

# SEMANTICS OF RECURSIVE TYPES



CS-642 project presentation  
Matt Bovel

May 31, 2026

# RECURSIVE TYPES

This Scala definition is recursive:

```
enum List[A]:  
  case Nil()  
  case Cons(head: A, tail: List[A])
```

Syntax of STLC + recursive types:

Terms  $t ::= x \mid \lambda x. t \mid t_1 t_2$

Types  $T ::= T_1 \rightarrow T_2 \mid \mathbf{X} \mid \mu \mathbf{X}. \mathbf{T}$

Additionally using unit, disjoint sum and pair types, we can define:

$$\text{List}[A] \triangleq \mu X. \text{Unit} + (A, X)$$

A recursive type can be unfolded:

$$\begin{aligned} \mu X. \text{Unit} + (A, X) \\ &\equiv \text{Unit} + (A, \mu X. \text{Unit} + (A, X)) \\ &\equiv \text{Unit} + (A, \text{Unit} + (A, \mu X. \text{Unit} + (A, X))) \end{aligned}$$

We want a typing rule to unfold:

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

And one to fold:

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

# SYNTAX & DEFINITIONAL INTERPRETER

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

```
Inductive Ty : Type :=
| TBool : Ty
| TFun  : Ty -> Ty -> Ty.
```

- We use de Bruijn indices for bindings
- Terms and values are separate
- Types don't have runtime semantics
- Lambdas capture their environment

```
Inductive Value : Type :=
| vbool : bool -> Value
| vabs   : (list Value) -> Term -> Value.
```

```
Fixpoint eval_wrong (env: list Value) (t: Term) : option Value :=
  match t with
  | tbool b => Some (vbool b)
  | tvar i  => nth_error env i
  | tabs _ b => Some (vabs env b)
  | tapp f a =>
    match (eval_wrong env f) with
    | Some (vabs envf b) =>
      match (eval_wrong env a) with
      | None => None
      | Some va => eval_wrong (va::envf) b
      end
    | _ => None
    end
  end.
```

# DEFINITIONAL INTERPRETER (TERMINATION)

We bound depth of recursive calls to ensure termination by adding a fuel parameter.

We return a layered option type: `None` means timeout, `Some None` means a runtime error.

```
Fixpoint eval (fuel: nat) (env: list Value) (t: Term) : option (option Value) :=
  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
    ...
```

## Type Soundness Proofs with Definitional Interpreters

Nada Amin\*    Tiark Rompf<sup>†</sup>

\*EPFL: {first.last}@epfl.ch

<sup>†</sup>Purdue University: {first}@purdue.edu



### Abstract

While type soundness proofs are taught in every graduate PL class, the gap between realistic languages and what is accessible to formal proofs is large. In the case of Scala, it has been shown that its formal model, the Dependent Object Types (DOT) calculus, cannot simultaneously support key metatheoretic properties such as environment narrowing and subtyping transitivity, which are usually required for a type soundness proof. Moreover, Scala and many other realistic languages lack a general substitution property.

The first contribution of this paper is to demonstrate how type soundness proofs for advanced, polymorphic, type systems can

Our motivation is twofold. The first is intellectual: It is commonly believed that proofs based on big-step semantics (of which definitional interpreters are a proper sub-category) are difficult or unsatisfactory. One criticism is that big-step semantics do not distinguish errors from nontermination. Thus, if taken as the basis of a type soundness proof, what is shown is only preservation, but not progress. Hence, the result is significantly weaker than a comparable small-step proof. Another commonly held belief is that advanced features such as mutable references require coinduction or other non-standard proof techniques in big-step.

With this paper, we present convincing evidence that, contrary to this view, definitional interpreters have no inherent drawbacks to

# SEMANTIC TYPES

The *value interpretation* defines what it means for a value to have a syntactic type:

```
Fixpoint interp (T: Ty) (v: Value) : 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:

```
Definition tinterp (env: list Value) (t: Term)
  (T: Value -> 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).
```

It allows us to *prove* typing rules:

```
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.
```

Proof.

...

Qed.

$$\frac{\Gamma \models f : A \rightarrow B \quad \Gamma \models a : A}{\Gamma \models f a : B}$$

## A Logical Approach to Type Soundness

AMIN TIMANY, Aarhus University, Aarhus, Denmark

ROBBERT KREBBERS, Radboud University Nijmegen, Nijmegen, Netherlands

DEREK DREYER, MPI-SWS, Saarbrücken, Germany

LARS BIRKEDAL, Aarhus University, Aarhus, Denmark

---

Type soundness, which asserts that “well-typed programs cannot go wrong,” is widely viewed as the canonical theorem one must prove to establish that a type system is doing its job. It is commonly proved using the so-called *syntactic approach* (also known as *progress and preservation*), which has had a huge impact on the study and teaching of programming language foundations. Unfortunately, syntactic type soundness is a rather weak theorem. It only applies to programs that are well typed in their entirety and thus tells us nothing about the many programs written in “safe” languages that make use of “unsafe” language features. Even worse, it tells us nothing about whether type systems achieve one of their main goals: enforcement of data abstraction. One

# INTERPRETATION OF RECURSIVE TYPES

Recursive types have a body, and an implicit de Bruijn binder. `TVar i` refers to `i`-th bound type.

```
Inductive Ty : Type :=  
  ...  
  | TVar : nat -> Ty  
  | TMu : Ty -> Ty.
```

Remember, we want  $\mu X.T \equiv T[X \mapsto \mu X.T]$ .

Tentative `interp` using substitution:

```
Fixpoint interp_wrong (T: Ty) (v: Value) : Prop :=  
  match T with  
  ...  
  | TVar i => False  
  | TMu B => interp (subst_ty 0 (TMu B) B)  
end.
```

We need *step-indexing* to break the circularity.

```
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.
```

Semantic typing quantifies over all steps:

```
Definition sem_typed (tenv: TyEnv) (t: Term)  
  (T: Ty) : Prop :=  
  forall env k,  
    Forall2 (fun T v => interp T k v) tenv env ->  
    tinterp env t (fun v => interp T k v).
```

# LATER TYPES

Remember, we want  $\mu X.T \equiv T[X \mapsto \mu X.T]$ .

But the well-founded version of `interp` gives us only  $\llbracket \mu X.T \rrbracket^k \equiv \llbracket T[X \mapsto \mu X.T] \rrbracket^{k-1}$ .

We define a *later* type  $\triangleright T$  to mean “ $T$  but one step later”, so we have:  $\mu X.T \equiv \triangleright T[X \mapsto \mu X.T]$ .

Interpretation of “later”:

```
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:

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

Can we get rid of these spooky triangles?

# DOWNWARD CLOSURE

We can avoid the later type in the folding rule if we require that semantic types are *downward closed*:

$$T^k \subseteq T^{k'} \text{ for all } k' \leq k$$

where  $S \subseteq T$  means  $\forall v, S v \implies T v$ .

This is not true with our definition, due to function types contravariance.

Downward-close `interp` by quantifying over smaller steps in the interpretation of function 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);
```

Thanks to downward closure of `interp`, we can now have  $\llbracket \mu X.T \rrbracket^k \subseteq \llbracket T[X \mapsto \mu X.T] \rrbracket^k$ , so we can get rid of the later type in the folding rule:

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

Later types in the wild:

$$\frac{\Gamma \vdash e : \triangleright T}{\Gamma \vdash \mathbf{coerce} \ e : T} \text{ (T-COERCE)}$$

$$\frac{\Gamma \mid x : \triangleright T \vdash \{\bar{d}\} : T}{\Gamma \vdash \nu x. \{\bar{d}\} : \mu x. T} \text{ (T-}\{\bar{\cdot}\}\text{-I)}$$

*Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris*  
Paolo G. Giarrusso, Léo Stefanescu, Amin Timany, Lars Birkedal, and Robbert Krebbers

# INDEXED JUDGEMENTS

Instead of using later types, can we move steps from the interpretation to the typing judgement?

$$\frac{\Gamma \vdash_k t : \mu X.T}{\Gamma \vdash_{k-1} t : T[X \mapsto \mu X.T]}$$

```
Definition sem_typed (tenv: TyEnv) (n: nat)
  (t: Term) (T: Ty) : Prop :=
  forall env k, n < k ->
    Forall2 (fun T v => interp T k v) tenv env ->
      tinterp env t (fun v => interp T (k - n) v).
```

It makes sense:

```
Lemma sem_typed_distribute: forall env tenv n e T,
  sem_typed tenv n e T ->
  (forall k,
    Forall2 (fun T v => interp T k v) tenv env) ->
  tinterp env e (fun v => forall j, interp T j v).
```

For the application rule to go through, we also need to allow a gap inside the function interpretation:

```
Inductive Ty : Type :=
  ...
  | TFun : nat -> Ty -> Ty -> Ty
  ...

interp (TFun g A B) k v :=
  exists env body, v = vabs env body
  /\ forall j (Hjlo: g < j) (Hjhi: j <= k) arg,
    interp A j arg ->
    tinterp (arg::env) body (interp B (j - g));
```

So we still have noise in the syntax,  
and now also in the typing rules:

$$\frac{\Gamma \vdash_i f : A \rightarrow_j B \quad \Gamma \vdash_k a : A}{\Gamma \vdash_{i+j+k} f a : B}$$

# EXISTENTIAL GAPS

---

Can we existentially quantify over the gaps?

No.

More in the report.



## An Indexed Model of Recursive Types for Foundational Proof-Carrying Code

Andrew W. Appel  
Princeton University

David McAllester  
AT&T Research

November 21, 2000

### Abstract

The proofs of “traditional” proof carrying code (PCC) are type-specialized in the sense that they require axioms about a specific type system. In contrast, the proofs of foundational PCC explicitly define all required types and explicitly prove all the required properties of those types assuming only a fixed foundation of mathematics such as

tion system. The semantics of the type constructors and the validity of the lemmas concerning them are proved rigorously but without mechanical verification by the designers of the PCC verification system. We will call this type-specialized PCC.

Unlike type-specialized PCC, the foundational PCC described by Appel and Felty [AF00b] avoids any commitment to a particular type system. In foundational PCC

# INSIGHT 1. GLOBAL STEPS

In the definitional interpreter, count global evaluation steps instead of depth of recursive calls (😓).

```
Equations? eval_sig (fuel: nat) (env: ValueEnv) (t: Term) :
  option ({fuel' : nat | fuel' < fuel} * option Value) by wf fuel lt :=
eval_sig 0 _ _ := None;
...
eval_sig (S fuel) env (tapp f a) with eval_sig fuel env f => {
  | None => None;
  | Some (exist _ fuel1 _, None) => Some (exist _ fuel1 _, None);
  | Some (exist _ fuel1 _, Some (vbool _)) => Some (exist _ fuel1 _, None);
  | Some (exist _ fuel1 _, Some (vabs envf body)) with eval_sig fuel1 env a => {
    | None => None;
    | Some (exist _ fuel2 _, None) => Some (exist _ fuel2 _, None);
    | Some (exist _ fuel2 _, Some va) with eval_sig fuel2 (va::envf) body => {
      | None => None;
      | Some (exist _ fuel3 _, r) => Some (exist _ fuel3 _, r)
    }
  }
}
```

## INSIGHT 2. MAKE STEPS ARITHMETIC WORK

1. Tie type steps to evaluation steps.
2. Decrease steps in the interpretation of *all* types *but* recursive types.
3. Interp recursive types as function power.

```
Definition SemTy : Type := nat -> Value -> Prop.
```

```
Definition tinterp (env: list Value) (k: nat)
  (t: Term) (T: SemTy) : Prop :=
  forall k' r, eval k env t = Some (k', r) ->
    exists v, r = Some v /\ T k' v.
```

```
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 mu_F (F: (SemTy) -> (SemTy)) : SemTy :=
  fun k v => fun_pow (S k) F Bot k v.
```

```
Fixpoint interp (tenv: list SemTy) (T: Ty)
  (k: nat) (v: Value) : 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 (interp tenv B)
  | TMu B =>
    mu_F (fun Self => interp (Self :: tenv) B) k v
  end.
```

## INSIGHT 3. CONTRACTIVE MAGIC

### Definition 11

The  $k$ -approximation of a set is the subset of its elements whose index is less than  $k$ :

$$\text{approx}(k, \tau) = \{\langle j, v \rangle \mid j < k \wedge \langle j, v \rangle \in \tau\}$$

```
Definition approx (k: nat) (F: (SemTy)): SemTy :=  
  fun j v => j < k /\ F j v.
```

### Definition 12

A well founded functional is a function  $F$  from types to types such that for any type  $\tau$  and  $k \geq 0$  we have

$$\text{approx}(k+1, F(\tau)) = \text{approx}(k+1, F(\text{approx}(k, \tau)))$$

```
Definition wf (F: SemTy -> SemTy) : Prop :=  
  forall k Z,  
    approx (S k) (F Z)  
    = approx (S k) (F (approx k Z)).
```

### Theorem 18

If  $F$  is well founded then  $\mu F = F(\mu F)$ . Hence the type inference lemmas for  $\mu$  in figure 2 hold for any well founded functional  $F$ .

```
Lemma F_well_founded: forall tenv B,  
  contractive B ->  
  wf (fun X => interp (X :: tenv) B).
```

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

Means the binder only occurs under other types.  
 $\mu X.X$  is not contractive, but  $\mu X.X \rightarrow X$  is.

## INSIGHT 3. CONTRACTIVE MAGIC (CONTINUED)

This allows us to prove the following rule:

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

Maybe we can have the contractiveness condition by construction.

```
enum List[A]:  
  case Nil()  
  case Cons(head: A, tail: List[A]) // okay  
  
type A = A // not okay
```

# CONCLUSION

---

We mechanized soundness of 3 different semantics for recursive types:

1. Depth-bounded + later types
2. Depth-bounded + indexed judgements
3. Global steps + contractiveness

This allowed us to better understand the tradeoffs between these approaches.

Approach 3. seems better than what I previously had (positivity restriction).

We also investigated the existential gap approach, and found it doesn't work.

Equations is a bit annoying to work with.