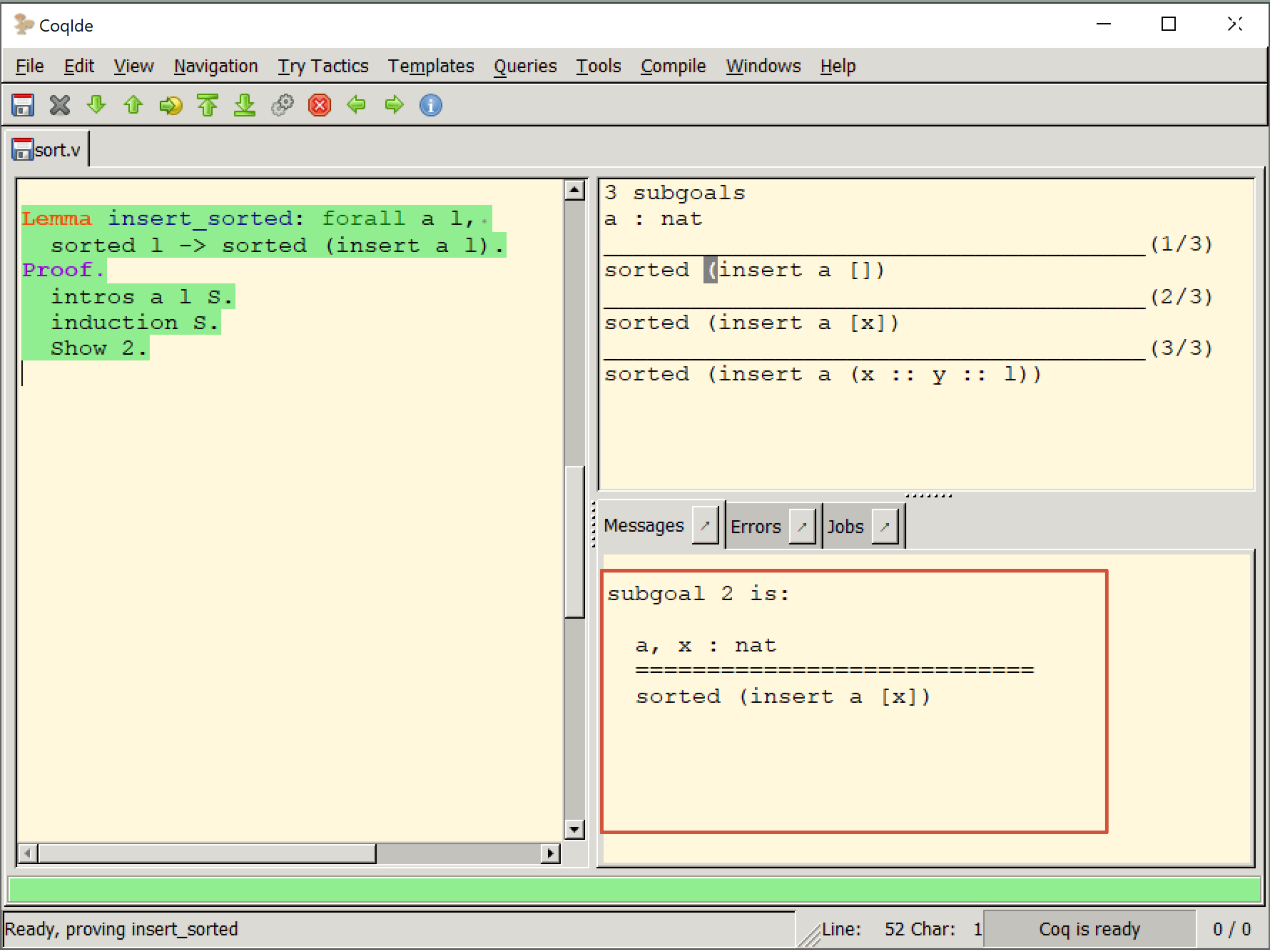


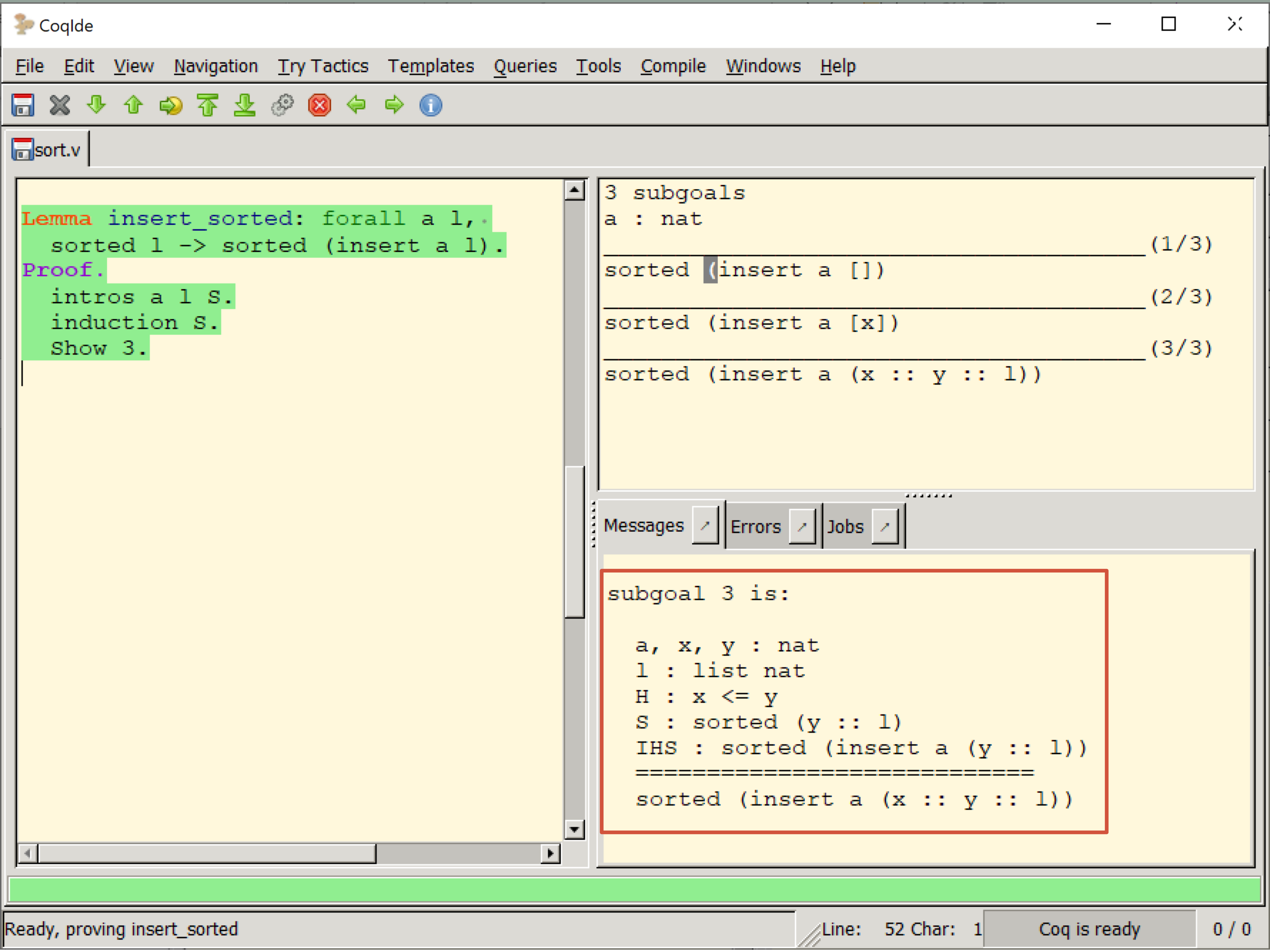
Proof assistants guide us in constructing a term/proof for a given type/theorem.

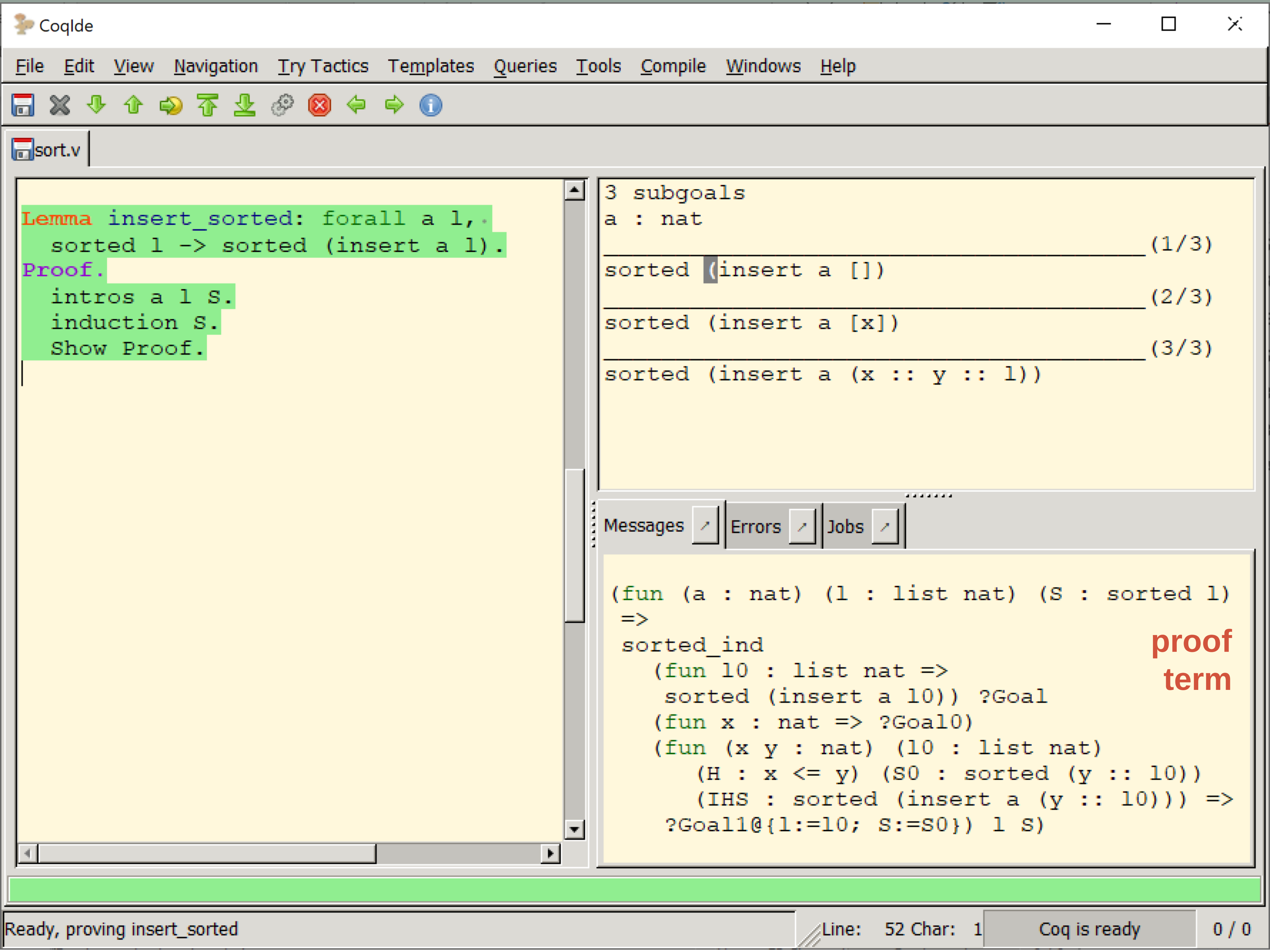

CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
sort.v
Lemma insert_sorted: forall a l,
sorted l -> sorted (insert a l).
Proof.
intros a l S.
1 subgoal
a : nat
l : list nat
S : sorted l
(1/1)
sorted (insert a l)
subgoal
local context
conclusion
Messages
Errors
Jobs
Ready, proving insert_sorted
Line: 50 Char: 1
Coq is ready
0 / 0


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
sort.v
A goal is a collection of subgoals.
Lemma insert_sorted: forall a l,
sorted l -> sorted (insert a l).
Proof.
intros a l S.
induction S.
3 subgoals
a : nat
(1/3)
sorted (insert a [])
(2/3)
sorted (insert a [x])
(3/3)
sorted (insert a (x :: y :: l))
Messages
Errors
Jobs
Ready, proving insert_sorted
Line: 51 Char: 1
Coq is ready
0 / 0


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
sort.v
Lemma insert_sorted: forall a l,
sorted l -> sorted (insert a l).
Proof.
intros a l S.
induction S.
Show 1.
3 subgoals
a : nat
(1/3)
sorted (insert a [])
(2/3)
sorted (insert a [x])
(3/3)
sorted (insert a (x :: y :: l))
Messages
Errors
Jobs
subgoal 1 is:
a : nat
============================
sorted (insert a [])
Ready, proving insert_sorted
Line: 52 Char: 1
Coq is ready
0 / 0

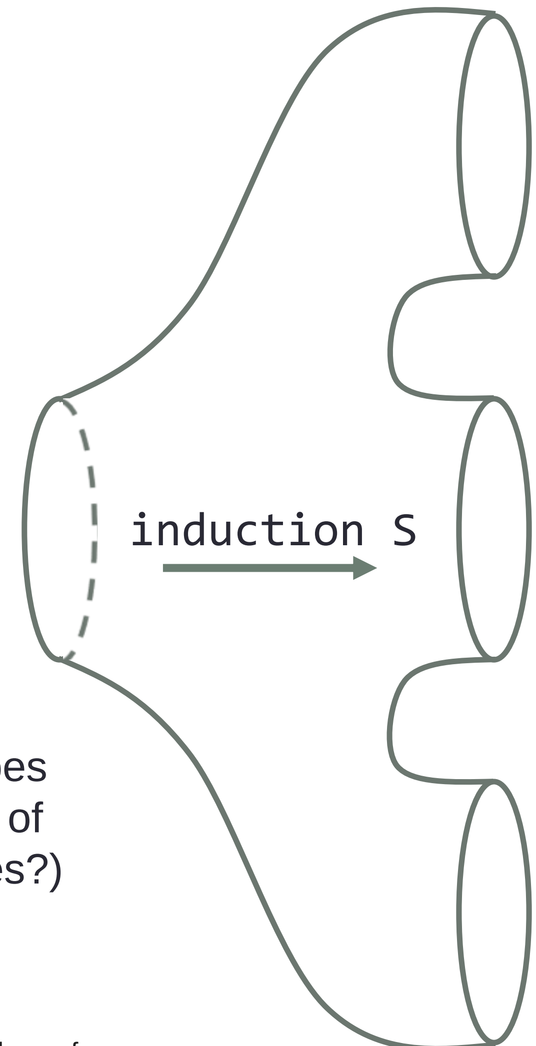

CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
sort.v
Lemma insert_sorted: forall a l,
sorted l -> sorted (insert a l).
Proof.
intros a l S.
induction S.
Show 2.
3 subgoals
a : nat
(1/3)
sorted (insert a [])
(2/3)
sorted (insert a [x])
(3/3)
sorted (insert a (x :: y :: l))
Messages
Errors
Jobs
subgoal 2 is:
a, x : nat
============================
sorted (insert a [x])
Ready, proving insert_sorted
Line: 52 Char: 1
Coq is ready
0 / 0


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
sort.v
Lemma insert_sorted: forall a l,
sorted l -> sorted (insert a l).
Proof.
intros a l S.
induction S.
Show 3.
3 subgoals
a : nat
(1/3)
sorted (insert a [])
(2/3)
sorted (insert a [x])
(3/3)
sorted (insert a (x :: y :: l))
Messages
Errors
Jobs
subgoal 3 is:
a, x, y : nat
l : list nat
H : x <= y
S : sorted (y :: l)
IHS : sorted (insert a (y :: l))
============================
sorted (insert a (x :: y :: l))
Ready, proving insert_sorted
Line: 52 Char: 1
Coq is ready
0 / 0


CoqIde
File Edit View Navigation Try Tactics Templates Queries Tools Compile Windows Help
sort.v
Lemma insert_sorted: forall a l,
sorted l -> sorted (insert a l).
Proof.
intros a l S.
induction S.
Show Proof.
3 subgoals
a : nat
(1/3)
sorted (insert a [])
(2/3)
sorted (insert a [x])
(3/3)
sorted (insert a (x :: y :: l))
Messages
Errors
Jobs
(fun (a : nat) (l : list nat) (S : sorted l)
=>
sorted_ind
(fun l0 : list nat =>
sorted (insert a l0)) ?Goal
(fun x : nat => ?Goal0)
(fun (x y : nat) (l0 : list nat)
(H : x <= y) (S0 : sorted (y :: l0))
(IHS : sorted (insert a (y :: l0))) =>
?Goal1@{l:=l0; S:=S0}) l S)
proof
term
Ready, proving insert_sorted
Line: 52 Char: 1
Coq is ready
0 / 0

Category of goals and tactics

```
=================================
forall (a : nat) (l : list nat),
sorted l -> sorted (insert a l)
```




```
a : nat
=====================
sorted (insert a [])
```

```
a, x : nat
=====================
sorted (insert a [x])
```

```
a, x, y : nat
l : list nat
H : x <= y
S : sorted (y :: l)
IHS : sorted (insert a (y :: l))
================================
sorted (insert a (x :: y :: l))
```

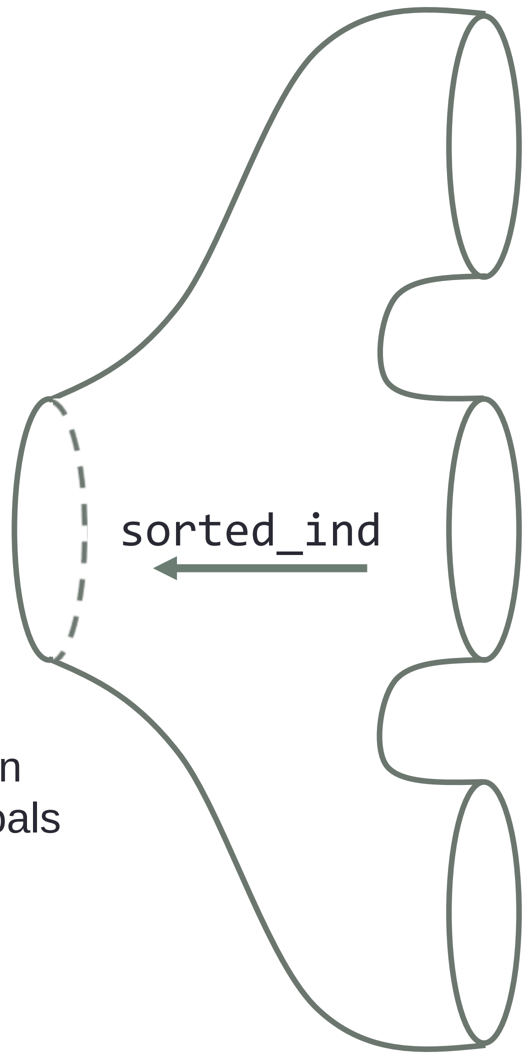
Ignore equalities between types from unfolding (δ-reductions) of definitions (weak ∞-categories?)

Finster, Eric, David Reutter, and Jamie Vicary. "A Type Theory for Strictly Unital ∞ -Categories." *arXiv preprint arXiv:2007.08307* (2020).

Category of types and terms

```
forall (a : nat),
sorted (insert a [])
```

```
forall (a : nat) (l : list nat),
sorted l -> sorted (insert a l)
```

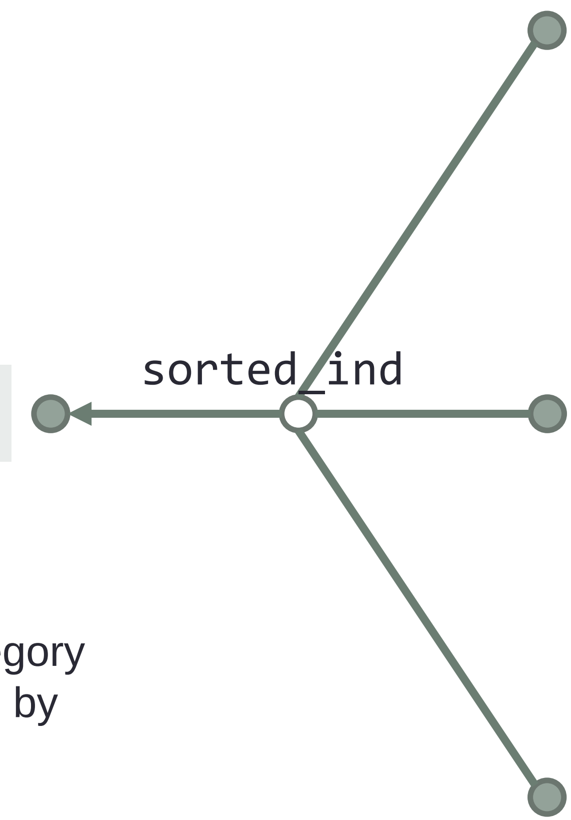



```
forall (a, x : nat),
sorted (insert a [x])
```

```
forall (a, x, y : nat)
(l : list nat)
(H : x <= y)
(S : sorted (y :: l))
(IHS : sorted (insert a (y :: l))),
sorted (insert a (x :: y :: l))
```

Some goal structures forgotten
by functor from category of goals
to category of types.

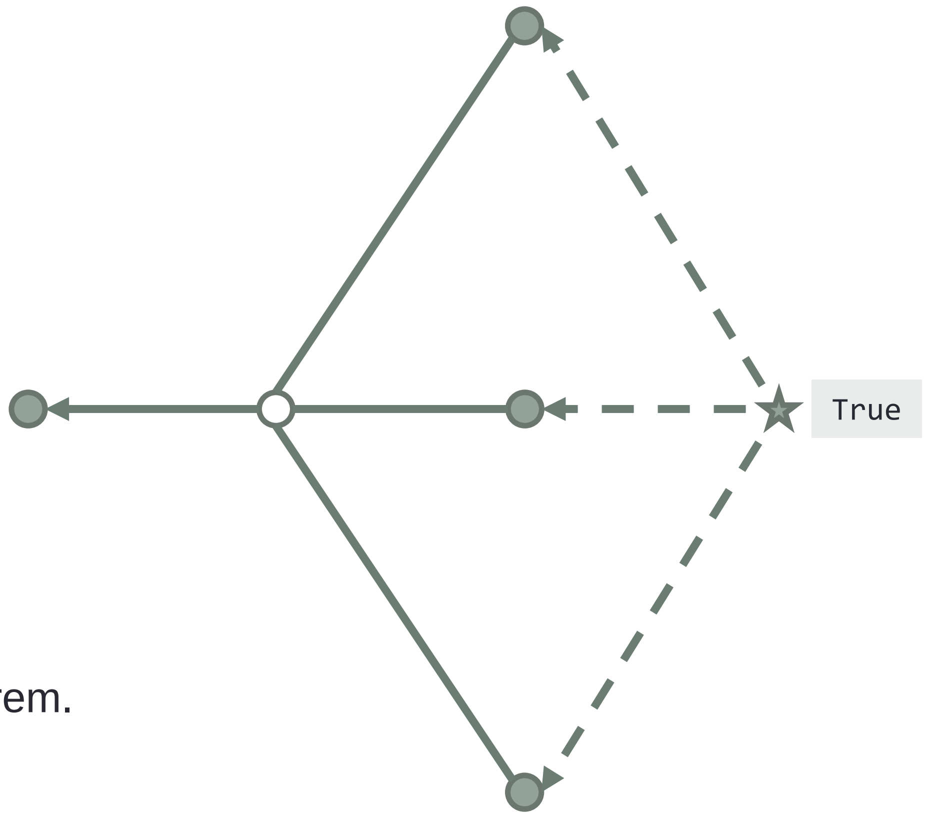
Category of types and terms



Morphisms in monoidal category of types can be represented by string diagrams.

Category of types and terms

```
forall (a : nat) (l : list nat),
sorted l -> sorted (insert a l)
```



A proof is a path from the terminal/unit type to the theorem.

Program synthesis

Intents as types, implementations as terms.

Example. Type of all sorting algorithms.

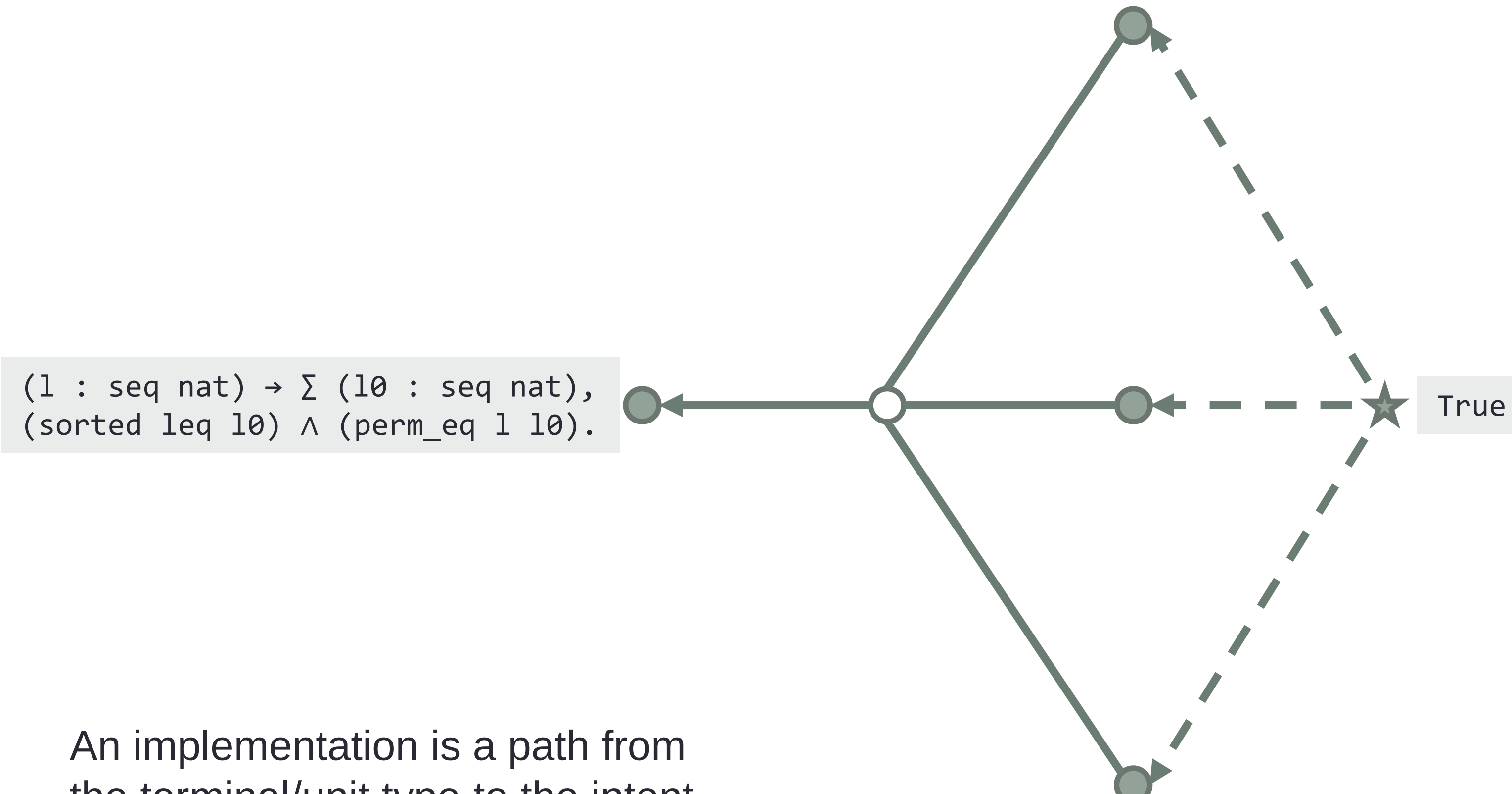
```
Theorem sort_spec :
(l : seq nat) → ∑ (l0 : seq nat),
(sorted leq l0) ∧ (perm_eq l l0).
```

Top-down (not bottom-up) synthesis of algorithm from specification.

Intents should specify constraints on the algorithm for greater reuse. They should not specify the steps of the algorithm.

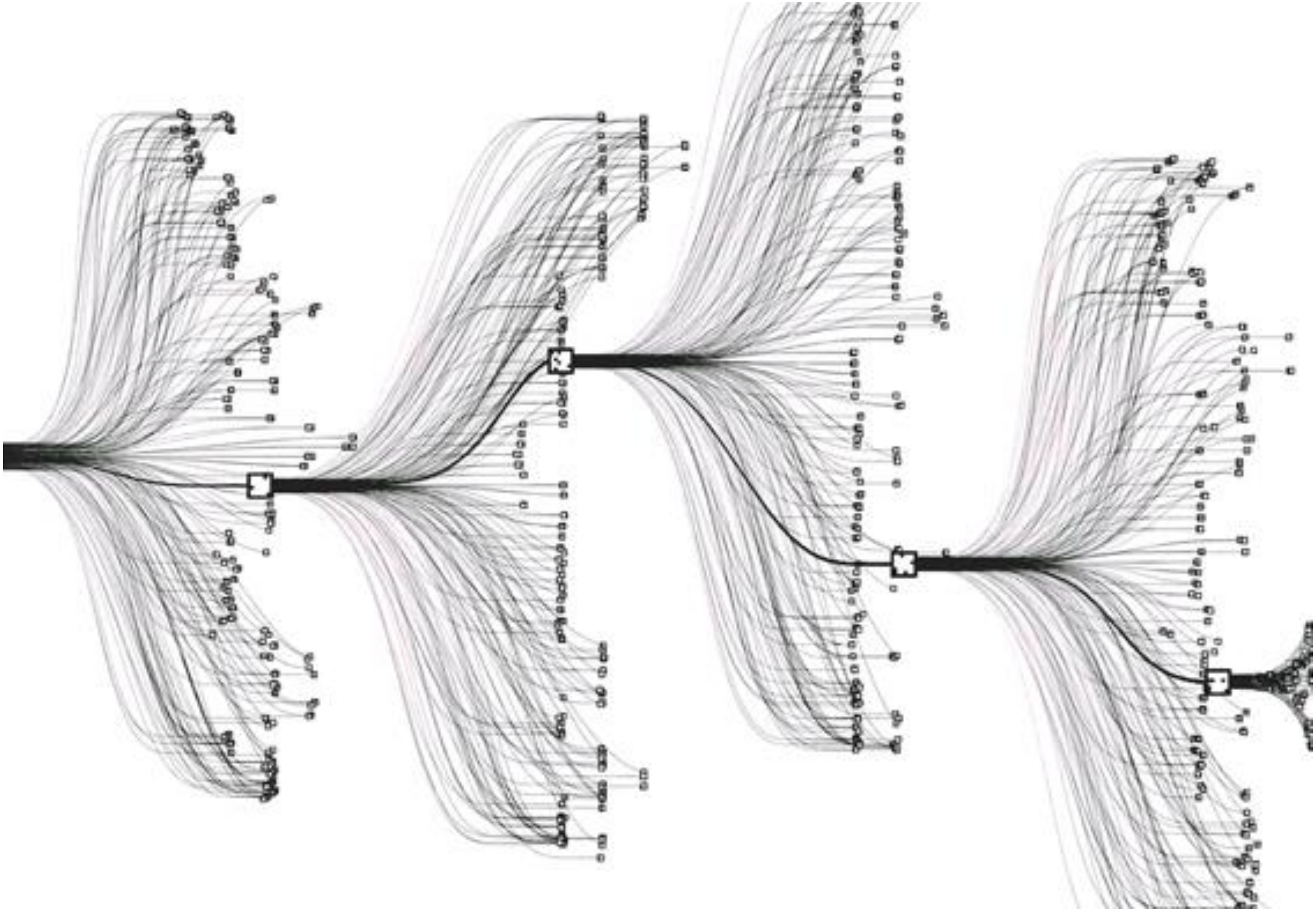
Instead of humans searching for implementations on StackOverflow, we prefer machines searching for implementations via intents.

Intents and implementations



An implementation is a path from the terminal/unit type to the intent.

Neural-symbolic program synthesis



Neural-symbolic program synthesis

Theorem proving

- CoqHammer
- CoqGym
- ProverBot
- GamePad

Program synthesis

- SketchAdapt
- DeepCoder
- DeepCode
- RobustFill
- Bayou
- GPT-3

Campbell, Mark. "Automated Coding: The Quest to Develop Programs That Write Programs." *Computer* 53, no. 2 (2020): 80-82.

Program synthesis

(joint work with Jin Xing Lim, Barnabe Monnot, Georgios Piliouras)

Synthesizing sort

Lessons learnt

Important to use good library of theorems about `sorted` and `perm_eq`.

Heavy use of reflection/transport to switch between prop and bool, and elaboration/unification to infer implicit/ambiguous arguments.

Many goals are trivial and solved automatically by good tactics.

Some goals were non-trivial. Opportunity to write better tactics that tackle other similar issues.

```
Theorem sort_spec (l : seq nat) : {l0 : seq nat & sorted leq l0 & perm_eq l l0}.
Proof.
  elim: l => [|a l [l0 s0 p0]].
    (* Base case of l *)
    by exists [::].
    (* Inductive case of l *)
    move: a l s0 p0.
    elim: l0 => [|b y IHy].
      (* Base case of l0 *)
      move => a l _ /perm_eq_nilP p0.
      by rewrite p0; exists [::a].
      (* Inductive case of l0 *)
      move => a l s0 H1.
      case: (leqP a b) => ab.
        (* Case of a <= b *)
        exists (a::b::y) => //=.
        by apply /andP.
        by rewrite perm_cons.
        (* Case of a > b *)
        case: (IHy a y) => {IHy} //=.                (* NON-TRIVIAL STEP *)
        move: (s0) => /path_sorted s0con //=.
        move => x H2 H3.
        exists (b::x) => //=.
        rewrite path_min_sorted //=.
          (* all (leq b) x *)
          move /(order_path_min leq_trans): s0 => s0.
          apply /allP.
          move: H3 s0 => /perm_eq_mem /eq_all_r <-s0 //=.
          move /ltnW: ab => ab.
          by rewrite ab s0.
          (* perm_eq (a :: l) (b :: x) *)
          apply/perm_eqP => //= P.
          move/perm_eqP: H1 => //= H1.
          move/perm_eqP: H3 => //= H3.
          rewrite H1; rewrite <- H3.
          by apply: addnCA.
Defined.
```

Extraction

```
Fixpoint sort (l: list nat)
  : list nat :=
    match l with
    | nil => nil
    | h::t => insert h (sort t)
    end.

Require Coq.extraction.Extraction.
Extraction Language OCaml.
Recursive Extraction sort.
```

Coq

Extracted code does not use OCaml primitives for bool, nat, list.

```
type bool = True | False
type nat = O | S of nat
type 'a list = Nil | Cons of 'a * 'a list

let rec leb n m =
    match n with
    | O -> True
    | S n' -> (match m with
               | O -> False
               | S m' -> leb n' m')

let rec insert i = function
| Nil -> Cons (i, Nil)
| Cons (h, t) ->
  (match leb i h with
   | True -> Cons (i, (Cons (h, t)))
   | False -> Cons (h, (insert i t)))

let rec sort = function
| Nil -> Nil
| Cons (h, t) -> insert h (sort t)
```

OCaml

https://softwarefoundations.cis.upenn.edu/vfa-current/Extract.html

Extraction

Extracting `sort_spec` to OCaml

Extracted code contains proofs of correctness of the algorithm which are not necessary for sorting.

```ocaml
let sort_spec l =
  let _evar_0_ = ExistT2 (Nil, __, __) in
  let _evar_0_0 = fun a l0 __top_assumption_ ->
    let _evar_0_0 = fun l1 ->
      let _evar_0_0 = fun a0 _ -> ExistT2 ((Cons (a0, Nil)), __, __) in
      let _evar_0_1 = fun b y iHy a0 _ ->
        let _evar_0_1 = fun _ -> ExistT2 ((Cons (a0, (Cons (b, y)))), __, __) in
        let _evar_0_2 = fun _ ->
          let _evar_0_2 = fun x -> ExistT2 ((Cons (b, x)), __, __) in
          let ExistT2 (x, _, _) = iHy a0 y __ __ in
          _evar_0_2 x in
        (match leqP a0 b with
         | LeqNotGtn -> _evar_0_1 __
         | GtnNotLeq -> _evar_0_2 __) in
      let rec f l2 a0 l3 = match l2 with
        | Nil -> _evar_0_0 a0 l3
        | Cons (y, l4) -> _evar_0_1 y l4 (fun a1 l5 _ _ -> f l4 a1 l5) a0 l3 in
      f l1 a l0 in
    let ExistT2 (x, _, _) = __top_assumption_ in
    _evar_0_0 x in
  let rec f = function
  | Nil -> _evar_0_
  | Cons (y, l1) -> _evar_0_0 y l1 (f l1) in
  f l
```

OCaml

Implementation theory

Type of expressions/programs
in the implementation language.
A theory is a language with equalities.

```
Inductive exp : Set :=
| Const : nat -> exp
| Plus  : exp -> exp -> exp
| Times : exp -> exp -> exp
```

Semantics for the
implementation language.
[Chlipala 2013]

```
Fixpoint denote (e : exp) : nat :=
  match e with
    | Const n => n
    | Plus  e1 e2 => plus (denote e1) (denote e2)
    | Times e1 e2 => mult (denote e1) (denote e2)
  end.
```

Using the proof assistant to construct
a program with the right semantics.

```
Theorem sort_spec_exp :
  ∑ (f : exp), ∀ (l : seq nat),
  let l0 := (denote f) l in
  (sorted leq l0) ∧ (perm_eq l l0).
```

Chlipala, Adam. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.

Verification vs synthesis

It is easier to check a solution (e.g. check factorization of a large integer) than to find a solution (e.g. find the factors of a large integer).

Software verification and synthesis both involve formal methods, but they require completely different tools, strategies and mindsets. (Compare bottom-up verification and top-down synthesis of insertion sort.)

In verification, all layers of software stack needs to be scrutinized to uncover system loopholes and avoid giving false guarantees.

In synthesis, we make assumptions about the semantics of primitive instructions in our software layer and focus on deriving new behavior. Engineers working on lower software layers verify those assumptions.

Program representation

Synthesized programs need not be extracted/compiled (e.g. from Coq to OCaml) before storage. They can be saved in compressed form and extracted at run-time. We could even store them via the proof script that synthesized the program.

Andrej Bauer on representation of mathematical theorems:

> We do not expect humans to memorize every proof of every mathematical statement they ever use, nor do we imagine that knowledge of a mathematical fact is the same thing as the proof of it. Humans actually memorize *proof ideas* which allow them to replicate the proofs whenever they need to. Proof assistants operate in much the same way, for good reasons.

Proof assistants facilitate translations from implementation languages to semantics (e.g. from structured cospans to algorithms) as well as from proof scripts to proof terms.

They call on type-engines to check if proof terms are well-formed or well-typed. They could be engine-agnostic or even theory-agnostic. (CatLab as proof assistant?)

http://math.andrej.com/2016/08/09/what-is-a-formal-proof/

Blockchain and incentives

Proofs/implementations can be made opaque or transparent.

Theorems/intents that use a prior result should depend only on its type and not the syntax of its term, unless it is a definition or ontology. Theorems should not break when the proof of a prior result is changed.

For an opaque proof/implementation, we can use blockchain to store the verification that it is well-typed without revealing its syntax.

We can use smart contracts to collect usage payment and deliver its syntax or API to other intents that need it for synthesis. These contracts create a hierarchy of incentives for conjectures or software goals.

Transport between theories

Library coq.sorting.permutation

```
Inductive Permutation : list A -> list A -> Prop :=
| perm_nil: Permutation [] []
| perm_skip x l l' : Permutation l l' -> Permutation (x::l) (x::l')
| perm_swap x y l : Permutation (y::x::l) (x::y::l)
| perm_trans l l' l'' : Permutation l l' -> Permutation l' l''
                                          -> Permutation l l''.
```

Library mathcomp.ssreflect.seq

```
Definition perm_eq s1 s2 :=
  all [pred x | count_mem x s1 == count_mem x s2] (s1 ++ s2).
```

How do we effectively indicate that the two definitions are equivalent?

How do we transport a theorem/intent from one theory to another?

Transport between types

More generally, how do we transport between equivalent types?

Example [Tabareau, Tanter & Sozeau 2019]
Types `Nat` (unary numbers) and `Bin` (binary numbers) are equivalent.
How do we effectively transport functions (e.g. addition, multiplication) and theorems (e.g. commutativity of addition) from one type to another?

Possible solutions

1. Boolean reflection
2. Parametric transport
3. Cubical transport
4. Univalent transport

Tabareau, Nicolas, Éric Tanter, and Matthieu Sozeau. "The Marriage of Univalence and Parametricity." *arXiv preprint arXiv:1909.05027* (2019).

Transport between types

Example. Sorting algorithms (joint work with Jin Xing Lim, Georgios Piliouras)

Sorting is writing the elements of a finite totally-ordered **set** in order to a **list**.

Insertion sort involves transporting/lifting this **set** to a **list**, and
applying the induction principle for **lists** as a tactic.

Merge sort involves transporting/lifting this **set** to a **binary tree**, and
applying the induction principle for **binary trees** as a tactic.

Quick sort involves transporting/lifting this **set** to a **binary search tree**, and
applying the induction principle for **binary search trees** as a tactic.

We may generalize and apply these tactics to any goal that
require the synthesis of functions over finite sets or lists,
e.g. search, maximum, minimum, average.

Unification

To decide if a prior theorem may be applied to a goal, we need unification.

```
Goal forall (a : nat) (l : list nat), sorted l -> sorted (insert a l)
```

```
Theorem sorted_ind : forall (P : list nat -> Prop),
  P [] ->
  (forall x : nat, P [x]) ->
  (forall (x y : nat) (l : list nat),
    x <= y -> sorted (y :: l) -> P (y :: l) -> P (x :: y :: l)) ->
      forall (l : list nat), sorted l -> P l
```

```
Unify forall (l : list nat), sorted l -> sorted (insert a l)
with  forall (l : list nat), sorted l -> ?P l
```

The solution to the above unification problem is

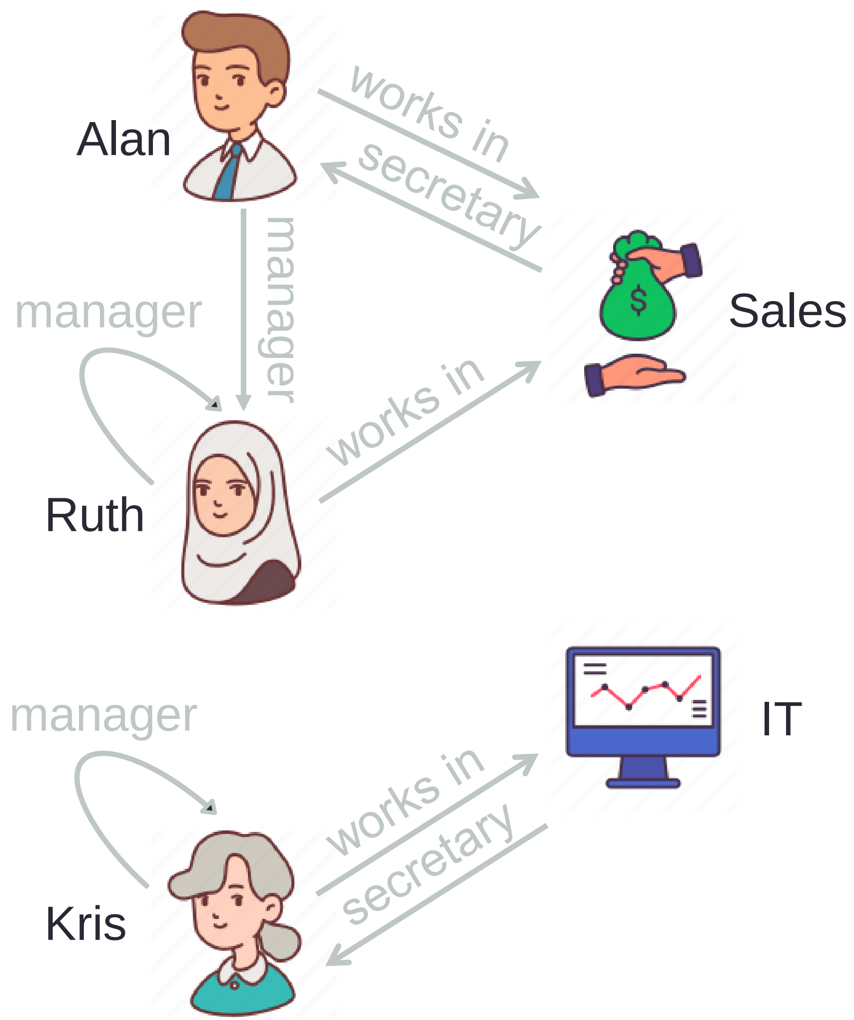
```
?P := fun l0 : list nat => sorted (insert a l0)
```

Unfolding a definition (δ-reductions) can break unification,
so it is not always advisable to unify normal forms with normal forms.

Knowledge Graphs

(joint work with Zhangsheng Lai, Liang Ze Wong)

Knowledge instances



Queries as types, answers as terms

```
Inductive empl :=
| alan
| ruth
| kris.

Inductive dept :=
| sales
| tech.
```

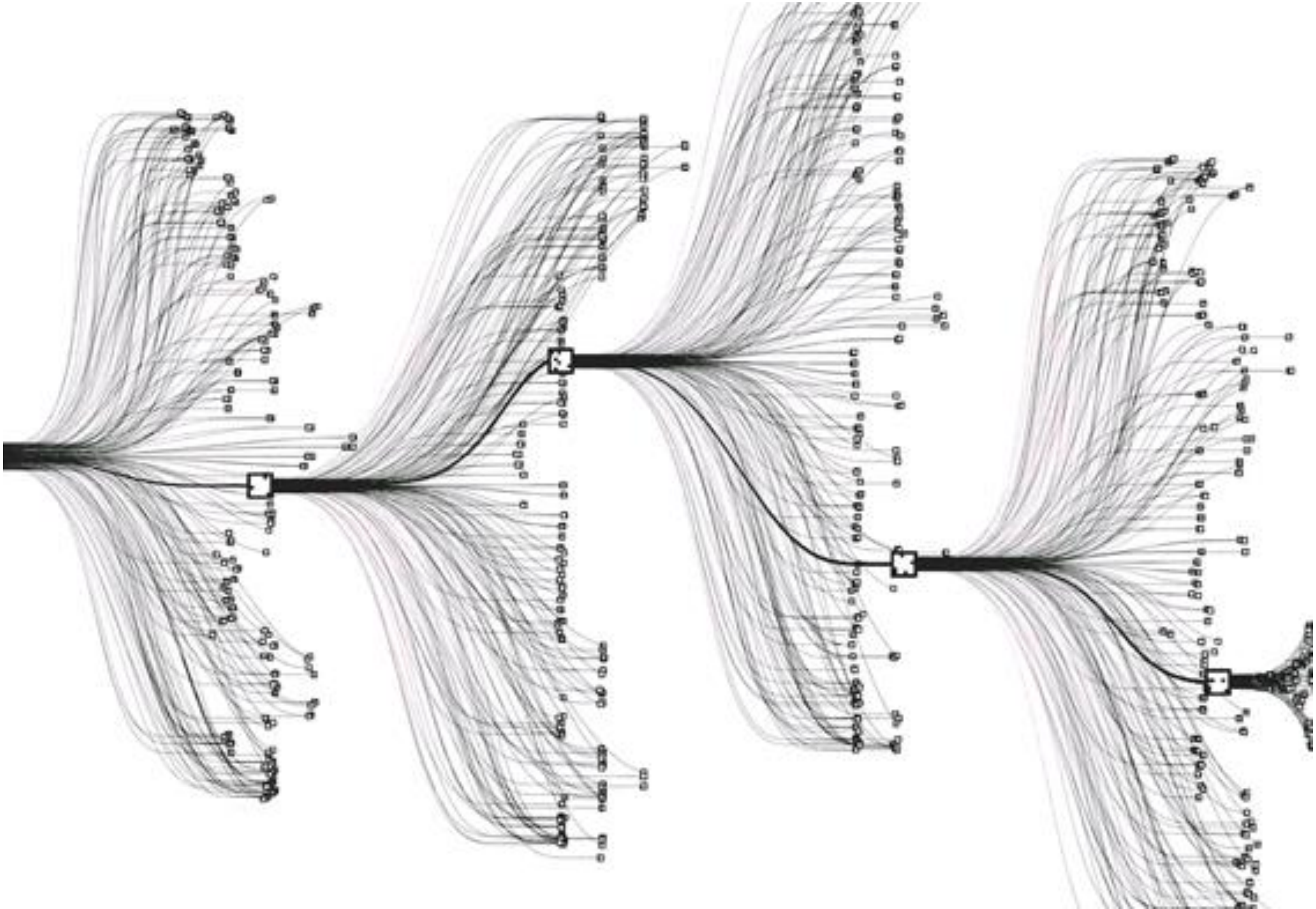
```
Definition works (e : empl) : dept :=
  match e with
  | alan => sales
  | ruth => sales
  | kris => tech
  end.

Definition mgr (e : empl) : empl :=
  match e with
  | alan => ruth
  | ruth => ruth
  | kris => kris
  end.

Definition sec (e : dept) : empl :=
  match e with
  | sales => alan
  | tech => kris
  end.
```

Queries as types, answers as terms

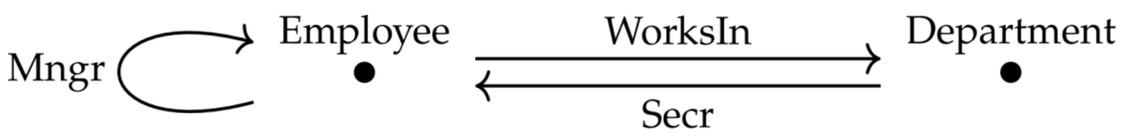
```
Structure ruth_sec_qry :=
{ ruth_sec  :> empl;
  ruth_dept : dept;
  eq_ruth_dept : ruth_dept = works ruth;
  eq_ruth_sec  : ruth_sec  = sec ruth_dept }.

Definition ruth_sec_ans : ruth_sec_qry.
Proof. unshelve eexists. Focus 3. auto. Focus 2. auto. Defined.
```

```
ruth_sec_ans =
{| ruth_sec  := alan;
   ruth_dept := sales;
   eq_ruth_dept := erefl sales;
   eq_ruth_sec  := erefl alan |}
```

Queries as types, answers as terms



Knowledge schemata



Department.Secr.WorksIn = Department

Employee.Mngr.WorksIn = Employee.WorksIn

[adapted from Fong & Spivak 2018]

```
Theorem eq_sec : forall d, works (sec d) = d.
Proof. destruct d; auto. Qed.

Theorem eq_mgr : forall e, works (mgr e) = works e.
Proof. destruct e; auto. Qed.
```

Fong, Brendan, and David I. Spivak. "Seven sketches in compositionality: An invitation to applied category theory." *arXiv preprint arXiv:1803.05316* (2018).

Knowledge schemata

We may use **`Structure`** to organize data about a schema,

```
Structure company : Type := Build_company
{ employee     : Set;
  department   : Set;
  works_in     : employee   -> department;
  secretary    : department -> employee;
  manager      : employee   -> employee;
  eq_secretary : forall d, works_in (secretary d) = d;
  eq_manager   : forall e, works_in (manager e) = works_in e }.
```

declare an instance of the schema,

```
Definition acme := Build_company empl dept works sec mgr eq_sec eq_mgr.
```

and prove theorems about all instances of the schema.

```
Theorem sec_mgr : forall (c : company) (d : department c),
  works_in c (manager c (secretary c d)) = d.
Proof. intros c d. rewrite <- eq_secretary. rewrite eq_manager. auto. Qed.
```

Mathematical structures

We may use **Structure** to organize data about a mathematical structure,

```
Structure abGrp : Type := AbGrp {
  carrier : Type;                zero : carrier;
  opp : carrier → carrier;       add : carrier → carrier → carrier;
  add_assoc : associative add;  add_comm : commutative add;
  zero_idl : left_id zero add;  add_oppl : left_inverse zero opp add }.
```

declare an instance of the mathematical structure,

```
Definition Z_abGrp := AbGrp Z Z0 Z1 Zopp Zadd ....
```

and prove theorems about all instances of the mathematical structure.

```
Theorem subr0 : ∀ (aG : abGrp) (x : carrier aG),
  add aG x (opp aG zero) = x.
```

Unification

How can a proof assistant know how to apply

```
Theorem sec_mgr : forall (c : company) (d : department c),
  works_in c (manager c (secretary c d)) = d.
```

towards simplifying `works (mgr (sec d))` to a department `d` unless we explicitly specify `works`, `mgr`, `sec` as fields to of an instance of `company`?

How can a proof assistant know how to apply

```
Theorem subr0 : ∀ (aG : abGrp) (x : carrier aG),
  add aG x (opp aG zero) = x.
```

towards simplifying `(Zadd z (Zopp Z0))` to an integer `z` unless we explicitly specify `Zadd`, `Zopp`, `Z0` as fields of an instance of `abGrp`?

This is a problem in *unification*.

```
works_in ?c  = works
add      ?aG = Zadd
```

Canonical instances

As we build complex hierarchies of structures in knowledge graphs, users should not be burdened by the tracking of structural information.

One solution is to declare some instances of a structure as *canonical*, e.g. the set of integers can be viewed as an additive or multiplicative monoid, but we could declare the additive monoid instance as canonical.

```
Definition acme := Build_company empl dept works sec mgr eq_sec eq_mgr.
Canonical acme : company.

Definition Z_abGrp := AbGrp Z Z0 Z1 Zopp Zadd ....
Canonical Z_abGrp : abGrp.
```

Coq adds each field of a canonical instance to a look-up table for unification.

```
employee   ?c = empl   =>  ? = acme
department ?c = dept   =>  ? = acme
...
```

(We could also use type classes in Coq but I'm agnostic.)

Mahboubi, Assia, and Enrico Tassi. "Canonical structures for the working Coq user." In *International Conference on Interactive Theorem Proving*, pp. 19-34. Springer, Berlin, Heidelberg, 2013.

Verification vs enumeration

Given finite type T (e.g. the world population) and $t : T$,
suppose we are interested in the subtype S_t (e.g. the siblings of t).

Verification (e.g. checking that $s : T$ is a sibling)
is easier than synthesis (e.g. finding any sibling),
which is easier than enumeration (e.g. finding all siblings).

Verification specified with Boolean predicates $p : (t : T) \to (s : T) \to \mathrm{bool}$.
Enumeration specified with enumeration functions $f : (t{:}T) \to \mathrm{list}\, T$.
Equivalence between predicate and enumeration $p\; t\; s = \mathrm{true} \leftrightarrow s \in f\; t$.

Of course, given a predicate p, we could enumerate by evaluating p on all of T.
However, *scheduling* [Patterson 2020] can give us more efficient enumerations.
How to schedule by transporting between predicates and enumerations?

Patterson, Evan. "(Co)relational computing in Catlab: The operad of UWDs and its algebras", MIT Categories Seminar, December 2020.

Transport by reflection

In the ssreflect library, the type `reflect P b` encodes the equivalence $P \leftrightarrow (b = \mathrm{true})$.

```
Inductive reflect (P : Prop) : bool -> Prop :=
| ReflectT (p : P)     : reflect P true
| ReflectF (np : ~ P) : reflect P false.
```

For example, this lemma states the equivalence $(b1 \wedge b2) \leftrightarrow (b1 \,\&\&\, b2 = \mathrm{true})$.

```
Lemma andP (b1 b2 : bool) : reflect (b1 /\ b2) (b1 && b2).
```

We apply the lemma to transport a proof of `(a && b = true)` to a proof of `ab : a /\ b`.

```
Lemma example a b :
  a && b ==> (a == b).
Proof.
  case: andP => [ab|nab].
```

```
a, b : bool
ab : a /\ b
========================
true ==> (a == b)

subgoal 2 is:
false ==> (a == b)
```

Transport by reflection

We use the type `(qreflect qt qe)` to encode the equivalence between membership in a subtype `(n \of qt)` and membership in a list `(n \in qe)`.

```
Definition qreflect {qb : query_pred} (qt : query_subType qb)
  (qe : query_enum) := ∀ (n : node), reflect (n \of qt) (n \in qe).
```

Suppose we have relations R_1, R_2 and their enumerations $E_i(s) = \{\, t \mid t\ R_i\ s \,\}$. Define relation R_3 by $u\ R_3\ s \Leftrightarrow \exists\, t, (u\ R_2\ t) \land (t\ R_1\ s)$. Let $E_3(s) = \{\, u \mid u\ R_3\ s \,\}$. The next theorem says that $E_3(s)$ is the union of $E_2(t)$ over all $t \in E_1(s)$.

```
Theorem qChainP _ _ _ _
          (_ : ∀ (n:node), qreflect (qr1 n) (qe1 n))
          (_ : ∀ (n:node), qreflect (qr2 n) (qe2 n)) : ∀ (s u : node),
            reflect (exists2 t : node, (t \of qr1 s) & (u \of qr2 t))
                    (u \in flatmap (fun t => qe2 t) (qe1 s)).
```

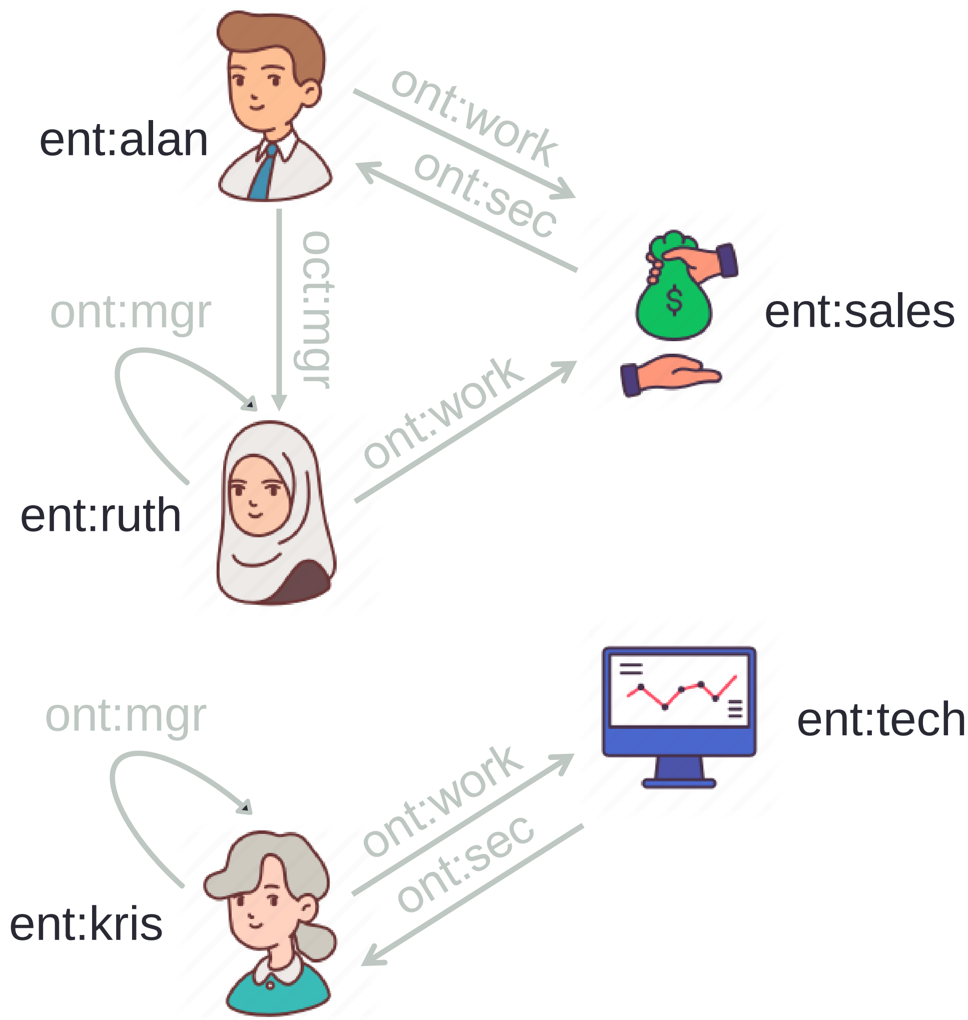
With more equivalences like `qChainP`, the proof assistant can derive enumerations for new queries by decomposing them into patterns and applying reflections.

Decentralized knowledge

In linked data, knowledge is not centralized in one database where we can derive inductive types.

```
Inductive empl :=
| alan
| ruth
| kris.

Inductive dept :=
| sales
| tech.
```

Moreover, access rights prevent a user from reading all available entries in the knowledge graph.



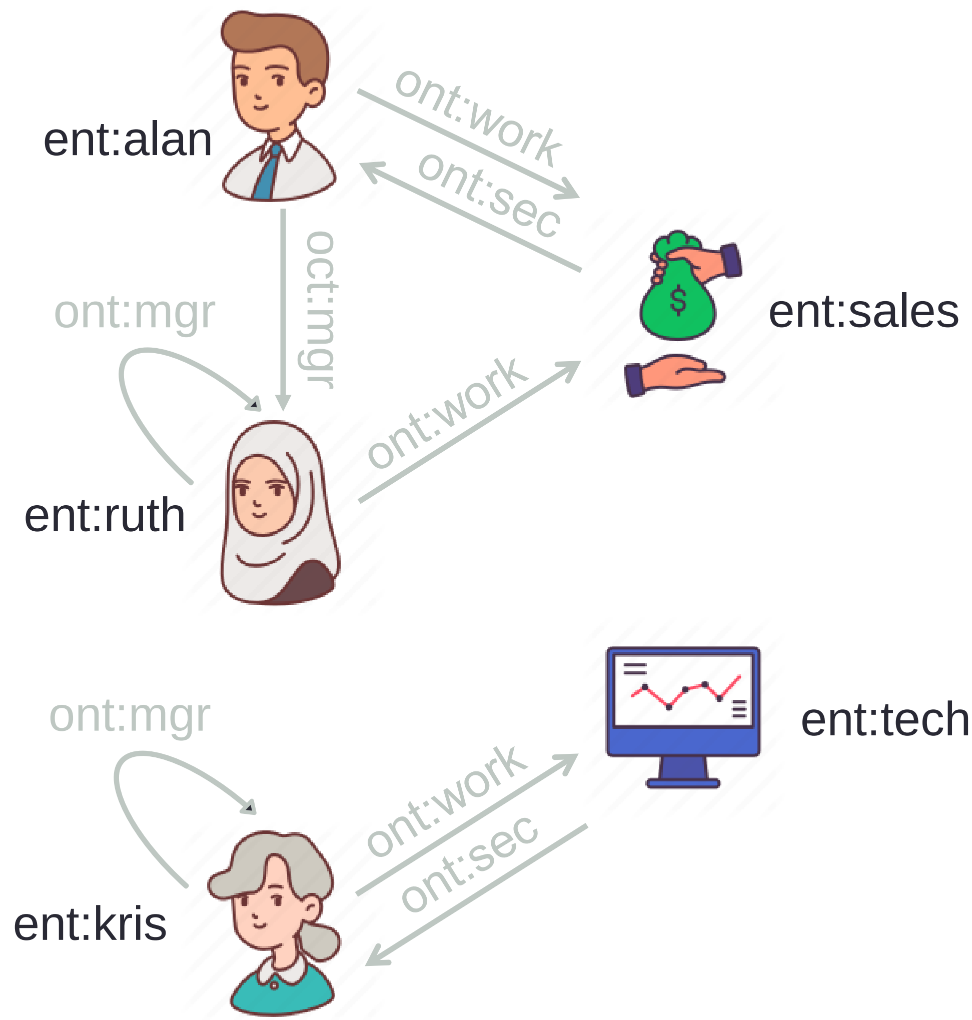
PREFIX ent: <http://acme.com/entity#>
PREFIX ont: <http://acme.com/ontology#>

Universe management

The universe of a user is thus built up by axiomatic types and terms in the same way a theory is built up in a logical framework.

```
Axiom empl : Type.
Axiom alan : empl.
Axiom ruth : empl.
Axiom kris : empl.

Axiom dept  : Type.
Axiom sales : dept.
Axiom tech  : dept.
```

Modules are imported to construct types on top of this user universe. Users may share universes and branch off at different levels. Polymorphism is necessary for localizing types at different levels.



PREFIX ent: <http://acme.com/entity#>
PREFIX ont: <http://acme.com/ontology#>

Conclusion

Program synthesis

Key research areas to fulfill the dream of using proof assistants for program synthesis.

- Transport
- Unification
- Theory management
- Universe management

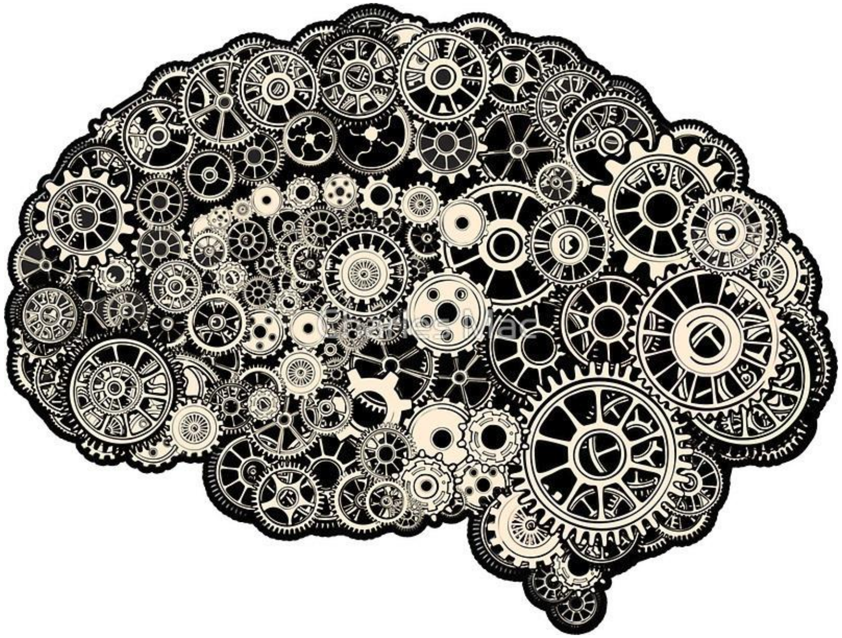
Diversity, equity, inclusion

Information is power. Code is access to power.

People are stripped of power by inability to code, or inability to control information because of code (e.g. platforms for social networks, gig economy).

- Diversity of voices, products, services
- Equal access to resources and growth
- Inclusion in supportive communities

Questions?



https://shaoweilin.github.io/