

Semantics of Recursive Types

CS-642 project report

Matt Bovel

2026-06-31

Abstract

We mechanize three logical soundness proofs of equi-recursive types built on a definitional interpreter and semantic types. All three rely on step-indexing to make the interpretation of recursive types well-founded, but differ in how the missing step is accounted for: later types, step-indexed typing judgements, and a contractiveness condition adapted from Appel and McAllester.

Keywords: definitional interpreters, simply-typed lambda calculus, semantic types, recursive types

Introduction

The goal of this project is to study the semantics of *equi-recursive types*, using a *definitional interpreter* and *semantic types*. We introduce these notions in this section.

Our approach differs from most modern developments, which use small-step semantics and iso-recursive types (Timany et al. 2024). We follow our previous work (Bovel et al. 2026): a definitional interpreter, whose advantages are detailed in (Amin and Rompf 2017), enables simpler proofs in the context of DOT, and equi-recursive types enable subtyping.

Syntax

Our base for this project is the simply-typed lambda calculus, with booleans. We add recursive types to this base setup in the next sections.

The syntax of types and terms is what one would expect. In addition, we define a separate syntax for values, which are returned by the interpreter:

```
Inductive Ty : Type :=
  | TBool : Ty
  | TFun : Ty → Ty → Ty.
```

```
Inductive Term : Type :=
  | tbool : bool → Term
  | tvar : nat → Term
  | tabs : Ty → Term → Term
  | tapp : Term → Term → Term.
```

```
Inductive Val : Type :=
  | vbool : bool → Val
  | vabs : (list Val) → Term → Val.
```

We use de Bruijn indices for variables and represent the environment as a list of values. Closure values capture their environment.

Definitional Interpreters

The idea of a definitional interpreter is to define the semantics of a language as a function, and not as an inductive definition. This makes it total and deterministic by construction.

To make it terminating, a fuel parameter is decremented at each recursive call. Crucially, here the fuel bounds the *depth* of the evaluation, not the number of steps.

The result is a layered option type: `None` means a timeout, `Some None` means that there was an error, and `Some (Some v)` means that the term evaluates to the value `v`.

```
Fixpoint eval (fuel: nat) (env: list Val)
  (t: Term) : option (option Val) :=
  match fuel with
  | 0 ⇒ None
  | S fuel' ⇒
    match t with
    ...
    | tapp f a ⇒
      match (eval fuel' env f) with
      | None ⇒ None
      | Some (Some (vabs envf b)) ⇒
```

```

      match (eval fuel' env a) with
      | None ⇒ None
      | Some None ⇒ Some None
      | Some (Some va) ⇒
        eval fuel' (va::envf) b
      end
    | _ ⇒ Some None
  ...

```

Semantic Types

A semantic type $\text{SemTy} := \text{Val} \rightarrow \text{Prop}$ is a predicate on values. It defines what it means for a value to have a type.

The *value interpretation* defines what it means for a Val to have a syntactic type. Note that this can be seen as a function $\text{Ty} \rightarrow \text{SemTy}$; it gives the logical denotation of a type:

```

Fixpoint interp (T: Ty) (v: Val) : Prop :=
  match T with
  | TBool ⇒ exists b, v = vbool b
  | TFun A B ⇒
    exists env body, v = vabs env body
    ∧ forall arg, interp A arg →
      tinterp (arg::env) body (interp B)
  end.

```

The *term interpretation* defines what it means for a *term* to have a *semantic type*. A term has a semantic type if it evaluates to a value (given enough fuel) and that value has the semantic type:

```

Definition tinterp (env: list Val)
  (t: Term)
  (T: Val → Prop) : Prop :=
  exists v fuel,
    eval fuel env t = Some (Some v) ∧ T v.

```

The *semantic typing judgement* is defined as:

```

Definition sem_typed (tenv: list Ty)
  (t: Term)
  (T: Ty) : Prop :=
  forall env,
    Forall2 interp tenv env →
      tinterp env t (interp T).

```

We write it $\Gamma \models t : T$. In the usual presentations, the typing judgement is often defined inductively. In our setting, we define this relation semantically, and then *prove* the typing lemmas are correct. For example, the application rule corresponds to the following lemma:

```

Lemma sem_typed_app: forall tenv f a A B,
  sem_typed tenv f (TFun A B) →
  sem_typed tenv a A →
  sem_typed tenv (tapp f a) B.

```

Recursive Types

A recursive type, written $\mu X.T$, is a type that is defined in terms of itself. The binder X can appear in T , and represents the type itself. Unrolling the recursive type, by substituting X with $\mu X.T$ in T , yields a type equivalent to the original one: $\mu X.T \equiv T[X \mapsto \mu X.T]$. This equivalence is captured by the following typing rules:

$$\frac{\Gamma \vdash t : \mu X.T}{\Gamma \vdash t : T[X \mapsto \mu X.T]} \quad \frac{\Gamma \vdash t : T[X \mapsto \mu X.T]}{\Gamma \vdash t : \mu X.T}$$

The goal of this project is to give a semantics to recursive types in the setup described above, and to prove the above typing rules. More precisely, this will mean extending the definition of `interp` and of `sem_typed` to reason about recursive types.

Approaches

In this section, we describe the three main approaches we explored to give a semantics to recursive types: later types, step-indexed judgements, and a contractiveness condition. All three build on the same foundation, a *step-indexed* value interpretation, which we introduce first.

Step-indexed relation

We want $\mu X.T \equiv T[X \mapsto \mu X.T]$. However, such a direct formulation would not be well-founded, as the definition of `interp` would be circular. To make it well-founded, we add a step-index parameter to the interpretation of types, which is decremented at each recursive call:

```

Equations? interp (T: Ty) (k: nat)
  (v: Value) : Prop
by wf (k, ty_size T)
(Equations.Prop.Subterm.lexprod _ _ lt lt) :=
...
interp (TMu B) 0 v := True;
interp (TMu B) (S k') v :=
  interp (subst_ty 0 (TMu B) B) k' v.

```

We use the Equations plugin to define this function, allowing us to use a lexicographic order on the step-index and the size of the type to ensure termination.

In this framework, a semantic type is now defined as $\text{SemTy} := \text{nat} \rightarrow \text{Val} \rightarrow \text{Prop}$.

Later types

If we use the above definition directly, the equation that we get is actually $\forall k, (\mu X.T)^k \equiv T[X \mapsto \mu X.T]^{k-1}$.

A simple way to account for this at the syntactic level is to introduce a new type constructor, which we call *later*, written $\triangleright T$ meaning “one step smaller, T will hold”:

```
interp (TLater T') 0 v := True;
interp (TLater T') (S k') v := interp T' k' v;
```

We can then prove the following typing rules for recursive types:

$$\frac{\Gamma \models t : \triangleright T[X \mapsto \mu X.T]}{\Gamma \models t : \mu X.T} \quad \frac{\Gamma \models t : \mu X.T}{\Gamma \models t : \triangleright T[X \mapsto \mu X.T]}$$

This approach is demonstrated in `stlc_rec_later.v`.

Additionally, we can get rid of the later type in the folding by forcing `interp` to be *downward closed*: if a value is in the interpretation of a type at step k , then it is also in the interpretation of that type at any smaller step:

```
Lemma interp_dc: forall T k v,
  interp T (S k) v → interp T k v.
```

We achieve this by quantifying over all smaller step-indices in the interpretation of function types, and of recursive types:

```
interp (TFun A B) k v :=
  exists env body, v = vabs env body
  ∧ forall j (Hj: j ≤ k) arg,
    interp A j arg →
    tinterp (arg::env) body (interp B j);
...
interp (TMu B) k v :=
  forall j (Hj: j < k),
    interp (subst_ty 0 (TMu B) B) j v.
```

This is demonstrated in `stlc_rec_later_dc.v`.

Later types have been used in gDOT, a semantic version of DOT built on Iris (Giarrusso et al. 2020).

Step-indexed judgement

To avoid polluting the syntax of types with later types, we can instead add a step-index to the typing judgement itself. This lets us account for the missing step in the unfolding rule without introducing later types:

$$\frac{\Gamma \models_k t : T[X \mapsto \mu X.T]}{\Gamma \models_k t : \mu X.T} \quad \frac{\Gamma \models_k t : \mu X.T}{\Gamma \models_{k-1} t : T[X \mapsto \mu X.T]}$$

To be able to type-check applications, we also need to parametrize function types by a step-index, which is used in the application rule:

$$\frac{\Gamma \models_i f : A \rightarrow_j B \quad \Gamma \models_k a : A}{\Gamma \models_{i+j+k} f a : B}$$

This approach is demonstrated in `stlc_rec_indexed.v`.

We also tried to existentially quantify over the step-index in the interpretation of recursive types, but this unfortunately breaks: depending on how we scope it, it is either too strong or too weak, and never works for both applications and abstractions. Our attempt can be found in `stlc_rec_indexed_exist.v`.

Contractiveness condition

Last but not least, we reproduced a classic approach to give a semantics to recursive types, which is to require that the recursive type is *contractive* (Appel and McAllester 2001).

This approach is based on the notion of “well-foundedness” of a function $F : \text{SemTy} \rightarrow \text{SemTy}$, defined as:

```
Definition approx (k: nat) (T: SemTy): SemTy :=
  fun j v ⇒ j < k ∧ T j v.
```

```
Definition well_founded (F: SemTy → SemTy)
  : Prop :=
```

```
  forall k Z,
    approx (S k) (F Z)
    = approx (S k) (F (approx k Z)).
```

and a fixed-point combinator μ , where $\mu F = F^{k+1}(\perp)$:

```

Fixpoint fun_pow {A: Type} (k: nat)
  (f: A → A) (z: A) : A :=
  match k with
  | 0 ⇒ z
  | S k ⇒ f (fun_pow k f z)
  end.

```

```

Definition Bot : SemTy := fun _ _ ⇒ False.

```

```

Definition mu_F (F: SemTy → SemTy) : SemTy :=
  fun k v ⇒ fun_pow (S k) F Bot k v.

```

Appel and McAllester (Appel and McAllester 2001) prove the following property of `mu_F` applied to a well-founded function `F`:

```

Theorem theorem_18: forall F,
  well_founded F →
  mu_F F = F (mu_F F).

```

We reproduce these results by mechanizing the paper's main lemmas in `appel_lemmas.v`, adapting its set-based definitions to our predicate-based setting.

How do we apply this property to our setting, i.e. what should our `F` be?

Our previous equation for recursive types did the heavy lifting at the syntactic level, using substitution. But `mu_F` is a semantic combinator. To better suit this semantic definition, we switch to an environment-based interpretation, where `TMu` can be expressed directly in terms of `mu_F`:

```

Fixpoint interp (tenv: list SemTy) (T: Ty)
  (k: nat) (v: Val) : Prop :=
  match T with
  | TVar i ⇒
    match nth_error tenv i with
    | Some A ⇒ A k v
    | None ⇒ False
    end
  | TBool ⇒ exists b, v = vbool b
  | TFun A B ⇒
    exists env body, v = vabs env body ∧
    forall j (Hj: j < k) arg,
      interp tenv A j arg →
      tinterp (arg::env) j body
      (fun j' w ⇒ interp tenv B j' w)
  | TMu B ⇒
    mu_F (fun Me ⇒ interp (Me :: tenv) B) k v
  end.

```

So our `F` is `fun Me ⇒ interp (Me :: tenv) B`! To make it well-founded, we intuitively need to prove that

`interp (Self :: tenv) B k` only uses `Self` at steps `j < k`. But this is not true in general: for example, the interpretation of `TMu (TVar 0)` at step `k` definitely uses itself at step `k`.

The trick is to make all *other* constructors under which binders could appear decrease a step — here, only `TFun`. We achieve this by changing the quantification from `j ≤ k` to `j < k`. Thanks to this change, any `TVar` *inside a function type* only uses types from the environment at steps `j < k`. The same holds for `TBool`, as it does not use the environment at all.

We formalize this by the notion of *contractiveness*:

```

Definition contractive (T: Ty) : Prop :=
  match T with
  | TVar _ ⇒ False
  | TBool ⇒ True
  | TFun _ _ ⇒ True
  | TMu _ ⇒ False
  end.

```

which allows us to prove the lemma:

```

Lemma F_well_founded: forall tenv B,
  contractive B →
  well_founded (fun Me ⇒ interp (Me :: tenv) B).

```

which in turn allows us to prove:

```

Lemma interp_mu_unfold: forall tenv B,
  contractive B →
  interp tenv (TMu B) =
  interp ((interp tenv (TMu B)) :: tenv) B.
Proof.
  intros **.
  exact (theorem_18 _ (F_well_founded tenv B H)).
Qed.

```

This is the crucial lemma (along with a classic lemma relating environment-based and substitution-based interpretations) to prove the following rules:

$$\frac{\text{contractive } T \quad \Gamma \models t : T[X \mapsto \mu X.T]}{\Gamma \models t : \mu X.T}$$

$$\frac{\text{contractive } T \quad \Gamma \models t : \mu X.T}{\Gamma \models t : T[X \mapsto \mu X.T]}$$

which are exactly the rules we wanted to prove, modulo the contractiveness requirements.

One last change is needed for the application rule to go through. Now that the function interpretation only constrains the body at strictly smaller steps ($j < k$), evaluating the body of an application must happen at a strictly smaller step-index than the application itself. This requires two changes.

First, we make the interpreter count *global* steps rather than recursion depth. Instead of evaluating each subterm with the same decremented fuel, we thread the leftover fuel through each sub-evaluation. To do so, `eval_sigma` returns its remaining fuel in a sigma type $\{\text{fuel}' : \text{nat} \mid \text{fuel}' < \text{fuel}\}$, which guarantees that the returned fuel is strictly less than the input one:

```
Equations? eval_sigma (fuel: nat)
                  (env: list Val)
                  (t: Term) :
  option (option (
    {fuel' : nat | fuel' < fuel} * Val))
  by wf fuel lt :=
```

We wrap it into an `eval` that hides the sigma type for convenience, which returns a result of type `option (option (nat * Val))`.

Second, and this is in fact standard in modern step-indexed models, the term interpretation relates the evaluation steps of the term to the step-index of its type's interpretation. More precisely, we require a term to have its type only *at the remaining fuel* after evaluation:

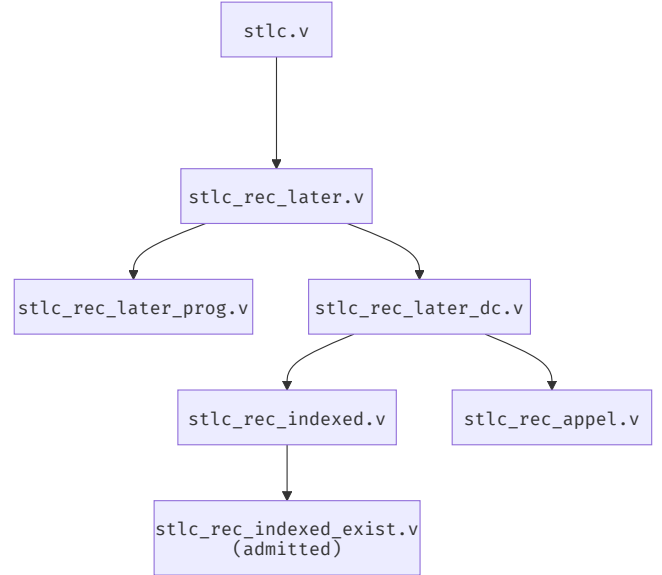
```
Definition tinterp (env: list Val) (k: nat)
                  (t: Term)
                  (T: SemTy) : Prop :=
  forall r, eval k env t = Some r →
    exists k' v, r = Some (k', v) ∧ T k' v.
```

Together, these ensure the body of an application is evaluated, and thus typed, at a strictly smaller step-index, corresponding to the function interpretation.

This approach is demonstrated in `stlc_rec_appel.v`.

Results

The approaches described above are all mechanized in Rocq (The Rocq Development Team 2025). The arrows in the diagram below mean “builds on”. While each file repeats the definitions of the previous ones, we tried to minimize the diff (except for `stlc_rec_appel.v`, which is almost completely different) to make it easier to follow the progression of ideas and the isolated impact of each change.



`stlc_rec_later_prog.v` was an attempt to formulate interp using a Program Fixpoint instead of Equations, which was strictly worse.

In addition to the files in the diagram, `list_lemmas.v` contains some lemmas about lists, `tactics.v` contains some custom tactics, and `appel_lemmas.v` contains the mechanization of the lemmas from Appel’s paper.

Timeline

- Week 7-8: base setup (`stlc.v`) and first formulations with recursive types, both with one-step mismatches.
- Week 9: skipped.
- Week 10: review of existing approaches, mechanization of Appel’s lemmas (`appel_lemmas.v`).
- Week 11-12: a few experiments with existentials, but mostly skipped.
- Week 13: later types (`stlc_rec_later.v`, `stlc_rec_later_dc.v`) and step-indexed judgments (`stlc_rec_indexed.v`).
- Week 14: working version with the contractiveness condition (`stlc_rec_appel.v`), presentation and report write-up.

The planned and executed timelines are close, though the agenda was pushed towards the end of the semester.

Related work

Already cited where relevant.

Future work

We are mainly interested in further exploring the contractiveness condition approach. First, we would like to validate that it scales to System F, dependent types, and union/intersection types. If so, we could adapt (Bovel et al. 2026) to use this approach instead of the positivity restriction.

We would also like to better understand the impact of the contractiveness condition, and how it scales to more complex languages. For example, could we define a language with self-referencing records $\{z = \dots\}$ whose syntactic types are all contractive by construction? From our current understanding, this should be possible, but would not necessarily scale to records with type members à-la DOT.

Conclusion

We mechanized the soundness of three different semantics for recursive types:

1. Depth-bounded fuel with later types
2. Depth-bounded fuel with step-indexed judgements
3. Global step counting with a contractiveness condition

This allowed us to better understand the trade-offs between these approaches. We also investigated the existential gap approach, and found that it does not work.

Acknowledgements

We would like to thank Eugène Flessel for pointing to Appel’s paper, and for the helpful discussions about it.

AI use disclosure

- Claude Code and ChatGPT were used extensively for exploring the design space, answering questions, producing code, refining, and checking it. The final code is ours.
- Claude Code was used to proofread and rephrase some sentences in the report.
- Copilot suggestions were enabled while writing code, making the presentation, and writing this report.

References

- Amin, Nada, and Tiark Rompf. 2017. “Type Soundness Proofs with Definitional Interpreters.” *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA), POPL ’17, 666–79. <https://doi.org/10.1145/3009837.3009866>.
- Appel, Andrew W., and David McAllester. 2001. “An Indexed Model of Recursive Types for Foundational Proof-Carrying Code.” *ACM Transactions on Programming Languages and Systems* (New York, NY, USA) 23 (5): 657–83. <https://doi.org/10.1145/504709.504712>.
- Bovel, Matt, Viktor Kunčák, and Martin Odersky. 2026. “First-Class Refinement Types for Scala.” <https://arxiv.org/abs/2605.08369>.
- Giarrusso, Paolo G., Léo Stefanescu, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. “Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris.” *Proceedings of the ACM on Programming Languages* (New York, NY, USA) 4 (ICFP): 114:1–29. <https://doi.org/10.1145/3408996>.
- The Rocq Development Team. 2025. *The Rocq Prover*. V. 9.1.0. Zenodo, released September 15. <https://doi.org/10.5281/zenodo.17473943>.
- Timany, Amin, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. “A Logical Approach to Type Soundness.” *Journal of the ACM* (New York, NY, USA) 71 (6): 40:1–75. <https://doi.org/10.1145/3676954>.