Mechanizing Proofs about Mendler-style Recursion

Rohan Jacob-Rao Andrew Cave Brigitte Pientka
McGill University
{rjacob18,acave1,bpientka}@cs.mcgill.ca

Abstract

We consider the normalization proof for a simply-typed lambda calculus with Mendler-style recursion using logical relations. This language is powerful enough to encode total recursive functions using recursive types. A key feature of our proof is the semantic interpretation of recursive types, which requires higher-kinded polymorphism in the reasoning language. We have implemented the proof in Coq due to this requirement. However, we believe this proof can serve as a challenge problem for other proof environments, especially those supporting binders, since our mechanization in Coq requires proofs of several substitution properties.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Logical relations, Recursive types, Mendler recursion, Proof assistants

1. Introduction

Recursive types are crucial in functional programming languages for defining common data types and proof languages for supporting inductive reasoning and deriving induction principles. However, even in the simplest setting of the simply typed lambda calculus with recursive types, one can easily express well-typed non-normalizing terms, i.e. terms that are not guaranteed to terminate. There are different approaches to tackle this problem. Many proof languages such as Coq (Bertot and Castéran 2004) or Agda (Norell 2007) restrict recursive types to those in which type variables only occur in positive positions. This enforces monotonicity of unrolling inductive definitions. A similar approach is adopted in proof theoretic foundations that support (co-)fix point definitions (see for example (Baelde 2012)). These foundations usually correspond to programs that use iteration instead of general recursion. Another approach is to use Mendler-style recursion (Mendler 1988). Here we enforce in the typing rule for recursion that all recursive calls must be on structurally smaller arguments, i.e. the arguments must be direct subterms.

In this paper we concern ourselves with a normalization proof for a simply-typed lambda calculus with Mendler-style recursion using logical relations and a set-based semantics. This is an interesting benchmark for mechanizing the meta-theory of formal systems for several reasons: first, we must model bindings on the level of terms and types. More importantly, our logical relation that characterizes reducible terms must be defined on open types. And finally, defining reducible terms of recursive type is elegantly achieved via a semantic interpretation using set intersections. As a consequence, mechanizing such proofs is challenging. The use of a set-based semantics also necessitates higher-kinded polymorphism in our meta-language. In fact proving normalization for the simply-typed lambda calculus with Mendler-style recursion is similar to proving normalization for System F and has similar challenges.

We have mechanized the normalization proof of the simply-typed lambda calculus with Mendler-style recursion in Coq[1]. One of the main reasons for this choice is that current proof environments that support encoding variable bindings and substitutions based on higher-order abstract syntax, such as Abella (Baelde et al. 2014) and Beluga (Pientka and Cave 2015), currently do not support higher-kinded polymorphism. Hence normalization proofs for System F or the one we consider in this paper are out of reach for such systems.

We model our term and type language using de Bruijn indices in Coq. However, we avoid term-level substitutions in our operational semantics by adopting an environment-based semantics. Instead of eagerly replacing a variable with a value, we keep the relation between variables and values explicit in the environment. This turns out to be a powerful and elegant tool not only in saving proof effort, but also in formulating the normalization theorem elegantly.

A subtle feature of our proof is the semantic interpretation of recursive types, which we are able to encode purely in second order logic. This is in contrast to to existing formulations (see for example (Abel 2010)) which rely on representing least fixed points using ordinals. In the Coq mechanization, we exploit the impredicativity of `Prop` to encode our interpretation of recursive types. Our proof involves a couple of intricate set theoretic lemmas, but generally enjoys the elegance of the logical relations proof technique. The result is a compact mechanization in Coq, just under 1400 lines of proof script.

This paper shares some lessons about effectively mechanizing such proofs. We hope that our tutorial-style description of the proof and mechanization can serve as a footprint for other similar styles of proofs in Coq or Agda. Moreover, it provides a useful benchmark for systems that support higher-order abstract syntax encodings and we hope this example makes a case for extending the power of such systems to support higher-kinded polymorphism.

2. Overview

In this section we give a brief overview of our normalization proof for a simply-typed lambda calculus with recursive types and a recursion construct. To make the recursive type constructor useful, we also include the unit type, sums and products as is conventional.

[1] Available at `http://cs.mcgill.ca/~rjacob18/rectypes.v`.

LFMTP '16 June 23, 2016, Porto, Portugal

DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

$\boxed{\Delta;\Gamma \vdash t : T}$ Term t has type T in the context Γ with type variables from Δ.

$$\frac{}{\Delta;\Gamma \vdash () : 1}\ \texttt{t-unit} \qquad \frac{x : T \in \Gamma}{\Delta;\Gamma \vdash x : T}\ \texttt{t-var} \qquad \frac{\Delta;\Gamma, x : R \vdash s : S}{\Delta;\Gamma \vdash \lambda x.\, s : R \to S}\ \texttt{t-lam} \qquad \frac{\Delta;\Gamma \vdash r : S \to T \quad \Delta;\Gamma \vdash s : S}{\Delta;\Gamma \vdash r\, s : T}\ \texttt{t-app}$$

$$\frac{\Delta;\Gamma \vdash r : R \quad \Delta;\Gamma \vdash s : S}{\Delta;\Gamma \vdash (r, s) : R \times S}\ \texttt{t-pair} \qquad \frac{\Delta;\Gamma \vdash r : T \times S}{\Delta;\Gamma \vdash \texttt{fst}\, r : T}\ \texttt{t-fst} \qquad \frac{\Delta;\Gamma \vdash r : S \times T}{\Delta;\Gamma \vdash \texttt{snd}\, r : T}\ \texttt{t-snd}$$

$$\frac{\Delta;\Gamma \vdash r : R}{\Delta;\Gamma \vdash \texttt{inl}\, r : R + S}\ \texttt{t-inl} \qquad \frac{\Delta;\Gamma \vdash s : S}{\Delta;\Gamma \vdash \texttt{inr}\, s : R + S}\ \texttt{t-inr} \qquad \frac{\Delta;\Gamma \vdash t : R + S \quad \Delta;\Gamma, x_1{:}R \vdash s_1 : T \quad \Delta;\Gamma, x_2{:}S \vdash s_2 : T}{\Delta;\Gamma \vdash \texttt{case}\, t\, \texttt{of inl}\, x_1 \Rightarrow s_1 \mid \texttt{inr}\, x_2 \Rightarrow s_2 : T}\ \texttt{t-case}$$

$$\frac{\Delta;\Gamma \vdash t : T[\mu X.\, T/X]}{\Delta;\Gamma \vdash t : \mu X.\, T}\ \texttt{t-fold} \qquad \frac{\Delta, X;\Gamma, f{:}X \to R \vdash s : T \to R}{\Delta;\Gamma \vdash \texttt{rec}\, f.\, s : \mu X.\, T \to R}\ \texttt{t-rec}$$

Figure 1. Typing Rules for Simply-Typed Lambda Calculus with Mendler-style Recursion

Our normalization proof uses the technique of logical relations, in the style of Tait (1967) and Girard et al. (1989). We give a semantic interpretation of each type in our language, denoting the set of values of each type. Our semantic interpretation of recursive types mirrors the least fixed point semantics using an intricate set intersection. To our knowledge our particular semantic interpretation and logical relation definition for recursive types is new, as it is solely expressed in second-order logic. We then show that given a term M of type A the resulting value of evaluating M is in the semantic interpretation of that type A. Our normalization statement is hence a kind of "semantic type preservation", because it shows that well-typed terms must evaluate to values of the corresponding semantic type.

As usual, the proof requires a generalization to open terms, i.e. terms that are well-typed with respect to a typing context, and a semantic interpretation of typing contexts described by a grounding well-typed value substitutions. In addition, as our types may be open as well, we also require a semantic interpretation of type variable context described by a grounding substitution for types. Our main theorem then uses these definition to say that every closed term evaluates to a value in the semantic set of its particular type. The proof is by induction on the typing derivation. The interesting cases of the proof are those involving Mendler-recursion and recursive types. As we use a set-based semantics, we rely on a couple of abstract set-theoretic lemmas in those cases of the proof. Though we do not prove it here, our language is type safe and deterministic.

3. Language

3.1 Syntax

We consider here an extension of the simply-typed lambda calculus with unit, pairs, disjoint sums and recursion. We write $()$ for unit, (t, s) for pairs, and the injections $\texttt{inl}\, t$ and $\texttt{inr}\, t$ into sums. In addition, we include $\texttt{fst}\, t$ and $\texttt{snd}\, t$ which project from a pair, and the case statement which analyzes a term of sum type. Crucially, we have an explicit recursion term $\texttt{rec}\, f.\, t$ which builds a term f which may refer to itself in its definition t. To write recursive functions we typically combine recursion with function abstraction. While we allow arbitrary recursive programs to be written in our syntax, our typing rules will guarantee that all functions are total.

$$\begin{array}{lll} \text{Types} & T, S ::= & 1 \mid T \to S \mid X \mid \mu X.\, T \mid \\ & & T \times S \mid T + S \\ \text{Terms} & t, s ::= & () \mid x \mid \lambda x.\, t \mid \texttt{rec}\, f.\, t \mid t\, s \mid \\ & & (t, s) \mid \texttt{fst}\, t \mid \texttt{snd}\, t \mid \texttt{inl}\, t \mid \texttt{inr}\, t \mid \\ & & (\texttt{case}\, t\, \texttt{of inl}\, x_1 \Rightarrow s_1 \mid \texttt{inr}\, x_2 \Rightarrow s_2) \end{array}$$

The type language includes the unit type, function types, and product and sum types as we mentioned. We also have recursive types of the form $\mu X.\, T$, where μ binds a type variable X in the body T. For example, we can define natural numbers and lists over a fixed type A as follows:

$$\begin{array}{lcl} \mathsf{Nat} & = & \mu X.\, 1 + X \\ \mathsf{List}\, A & = & \mu X.\, 1 + A \times X \end{array}$$

We take the *equirecursive* view of recursive types, with no explicit introduction form for terms of recursive types. As a consequence, we simply write for the empty list $\texttt{inl}\, ()$ and for the number 2 simply $\texttt{inr}\, (\texttt{inr}\, (\texttt{inl}\, ()))$.

3.2 Typing

We define typing of terms (Fig. 1) using two separate contexts: Γ for typing assumptions about term-level variables, and Δ to keep track of type variables that are in scope.

$$\begin{array}{lcll} \text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : T \\ \text{Type Variable Contexts} & \Delta & ::= & \cdot \mid \Delta, X \end{array}$$

The typing rules for functions, function application, pairs, projections, injections and case-expressions are straightforward. We concentrate on the typing rules for recursive types, namely `t-fold` and `t-rec`.

The `t-fold` rule implements the equirecursive view of recursive types, in which a term can type check against both a recursive type and its unfolded form. There is no explicit constructor around the term in the conclusion of the rule, as there is in the isorecursive view. As is usual in equirecursive systems, we do not have type uniqueness, since for example $\texttt{inl}\, ()$ can be interpreted both as zero and the empty list.

The Mendler-style typing rule `t-rec` specifies recursive functions on a recursive type $\mu X.\, T$, producing a result of type R. The trick to the rule is that, in checking the body of the function s, the typing assumption says that f works only on inputs of the type variable X. The only data of type X possibly appearing in s is the recursive data exactly one layer less than the argument of type $\mu X.\, T$. For example when we recurse over Nat, then the only recursive calls we can make are those on the predecessor of the input number. Hence, the recursive call can only be made on structurally smaller data, guaranteeing termination.

To illustrate, consider the program that computes the length of a list and has type $\mathsf{List} \to \mathsf{Nat}$.

$$\begin{array}{l} \texttt{rec}\, len.\, \lambda l. \\ \quad \texttt{case}\, l\, \texttt{of inl}\, x_l \Rightarrow \texttt{inl}\, () \mid \texttt{inr}\, x_r \Rightarrow \texttt{inr}\, (len\, (\texttt{snd}\, x_r)) \end{array}$$

$\boxed{t[\rho] \Downarrow v}$ Term t under environment ρ evaluates to v.

$$\frac{}{(\lambda\, x.\, t)[\rho] \Downarrow (\lambda\, x.\, t)[\rho]}\ \texttt{e-lam} \qquad \frac{}{(\texttt{rec}\, f.\, t)[\rho] \Downarrow (\texttt{rec}\, f.\, t)[\rho]}\ \texttt{e-rec} \qquad \frac{r[\rho] \Downarrow c \quad s[\rho] \Downarrow u \quad c \cdot u \Downarrow v}{(r\, s)[\rho] \Downarrow v}\ \texttt{e-app}$$

$$\frac{r[\rho] \Downarrow u \quad s[\rho] \Downarrow w}{(r, s)[\rho] \Downarrow (u, w)}\ \texttt{e-pair} \qquad \frac{t[\rho] \Downarrow (v, w)}{(\texttt{fst}\, t)[\rho] \Downarrow v}\ \texttt{e-fst} \qquad \frac{t[\rho] \Downarrow (u, v)}{(\texttt{snd}\, t)[\rho] \Downarrow v}\ \texttt{e-snd}$$

$$\frac{t[\rho] \Downarrow v}{(\texttt{inl}\, t)[\rho] \Downarrow \texttt{inl}\, v}\ \texttt{e-inl} \qquad \frac{t[\rho] \Downarrow v}{(\texttt{inr}\, t)[\rho] \Downarrow \texttt{inr}\, v}\ \texttt{e-inr} \qquad \frac{t[\rho] \Downarrow \texttt{inl}\, u \quad s_1[\rho, u/x_1] \Downarrow v}{(\texttt{case}\, t\, \texttt{of}\ \texttt{inl}\, x_1 \Rightarrow s_1 \mid \texttt{inr}\, x_2 \Rightarrow s_2)[\rho] \Downarrow v}\ \texttt{e-case-inl}$$

$$\frac{}{()[\rho] \Downarrow ()}\ \texttt{e-unit} \qquad \frac{v/x \in \rho}{x[\rho] \Downarrow v}\ \texttt{e-var} \qquad \frac{t[\rho] \Downarrow \texttt{inr}\, u \quad s_2[\rho, u/x_2] \Downarrow v}{(\texttt{case}\, t\, \texttt{of}\ \texttt{inl}\, x_1 \Rightarrow s_1 \mid \texttt{inr}\, x_2 \Rightarrow s_2)[\rho] \Downarrow v}\ \texttt{e-case-inr}$$

$\boxed{c \cdot u \Downarrow v}$ Closure c applied to value u evaluates to v.

$$\frac{t[\rho, u/x] \Downarrow v}{(\lambda\, x.\, t)[\rho] \cdot u \Downarrow v}\ \texttt{e-app-lam} \qquad \frac{t[\rho, (\texttt{rec}\, f.\, t)[\rho]/f] \Downarrow c \quad c \cdot u \Downarrow v}{(\texttt{rec}\, f.\, t)[\rho] \cdot u \Downarrow v}\ \texttt{e-app-rec}$$

Figure 2. Environment-based Big-step Semantics for Lambda Calculus with Mendler-style Recursion

Note that the `inl` and `inr` constructors appear both in analyzing the list and in constructing the new natural number that will be the length. Further, observe that $\texttt{snd}\, x_r$ refers to the tail of the list we are analyzing, as we ignore the head of the list.

Let us see how this example type checks using the typing rule `t-rec`. To see why the recursive call to len is allowed in this case, we show the typing assumptions generated along the way to type check the application of len:

$$len : X \to \mathsf{Nat},\ l : 1 + A \times X,\ x_l : 1,\ x_r : A \times X$$

Hence $\texttt{snd}\, x_r$ has type X, which means we are allowed to pass it to the function len to compute the length of the tail of the list. Note that this is actually the only way we could have used len: the only data of type X is the list exactly one constructor smaller. This ensures that recursive calls are made only on structurally smaller data, and hence that recursive functions must terminate.

We briefly explain here why the usual construction of a nonterminating program using negative recursive types is not possible in our language. In languages with unrestricted recursive types and a definable "unfold" function, one can define a nonterminating program even without recursion. The simplest way to do this is by indirectly constructing a program of the type $T \equiv \mu X.\, X \to X$. The first step of the construction is to define an unfold function of type $T \to (T \to T)$. One might try this using a term of the form $\texttt{rec}\, f.\, \lambda\, x.\, t$ for some t, such as $\texttt{rec}\, f.\, \lambda\, x.\, f\, x$. However, as we begin to type check this term using the `t-rec` and `t-lam` rules, we see that we cannot use either f or x in t.

$$\frac{\dfrac{X; f : X \to (T \to T), x : X \to X \vdash t : T \to T}{X; f : X \to (T \to T) \vdash \lambda\, x.\, t : (X \to X) \to (T \to T)}\ \texttt{t-lam}}{\cdot; \cdot \vdash \texttt{rec}\, f.\, \lambda\, x.\, t : T \to (T \to T)}\ \texttt{t-rec}$$

The problem is that both f and x take inputs of type X, which is a freshly created type variable. This gives an idea why we are unable to produce nonterminating terms, even with negative recursive types.

We remark that this form of Mendler recursion is very restrictive for a couple of reasons. Firstly, we cannot directly encode recursive calls on data more than one layer smaller. Second, we cannot use structurally smaller data, such as the tail of the list, for anything other than a recursive call! For more expressive forms of Mendler-style recursion, we refer the reader to for example Abel (2010). However, it is worth noting that even for those extended systems the overall structure and challenges in the normalization proof remain.

3.3 Evaluation

We describe evaluation of our language using a big-step operational semantics with value environments that keep track of variable – value bindings. To understand the evaluation judgments, we need to define values and environments.

Closures	c	$::=$	$(\lambda\, x.\, t)[\rho] \mid (\texttt{rec}\, f.\, t)[\rho]$
Values	v	$::=$	$() \mid (v, u) \mid \texttt{inl}\, v \mid \texttt{inr}\, v \mid c$
Environments	ρ	$::=$	$\cdot \mid \rho, v/x$

Each value corresponds to an introduction form of the language. This is easy to see for the unit value, pairs and injections into sums. The other two values are function *closures*, which are either lambda terms or recursive terms that are closed under an environment ρ. An environment is simply a list of values representing a simultaneous substitution of variables with values. Hence a function term f can be *closed* under an environment ρ if all of its free variables are accounted for in ρ, and this pair $f[\rho]$ is called a closure.

We describe the evaluation rules in Fig. 2. Note that the notation $t[\rho]$ used in defining the evaluation judgment for terms is overloaded – general closures of terms paired with their environments are not part of the syntax of language. Rather the evaluation judgment relates a term t in the environment ρ to the value v. The evaluation rules are mostly straightforward. In the rule for `e-lam` and `e-rec` we return an actual closure. To evaluate $r\, s$, we first evaluate r to a closure, i.e. either $(\lambda\, x.\, t)[\rho]$ or $(\texttt{rec}\, f.\, t)[\rho]$, and s to a value w. We then continue to extending the environment ρ. When we encounter $(\lambda\, x.\, t)[\rho]$ we simply extend the environment ρ with the value w. When we encounter $(\texttt{rec}\, f.\, t)[\rho]$, we first continue to unroll the recursive function and evaluate $t[\rho, (\texttt{rec}\, f.\, t)[\rho]/f]$ to a closure before extending the runtime environment with w. This models the usual functional interpretation of a fixed point or recursion operator. Our typing rules guarantee that recursive calls are only made on smaller arguments. The remaining rules are mostly straightforward.

The key benefit of our use of environments is that it avoids the notion of substitution on terms. This is very convenient during mechanization, as we do not have to implement capture-avoiding substitution nor prove properties about term-level substitutions. En-

vironments also help us formulate our semantics and our statement of the normalization theorem, as we will see in the next sections.

4. Semantics

The main idea of the proof, following the method of logical relations, is to construct a semantic model of our syntactic language. In this case, we define a set-theoretic interpretation of each type, which we use in the statement of the normalization theorem. As the proof is by induction on the typing derivation, this statement of the theorem will give more powerful induction hypotheses to use in each case of the proof.

Let Val be the set of all values allowed by our syntax. Before we can give the set interpretation of types, we need to define a semantic interpretation of type variable contexts. Assume a set $TpCtx$ containing all valid type variable contexts constructed from our syntax, and a countably infinite set $TpVar$ of type variables. Then a *type variable mapping* is a function $\eta : TpVar \to \mathcal{P}(Val)$, mapping each type variable to a subset of values (possibly the empty set). The function $\mathcal{D}$ gives the set of type variable mappings associated with a given type variable context.

$$\begin{array}{lcl} \mathcal{D} : TpCtx \to \mathcal{P}(TpVar \to \mathcal{P}(Val)) \\ \mathcal{D}[\cdot] & = & \{\_ \mapsto \emptyset\} \\ \mathcal{D}[\Delta, X] & = & \{\eta[X \leftarrow \mathcal{C}] \mid \eta \in \mathcal{D}[\Delta], \mathcal{C} \in \mathcal{P}(Val)\}. \end{array}$$

The definition says that a type variable mapping conforms to a type variable context as long as it does not refer to type variables outside that context. Though simple, it is necessary to describe how to interpret type variables occurring in types.

The interpretation of types $\mathcal{V}$ is more interesting, describing the set of values of a type under a type variable mapping. For that we define some "semantic types", which give set-theoretic analogues of our syntactic type constructors. For $\mathcal{A}, \mathcal{B} \in \mathcal{P}(Val)$, define the following sets.

$$\begin{array}{l} \mathcal{A} \times \mathcal{B} = \{(v, w) \mid v \in \mathcal{A}, w \in \mathcal{B}\} \\ \mathcal{A} + \mathcal{B} = \{\mathtt{inl}\, v \mid v \in \mathcal{A}\} \cup \{\mathtt{inr}\, w \mid w \in \mathcal{B}\} \\ \mathcal{A} \to \mathcal{B} = \{v \in Val \mid \forall u \in \mathcal{A}.\ \exists w \in \mathcal{B}.\ v \cdot u \Downarrow w\} \end{array}$$

The first two sets are set analogues of the product and sum type constructors respectively, and we overload the connectives to reflect this. The latter gives a semantic notion of function types, each containing values which, when applied to any value in the first set, evaluate to some value in the second set.

With this notation we define the semantic interpretation $\mathcal{V}$, taking a type in Tp and a type variable mapping η to give a subset of Val.

$$\begin{array}{lcl} \mathcal{V} : Tp \to (TpVar \to \mathcal{P}(Val)) \to \mathcal{P}(Val) \\ \mathcal{V}[1](\eta) & = & \{()\} \\ \mathcal{V}[T \times S](\eta) & = & \mathcal{V}[T](\eta) \times \mathcal{V}[S](\eta) \\ \mathcal{V}[T + S](\eta) & = & \mathcal{V}[T](\eta) + \mathcal{V}[S](\eta) \\ \mathcal{V}[T \to S](\eta) & = & \mathcal{V}[T](\eta) \to \mathcal{V}[S](\eta) \\ \mathcal{V}[X](\eta) & = & \eta(X) \\ \mathcal{V}[\mu X.\, F](\eta) & = & \boldsymbol{\mu}(\mathcal{X} \mapsto \mathcal{V}[F](\eta[X \leftarrow \mathcal{X}])) \end{array}$$

where for any $\mathcal{F} : \mathcal{P}(Val) \to \mathcal{P}(Val)$,

$$\boldsymbol{\mu}(\mathcal{F}) = \bigcap_{\forall \mathcal{X}.\ \mathcal{X} \subseteq \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{C}} \mathcal{C}.$$

We first explain the simple cases of the interpretation. The unit case is clear, since () is the only possible value of type 1. The product, sum and function type cases utilize our semantic types over the interpretations of the component types. The case for a type variable simply invokes the type variable mapping that is passed to the function, looking up the set of values referenced by that type variable. The difficult case is that of a recursive type.

The $\boldsymbol{\mu}$ operator is so named to evoke the notion of a semantic fixed point operator, mirroring the syntactic μ construct. It transforms a semantic function $\mathcal{F}$ on sets of values to its least fixed point. This is achieved via the set intersection, which gets the smallest set satisfying the closure property (appearing below the intersection symbol). Technically the closure property on $\mathcal{C}$ is not a fixed point property, but rather a "pre-fixed point" property, stating that subsets $\mathcal{X}$ of $\mathcal{C}$ remain subsets under $\mathcal{F}$. This gives us the least fixed point operator that we want. In the interpretation of a recursive type, we apply $\boldsymbol{\mu}$ to the semantic function that maps a set of values to the semantic interpretation of the body of the recursive type under an extended type variable mapping. This definition is inspired by encodings of recursive types into variants of System F^{ω} (Abel et al. 2005), and is key to our proof.

Note that other semantic interpretations of recursive types, such as in Abel (2010), use a formulation based on ordinals. This representation is something of the form

$$\boldsymbol{\mu}\mathcal{F} = \bigcup_{i \in \omega_1} \mathcal{F}^i(\phi),$$

where ω_1 is the first uncountable ordinal. We prefer our formulation because it can be encoded directly in second-order logic. This is particularly advantageous in the mechanization, as we do not rely on a large proof library of ordinal properties.

The final piece of machinery we need is a semantic interpretation $\mathcal{G}$ of typing contexts, which describes the value substitutions, i.e. environments, that are well-typed under a given context. Assume a set Ctx of well-formed typing contexts and a set Env of environments. Then $\mathcal{G}$ takes a context from Ctx and a type variable mapping to give the environments in Env that match the context. That is, if $\rho \in \mathcal{G}[\Gamma](\eta)$, then every value in ρ is in the semantic interpretation of the corresponding type in Γ. This is expressed in the following definition.

$$\begin{array}{lcl} \mathcal{G} : Ctx \to (TpVar \to \mathcal{P}(Val)) \to \mathcal{P}(Env) \\ \mathcal{G}[\cdot](\eta) & = & \{\cdot\} \\ \mathcal{G}[\Gamma, x : T](\eta) & = & \{\rho, v/x \mid \rho \in \mathcal{G}[\Gamma](\eta), v \in \mathcal{V}[T](\eta)\}. \end{array}$$

With these formal semantics we are ready to state the normalization theorem and describe the proof.

5. Normalization

5.1 Theorem Statement

The main theorem is stated using the semantic machinery we built in the previous section. It is required to give us a sufficiently strong induction hypothesis. The assumptions in the theorem are that we have a well-typed term t under contexts Δ and Γ, and we have a type variable mapping and environment that conform to Δ and Γ respectively. The conclusion is that t evaluates to some value v in the semantic interpretation of the type.

Theorem. *If* $\Delta; \Gamma \vdash t : T$, $\eta \in \mathcal{D}[\Delta]$ *and* $\rho \in \mathcal{G}[\Gamma](\eta)$ *then* $t[\rho] \Downarrow v$ *for some* $v \in \mathcal{V}[T](\eta)$.

A simple statement of normalization for closed terms can be recovered by considering the case in which Δ and Γ are empty. The empty type variable mapping and environment satisfy the latter assumptions, giving the following corollary. We assume a natural syntax for our judgments with empty contexts and environment.

Corollary. *If* $t : T$ *then* $t \Downarrow v$ *for some value* v.

The proof of the main theorem is by induction on the typing derivation $\mathcal{D}$. This gives a case for each typing rule, which we explain in the remainder of this section.

5.2 Simple Cases

To give a sense of the proof structure, we sketch some of the simple cases here. As a warm-up, suppose the root of the typing derivation is `t-unit`. In this case $t = ()$ and $T = 1$. By `e-unit`, $()[\rho] \Downarrow ()$ and we observe that $()$ is the sole member of $V[1](\eta)$. This completes the unit case.

For the variable case, assume the root of the typing derivation is `t-var`, so $t = x$ and $x{:}T \in \Gamma$. We need to show that $\exists v \in V[T](\eta)$ such that $v/x \in \rho$. This is proved by induction on the structure of Γ, though we omit the proof here. Using this fact and `e-var`, we obtain that $x[\rho] \Downarrow v \in V[T](\eta)$, completing the variable case.

The last simple case we show is the lambda abstraction case. Suppose the root of the typing derivation is `t-lam`, so $t = \lambda\, x.\, s$ and $T = R \to S$. We claim that $v = (\lambda\, x.\, s)[\rho] \in \mathcal{V}[R \to S](\eta)$, since we know that $(\lambda\, x.\, s)[\rho] \Downarrow (\lambda\, x.\, s)[\rho]$ by `e-lam`. Unfolding the definition of $\mathcal{V}[R \to S](\eta)$, it remains to show that $\forall u \in \mathcal{V}[R](\eta), v \cdot u \Downarrow w$ for some $w \in \mathcal{V}[S](\eta)$. Assume $u \in \mathcal{V}[R](\eta)$. Then $\rho, u/x \in \mathcal{G}[\Gamma, x{:}R](\eta)$, and $\Delta; \Gamma, x{:}R \vdash s : S$ by `t-lam`. Using the induction hypothesis, we learn that $\exists w \in V[S](\eta)$ such that $s[\rho, u/x] \Downarrow w$. Finally by `e-app-lam` we get that $(\lambda\, x.\, s)[\rho] \cdot u \Downarrow w$, which is what we needed.

We omit the cases for `t-app`, `t-pair`, `t-fst`, `t-snd`, `t-inl`, `t-inr` and `t-case`. They follow the structure we have shown above and do not provide any novelty. The cases of interest in this paper are the `t-fold` and `t-rec` cases, which involve recursive types.

5.3 Fold Case

To prove the `t-fold` case, we will first need a lemma regarding type substitutions. This is because recursive types are unrolled using type substitution. The lemma says that the semantic interpretation of a type under a substitution is equal to the type interpretation under the appropriately extended type variable mapping.

Lemma (Semantics of type substitution)**.** *For types T and S and a type variable mapping η,*

$$\mathcal{V}[T[S/X]](\eta) = \mathcal{V}[T](\eta[X \leftarrow \mathcal{V}[S](\eta)]).$$

The proof of this lemma is by induction on the structure of T, using the definition of type substitutions. Though not complicated, we will see later that this proof is not so straight-forward in the mechanization. There, we will require a whole framework of substitution mechanisms and a suite of more general lemmas. We shall omit the proof here and revisit it when we describe the mechanization.

The other lemma we need for this case is a semantic one, showing that the semantic fixed point operator $\boldsymbol{\mu}$ is really a (pre-)fixed point.

Lemma (Pre-fixed point)**.** *Let $\mathcal{F} : \mathcal{P}(Val) \to \mathcal{P}(Val)$ and $\boldsymbol{\mu}$ as defined in our semantics. Then $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \subseteq \boldsymbol{\mu}\mathcal{F}$.*

This is not strictly a fixed point property because the result is a set inclusion instead of an equality. Nevertheless the lemma will be sufficient for proving this case of the theorem.

Proof. As the set on the right-hand side is an intersection, we can show inclusion by showing inclusion in each set in the intersection. So let $\mathcal{C}$ be an arbitrary set in the intersection, that is a subset of Val satisfying the property that $\forall \mathcal{X}.\ \mathcal{X} \subseteq \mathcal{C} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{C}$. We need to show that $\mathcal{F}(\boldsymbol{\mu}\mathcal{F}) \subseteq \mathcal{C}$. From the assumed property of $\mathcal{C}$, it suffices to show that $\boldsymbol{\mu}\mathcal{F} \subseteq \mathcal{C}$. We now have a set inclusion with an intersection on the left. To prove it, we need just one set in the intersection that is a subset of $\mathcal{C}$. That is, we need a $\mathcal{D} \subseteq \mathcal{C}$ such that $\forall \mathcal{X}.\ \mathcal{X} \subseteq \mathcal{D} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{D}$. However, $\mathcal{C}$ itself satisfies this property from our original assumption. □

To prove the `t-fold` case, we start by applying the induction hypothesis to the typing subderivation. This says that there is a $v \in \mathcal{V}[F[\mu X.\, F/X]](\eta)$ such that $t[\rho] \Downarrow v$. Define $\mathcal{F} : \mathcal{P}(Val) \to \mathcal{P}(Val)$ as $\mathcal{F}(\mathcal{X}) = V[F](\eta[X \leftarrow \mathcal{X}])$, so that $V[\mu X.\, F](\eta) = \boldsymbol{\mu}\mathcal{F}$ by definition. To finish the case we need to only show that $v \in V[\mu X.\, F](\eta) = \boldsymbol{\mu}\mathcal{F}$. By the type substitution lemma, we have $\mathcal{V}[F[\mu X.\, F/X]](\eta) = \mathcal{V}[F](\eta[X \leftarrow \mathcal{V}[\mu X.\, F](\eta)])$. Rewriting the right-hand set further using $\mathcal{F}$ gives $\mathcal{V}[F](\eta[X \leftarrow \boldsymbol{\mu}\mathcal{F}])$, and using the definition once more yields $\mathcal{F}(\boldsymbol{\mu}\mathcal{F})$. However, by the previous lemma we know this is a subset of $\boldsymbol{\mu}\mathcal{F}$. Hence $v \in \boldsymbol{\mu}\mathcal{F}$, completing the proof of this case.

5.4 Recursion Case

The only remaining case is the `t-rec` case. To prove it we will need another semantic result regarding the semantic fixed point $\boldsymbol{\mu}$. It gives a sufficient condition for membership in the semantic function space from a fixed point.

Lemma (Function space from a fixed point)**.** *Suppose $v \in Val$, $\mathcal{C} \subseteq Val$ and $\mathcal{F} : \mathcal{P}(Val) \to \mathcal{P}(Val)$.*

If $\forall \mathcal{X} \subseteq Val.\ v \in \mathcal{X} \to \mathcal{C} \implies v \in \mathcal{F}(\mathcal{X}) \to \mathcal{C}$, then $v \in \boldsymbol{\mu}\mathcal{F} \to \mathcal{C}$.

Proof. Let us define some useful notation. For $v \in Val$ and $\mathcal{B} \subseteq Val$, let $\mathcal{E}_v(\mathcal{B}) = \{u \in Val \mid \exists w \in \mathcal{B}\ v \cdot u \Downarrow w\}$. Note that this definition is closely related to that of a semantic function space, as for $\mathcal{A} \subseteq Val$, $v \in \mathcal{A} \to \mathcal{B} \iff \mathcal{A} \subseteq \mathcal{E}_v(\mathcal{B})$. Restating the lemma using our $\mathcal{E}$ notation, it says that if $\forall \mathcal{X} \subseteq Val.\ \mathcal{X} \subseteq \mathcal{E}_v(\mathcal{C}) \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{E}_v(\mathcal{C})$ then $\boldsymbol{\mu}\mathcal{F} \subseteq \mathcal{E}_v(\mathcal{C})$.

Assume the premise is true. Unfolding the definition of $\boldsymbol{\mu}\mathcal{F}$, we need to prove that

$$\bigcap_{\forall \mathcal{X}.\ \mathcal{X} \subseteq \mathcal{W} \implies \mathcal{F}(\mathcal{X}) \subseteq \mathcal{W}} \mathcal{W} \subseteq \mathcal{E}_v(\mathcal{C}).$$

As in the last lemma, it suffices to show that just one $\mathcal{W}$ in the intersection is included in $\mathcal{E}_v(\mathcal{C})$. However, $\mathcal{E}_v(\mathcal{C})$ itself is in the intersection because of the assumption. Moreover it is a subset of itself so we are done. □

The final lemma we need is a form of backward closure for recursive terms. It says that if the unrolling of a recursive term evaluates to a function value, then the recursive term itself is a function value.

Lemma (Backward closure)**.** *Let t be a term, ρ an environment and $\mathcal{A}, \mathcal{B} \subseteq Val$.*

If $t[\rho, (\mathtt{rec}\, f.\, t)[\rho]/f] \Downarrow v$ for some $v \in \mathcal{A} \to \mathcal{B}$, then $(\mathtt{rec}\, f.\, t)[\rho] \in \mathcal{A} \to \mathcal{B}$.

Proof. Assume the premise. By the definition of the semantic function space, we have $\forall u \in \mathcal{A}.\ \exists w \in \mathcal{B}.\ v \cdot u \Downarrow w$. Then by `e-app-rec` we gain directly that $(\mathtt{rec}\, f.\, t)[\rho] \cdot u \Downarrow w$, showing that $(\mathtt{rec}\, f.\, t)[\rho] \in \mathcal{A} \to \mathcal{B}$. □

We can finally address the `t-rec` case. By `e-rec`, we know $(\mathtt{rec}\, f.\, s)[\rho] \Downarrow (\mathtt{rec}\, f.\, s)[\rho]$, so we only need to show that

$$(\mathtt{rec}\, f.\, s)[\rho] \in \mathcal{V}[\mu X.\, F \to R](\eta).$$

Simplifying the right-hand set using the semantics for function types and recursive types, this is equivalent to showing

$$(\mathtt{rec}\, f.\, s)[\rho] \in \boldsymbol{\mu}(\mathcal{X} \mapsto \mathcal{V}[F](\eta[X \leftarrow \mathcal{X}])) \to \mathcal{V}[R](\eta).$$

By our semantic lemma for this case, it suffices to show that, for all $\mathcal{X} \subseteq Val$,

$$(\texttt{rec}\, f.\, s)[\rho] \in \mathcal{X} \to V[R](\eta) \text{ implies}$$
$$(\texttt{rec}\, f.\, s)[\rho] \in \mathcal{V}[F](\eta[X \leftarrow \mathcal{X}]) \to \mathcal{V}[R](\eta).$$

So suppose $\mathcal{X} \subseteq Val$ such that $(\texttt{rec}\, f.\, s)[\rho] \in \mathcal{X} \to \mathcal{V}[R](\eta)$. By backward closure, it suffices to show that

$$\exists v \in \mathcal{V}[F](\eta[X \leftarrow \mathcal{X}]) \to \mathcal{V}[R](\eta)$$
$$\text{such that } s[\rho, (\texttt{rec}\, f.\, s)[\rho]/f] \Downarrow v.$$

Now, the induction hypothesis says that

$$\forall \eta' \in \mathcal{D}[\Delta, X].\ \forall \rho' \in \mathcal{G}[\Gamma, f : X \to R](\eta').$$
$$\exists v \in \mathcal{V}[F \to R](\eta').\ s[\rho'] \Downarrow v.$$

We have $\eta[X \leftarrow \mathcal{X}] \in \mathcal{D}[\Delta, X]$ since $\eta \in \mathcal{D}[\Delta]$ and $\mathcal{X} \subseteq Val$. Also, $\rho, (\texttt{rec}\, f.\, s)[\rho]/f \in \mathcal{G}[\Gamma, f : X \to R](\eta')$ because $\rho \in \mathcal{G}[\Gamma](\eta)$ implies $\rho \in \mathcal{G}[\Gamma](\eta')$ and

$$(\texttt{rec}\, f.\, s)[\rho] \in \mathcal{V}[X \to R](\eta') = \mathcal{X} \to \mathcal{V}[R](\eta).$$

Thus instantiating $\eta' = \eta[X \leftarrow \mathcal{X}]$ and $\rho' = \rho, (\texttt{rec}\, f.\, s)[\rho]/f$ in the induction hypothesis means there is a

$$v \in \mathcal{V}[F \to R](\eta') = \mathcal{V}[F](\eta[X \leftarrow \mathcal{X}]) \to \mathcal{V}[R](\eta)$$

such that $s[\rho'] = s[\rho, (\texttt{rec}\, f.\, s)[\rho]/f] \Downarrow v$. But this is exactly what we were required to show! Hence we have completed the `t-rec` case and the entire proof.

6. Mechanization

6.1 Overview

This section describes the implementation of our proof in the proof assistant Coq. Most of the proof translates smoothly into the appropriate encodings in Coq. However, a substantial part of the proof which is left implicit in the paper version is the treatment of substitutions on types and the semantic properties that hold of them. In the mechanization, we must deal with this explicitly so we use a de Bruijn representation of type variables and a subtle framework of parallel substitutions and renamings. In the following subsections we describe how we encode the syntax of our language, our substitution framework, implementation of some set-theoretic notions, lemmas about the semantics, and finally the main proof.

6.2 Representation of Syntax

We represent the syntax of our language using inductive data types in Coq. Typing and evaluation are also represented using inductive types, which allow us to reason about them inductively. In particular, we are able to perform induction on typing derivations, which is the main proof strategy for the normalization theorem.

There is a subtle aspect to encoding syntax in a proof language, which is how to represent variables and binding structures. Our solution here is to use de Bruijn indices for both term- and type-level variables. For term variables, this representation is convenient because the only operations we perform on them are pushing onto a typing context or environment, and looking them up in those lists by index. We never need to implement substitutions on terms due to our environment-based evaluation. Hence the de Bruijn representation for term-level variables is ideal for our purposes.

However, the situation for type variables is quite different because we use syntactic substitutions during type checking. In particular, we perform a type substitution when unfolding recursive types in the `t-fold` rule, and rename type variables when we inspect the body of a recursive type in the `t-rec` rule. Hence we need an explicit definition of type substitutions and a lemma about the semantics of types under such substitutions. The next subsection describes the substitution framework we need for this.

6.3 Substitution Framework

We use the idea of parallel, or simultaneous, substitutions on types. This means that substitutions can replace not just a single type variable, but several at a time. This approach gives a general way to compose and reason about substitutions.

One issue we must deal with is showing that our definitions of type substitution and composition of substitutions are well-founded, i.e. terminating. These mutual definitions are not structurally recursive and using a naive approach to parallel substitutions makes it non-trivial to prove them terminating in Coq. We favour a treatment in which well-foundedness is checked automatically by Coq's termination checking.

To do this we introduce the auxiliary notion of a *renaming*, which is well described in (Benton et al. 2012). A renaming is very similar to a substitution except that it only replaces variables with other variables, as opposed to replacing them with arbitrary types. With this notion we then define the composition of substitutions with renamings instead of with other substitutions. This relies on definitions of types under renamings as well as composition of renamings. Through this extra layer of indirection, our definitions will be evidently total according to Coq's termination checker. The cost of this approach, as in the previously referenced work, is a bloated set of lemmas to prove about the interaction between these definitions and our semantics.

Before we define this framework, let us restate the definition of the type language using the de Bruijn representation of type variables.

$$\text{Types} \quad T, S ::= 1 \mid T \to S \mid \mathrm{Var}\, k \mid \mu T \mid T \times S \mid T + S$$

Note that variables are represented by natural numbers, and that recursive types no longer bind a variable by name, but instead introduce a variable of index 0, while "shifting" the other variables up by one. To implement this in type checking, we require the notion of substitutions and renamings as mentioned.

Substitutions and renamings consist either of a shift, which adds to each of the indices representing free variables, or an extension with a type (variable). We overload the notation for a shift to use it in both syntactic categories.

$$\begin{array}{lllll} \text{Substitutions} & \sigma & ::= & \uparrow^n \mid \sigma, T \\ \text{Renamings} & \pi & ::= & \uparrow^n \mid \pi, k \end{array}$$

Observe that the only difference between the two constructs is that substitutions are extended with types whereas renamings are extended with natural numbers representing type variables.

In order to define type substitution we first need some more primitive concepts, namely: composition of renamings, types under renamings, and substitutions composed with renamings. In fact, each of these operations is required to define the next one, resulting in a chain of dependencies.

Building from the bottom up, the first operation we need is that of composing renamings. The syntax $\pi_1[\pi_2]$ refers to the renaming obtained by composing the two renamings π_1 and π_2. The definition is by cases on the first renaming.

$$\uparrow^n [\pi] = \mathrm{pop}\, n\, \pi$$
$$(\pi', k)[\pi] = \pi'[\pi], \pi(k)$$

We rely on an auxiliary "pop" function that drops indices from a renaming, or adds to the shift if the renaming is a shift. The other operation used is that of a lookup $\pi(k)$, which simply indexes into the list. We will assume identical pop and lookup functions for substitutions later.

Now we can define types under renamings, denoted $T[\pi]$, this time by cases on T.

$$
\begin{aligned}
1[\pi] &= 1 \\
(T \times S)[\pi] &= T[\pi] \times S[\pi] \\
(T + S)[\pi] &= T[\pi] + S[\pi] \\
(T \to S)[\pi] &= T[\pi] \to S[\pi] \\
(\mathrm{Var}\, k)[\pi] &= \mathrm{Var}(\pi(k)) \\
(\mu T)[\pi] &= \mu(T[\pi[\uparrow^1], 0])
\end{aligned}
$$

Most of the cases distribute the renaming π according to the structure of the type. In the variable case, we look up the renaming to get the correct de Bruijn index. In the last case, we carry π under the μ binder by introducing a new variable at index 0 and shifting the variables in π up by one.

With this definition we can define substitutions composed with renamings, in the form $\sigma[\pi]$.

$$
\begin{aligned}
\uparrow^n [\pi] &= \langle \mathrm{pop}\, n\, \pi \rangle \\
(\sigma', T)[\pi] &= \sigma'[\pi], T[\pi]
\end{aligned}
$$

Here we have denoted with angle brackets the conversion from a renaming to a substitution, which is straight-forward (wrapping each index with the variable constructor Var). In the second case of an extended substitution, we rely on our previous operation of a type under a renaming.

At the top of our chain of definitions, we finally have type substitution.

$$
\begin{aligned}
1[\sigma] &= 1 \\
(T \times S)[\sigma] &= T[\sigma] \times S[\sigma] \\
(T + S)[\sigma] &= T[\sigma] + S[\sigma] \\
(T \to S)[\sigma] &= T[\sigma] \to S[\sigma] \\
(\mathrm{Var}\, k)[\sigma] &= \sigma(k) \\
(\mu T)[\sigma] &= \mu(T[\sigma[\uparrow^1], \mathrm{Var}\, 0])
\end{aligned}
$$

This is extremely similar to the definition of types under renamings. Again the first four cases are structural on the type, and the variable case simply looks up the appropriate entry of the substitution. The final case carries σ under the binder by shifting the free variables and adding a new type variable at index 0. This gives us all the notions we need to prove semantic properties of types under substitutions.

6.4 Modelling Sets and Semantics

In this subsection we take a brief detour to explain how we model sets and some of our semantic structures in Coq. In particular, this is enlightening as to why we rely on Coq's support for higher-kinded polymorphism.

Since our semantic interpretation yields mathematical sets in our theory, we need a way to encode sets in Coq. We do so using functions from a type `S` into `Prop`, the *kind* of propositions. In Coq we define a type synonym as follows.

```
Definition set (S : Type) : Type := S -> Prop.
```

For example, a subset of values (a member of $\mathcal{P}(Val)$) is represented by a function of type `value -> Prop`, indicating whether a given value is in the subset or not. This representation works well for implementing concepts such as set intersection, semantic function spaces, and even our semantic fixed point operator. We use these set operations to define our semantic interpretation of types $\mathcal{V}[T](\eta)$.

Note that the definition of our fixed point operator $\boldsymbol{\mu}$ relies on Coq's support for higher-kinded polymorphism. One can see this from its Coq definition given below.

```
Definition prefp (F : set value -> set value)
  : set value :=
  fun v => forall C : set value,
    (forall X, subset X C -> subset (F X) C)
    -> C v.
```

The key is that we quantify over *type constructors* `C` and `X` of type `value -> Prop`. This requires not only ordinary polymorphism which allows quantification over types, but *higher-order* or *higher-kinded* polymorphism. This is the main reason we chose to use Coq over other reasoning languages.

6.5 Semantics of Substitutions

To use the substitution framework we described, we need lemmas that allow us to reason about them. For that we need semantic interpretations of both substitutions and renamings. These interpretations will in fact produce type variable mappings.

Type variable mappings are modelled as lists of sets of values. They are implemented using the following syntax, where χ represents a set of values.

$$\text{Type Variable Mappings} \quad \eta \quad ::= \quad \cdot \mid \eta, \chi$$

This representation is sufficiently expressive because it takes advantage of our de Bruijn representation of type variables. Type variable mappings refer to type variables implicitly by index in the list.

Now let us look at the semantic interpretation of renamings.

$$
\begin{aligned}
\mathcal{V}^{**}[\uparrow^n](\eta) &= \mathrm{pop}\, n\, \eta \\
\mathcal{V}^{**}[\pi', k](\eta) &= \mathcal{V}^{**}[\pi'](\eta), \eta(k)
\end{aligned}
$$

Here we refer to a "pop" function which removes elements from a type variable mapping, and a lookup function $\eta(k)$ which retrieves the set at a given index. These are similar to the operations we used earlier for renamings and substitutions. They are all defined in our mechanization and we proved some composition properties about them.

Next we see the interpretation of substitutions.

$$
\begin{aligned}
\mathcal{V}^{*}[\uparrow^n](\eta) &= \mathrm{pop}\, n\, \eta \\
\mathcal{V}^{*}[\sigma', T](\eta) &= \mathcal{V}^{*}[\sigma'](\eta), \mathcal{V}[T](\eta)
\end{aligned}
$$

The idea of both of these definitions is to capture the notion of a shift and to bootstrap the semantic interpretation of types.

The $\mathcal{V}^*$ and $\mathcal{V}^{**}$ notation here is intentionally similar to the $\mathcal{V}$ notation of semantic types. These semantic notions are all intended to compose in a regular fashion. This is made precise in the following set of lemmas, which correspond to the four operations we defined in Section 6.3.

$$\mathcal{V}^{**}[\pi_1[\pi_2]](\eta) = \mathcal{V}^{**}[\pi_1](\mathcal{V}^{**}[\pi_2](\eta)) \tag{1}$$

$$\mathcal{V}[T[\pi]](\eta) = \mathcal{V}[T](\mathcal{V}^{**}[\pi](\eta)) \tag{2}$$

$$\mathcal{V}^{*}[\sigma[\pi]](\eta) = \mathcal{V}^{*}[\sigma](\mathcal{V}^{**}[\pi](\eta)) \tag{3}$$

$$\mathcal{V}[T[\sigma]](\eta) = \mathcal{V}[T](\mathcal{V}^{*}[\sigma](\eta)) \tag{4}$$

We have proved all of these properties in our mechanization. As with the definitions involving renamings and substitutions, each one depends on the one before it.

6.6 Main Lemmas and Theorem

With our syntax modelled and the framework for substitutions set up, we are able to address the main normalization theorem. As in the paper version, the proof is by induction on the typing derivation, which in Coq is encoded as a term of the inductive

type that represents the typing relation. Each case requires using the induction hypotheses and the appropriate evaluation rules.

As in the paper version, it is the `t-fold` and `t-rec` cases that are interesting. They use our abstract lemmas about the semantic fixed point operator $\boldsymbol{\mu}$, the proofs of which are smoothly translated using our encoding of sets from Section 6.4. In addition the `t-fold` case uses the type substitution lemma (4) specialized to a single substitution, as that is how recursive types are unfolded in `t-fold` rule. Finally, hidden in the paper proof, the `t-rec` case uses the type renaming lemma (2), as there are implicit shifts in the premise of the `t-rec` rule which we model using renamings.

Another challenge of the proof we should mention is caused by Coq's notion of equality. Because Coq's type theory is intensional, there are times when we cannot directly rewrite terms using an extensional equality. A common instance of this issue is when we wish to rewrite a term involving a set to one involving a set with exactly the same elements. Since sets are modelled as functions into `Prop`, this is not directly possible because the sets are extensionally equal but not definitionally equal. Our solution to this problem is to define a separate notion of set equality and auxiliary lemmas that allow us to rewrite terms using this equality.

An example where we need such auxiliary lemmas is the following. We would like to rewrite $\mathcal{V}[T](\eta)$ to $\mathcal{V}[T](\eta')$ if we know η and η' are equal, but because sets and type variable mappings both use custom notions of equality, we need to provide lemmas that explicitly justify this transformation. We prove a number of lemmas regarding such preservation of equality under semantic structures. We also need proofs of reflexivity, symmetry and transitivity of our equality relations in order to use them effectively. Though technically easy, these proofs are tedious to write, and could perhaps be avoided with richer facilities for proving equality in our reasoning language.

7. Related and Future Work

Mechanizing logical relations proofs is not a new achievement, with normalization proofs of System F having being mechanized since the early 1990's, described for example in (Altenkirch 1993) or (Berardi 1990). Mendler discovered his recursion strategy even earlier (Mendler 1988). However, we have not seen a mechanized normalization proof for a total language with general recursive types and this style of recursion. In doing this we have also given a concise set-theoretic semantics, which we have not seen for recursive types in this direct form. There are on the other hand encodings of recursive types into normalizing languages (Abel et al. 2005), namely System F^{ω}, which were the inspiration for our interpretation.

Our formulation of the normalization problem lends itself to a relatively straight-forward mechanization, which could be instructive for those carrying out similar proofs in the community. Our approach involving an environment-based evaluation and semantically interpreted substitutions can be adapted to other object languages and proof environments. These techniques are beneficial because they allow us to avoid reasoning about term-level substitutions and organize our proofs about type-level substitutions in a modular structure.

A design decision we made during the mechanization was whether to use an explicit substitution calculus (Abadi et al. 1990) to handle substitutions on the type level. We did not take this route because it slightly obscures the surface-level presentation of our object language. However, we suspect the technique could simplify our reasoning about type substitutions by avoiding the need for renamings. Recall that renamings were an auxiliary notion to help us pass Coq's termination checking. With explicit substitutions, equations about substitutions which are cases of functions in our treatment become lemmas to prove, and vice versa. It may be that some of our functions written in terms of renamings to be evidently total can be proven without renamings in the alternative approach. This is one that we can try in the future and see how it changes the proof.

There are a number of ways in which one could extend our proof as well. A straight-forward first extension would be that to include polymorphism as in System F. We have formulated this on paper already, although not yet in the Coq development. More interestingly, we would like to consider how this proof extends to richer type theories, namely dependent types or indexed types. We believe that our proof framework is easily extensible to such systems, which would allow a formalization of a rich total language comparable to current proof languages such as Agda and Coq. In particular, it would allow us to mechanize the meta-theory of Beluga extending (Pientka and Abel 2015).

Finally we would like to extend our formalization to more sophisticated forms of Mendler-style recursion, such as course-of-value recursion (Abel 2010), which uses subtyping to permit a wider range of programs. This would lead to the first mechanization of such recursion schemes as far as we know.

8. Conclusion

We have given a tutorial-style explanation of a normalization proof for a simply-typed lambda calculus with Mendler-style recursion. Compared to existing normalization proofs that rely on ordinals, our reducibility candidates can be formulated solely in second-order logic. This leads to a compact mechanization in Coq despite the overhead of modelling type variables and type-level substitutions due to the presence of recursive types. Such overhead can in principle be eliminated by using proof environments that support higher-order abstract syntax (HOAS) encodings. However, these proof environments currently do not support higher-kinded polymorphism, which we require to encode our semantics. We see our work as a case study to understand the limitations and strengths of current proof environments. In particular, we believe it serves as a good challenge problem to proof environments that support HOAS encodings. Adding support for higher-kinded polymorphism to these systems seems key to mechanizing semantic models of type systems or logics to reason about concurrent, higher-order, and imperative languages (see for example (Ahmed 2004; Perconti and Ahmed 2014; Sieczkowski et al. 2015)) and to broaden their application domain.

References

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lèvy. Explicit substitutions. In *17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 31–46. ACM Press, 1990.

A. Abel. Termination checking with types. *RAIRO - Theoretical Informatics and Applications*, 38(4):277–319, 3 2010.

A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333: 3–66, 2005. ISSN 03043975.

A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993. ISBN 3-540-56517-5.

D. Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1):2:1–2:44, 2012.

D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

N. Benton, C.-K. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012.

S. Berardi. Girard normalization proof in LEGO. In *Proceedings of the First Workshop on Logical Frameworks*, pages 67–78, 1990.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989. ISBN 0-521-37181-3.

N. P. F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, USA, 1988. AAI8804634.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007. Technical Report 33D.

J. T. Perconti and A. Ahmed. Verifying an open compiler using multi-language semantics. In Z. Shao, editor, *23rd European Symposium on Programming Languages and Systems ESOP*, Lecture Notes in Computer Science (LNCS 8410), pages 128–148. Springer, 2014. doi: 10.1007/978-3-642-54833-8_8.

B. Pientka and A. Abel. Structural recursion over contextual objects. In T. Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 273–287. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 2015.

B. Pientka and A. Cave. Inductive Beluga: Programming proofs (System description). In A. P. Felty and A. Middeldorp, editors, *25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 2015.

F. Sieczkowski, A. Bizjak, and L. Birkedal. Modures: A Coq library for modular reasoning about concurrent higher-order imperative programming languages. In C. Urban and X. Zhang, editors, *6th International Conference of Interactive Theorem Proving (ITP)*, Lecture Notes in Computer Science (LNCS 9236), pages 375–390. Springer, 2015. doi: 10.1007/978-3-319-22102-1_25.

W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.