Course-of-Value Induction in Cedille

Denis Firsov, Larry Diehl, Christopher Jenkins, and Aaron Stump

University of Iowa, USA
{firstname-lastname}@uiowa.edu

──── **Abstract** ────

In the categorical setting, histomorphisms model a course-of-value recursion scheme that allows functions to be defined using arbitrary previously computed values. In this paper, we use the Calculus of Dependent Lambda Eliminations (CDLE) to derive a lambda-encoding of inductive datatypes that admits course-of-value induction. Similar to course-of-value recursion, course-of-value induction gives access to inductive hypotheses at arbitrary depth of the inductive arguments of a function. We show that the derived course-of-value datatypes are well-behaved by proving Lambek's lemma and characterizing the computational behavior of the induction principle. Our work is formalized in the Cedille programming language and also includes several examples of course-of-value functions.

1 Introduction

Dependently typed programming languages with built-in infrastructure for defining inductive datatypes allow programmers to write functions with complex recursion patterns. For example, in Agda [12] we can implement the natural definition of Fibonacci numbers:

```
fib : Nat → Nat
fib zero = zero
fib (suc zero) = 1
fib (suc (suc n)) = fib (suc n) + fib n
```

This definition is accepted by Agda because its built-in termination checker sees that all recursive calls are done on structurally smaller arguments. In contrast, in pure polymorphic lambda calculi (e.g., System F), inductive datatypes can be encoded by means of impredicative quantification (without requiring additional infrastructure). For example, if we assume that `F` is a well-behaved positive scheme (e.g., a functor), then we can express its least fixed point as an initial Mendler-style F-algebra. A Mendler-style algebra differs from a traditional F-algebra (`F X → X`) in that it takes an additional argument (of type `R → X`), which corresponds to a function for making recursive calls. Mendler-style algebras introduce a polymorphic type `R` for *recursive* scheme arguments, allowing recursive function calls to be restricted to structurally smaller arguments. At the same time, the polymorphic type prevents any kind of further inspection of those arguments (for the remainder of this paper we switch to code written in Cedille [16], a dependently typed programming language supporting impredicative type quantification):

```
AlgM ◀ (⋆ → ⋆) → ⋆ → ⋆ = λ F. λ X. ∀ R : ⋆. (R → X) → F R → X.
FixM ◀ (⋆ → ⋆) → ⋆ = λ F. ∀ X : ⋆. AlgM F X → X.
foldM ◀ ∀ F : ⋆ → ⋆. ∀ X : ⋆. AlgM F X → FixM F → X
 = Λ F. Λ X. λ alg. λ v. v alg.
```

The simple recursion pattern provided by `foldM` (also known as catamorphism) can be tricky to work with. Let us define natural numbers as the least fixed point of the functor

`NF X = 1 + X`. Hutton [6] explained that it is possible to use the universality law of initial F-algebras to show that there is no algebra `g : AlgM NF Nat` such that `fib = foldM g`. The reason for this is that in the third equation (of the natural definition of `fib`), the recursive call is made not only on the direct predecessor of the argument (`suc n`), but also on the predecessor of the predecessor (`n`).

The usual workaround involves "tupling". More specifically, we define an algebra `AlgM NF (Nat × Nat)` where the second `Nat` denotes the previous Fibonacci number. Then, we fold the input with `fibAlg`, and finally return the first projection of the tuple (here π_i denotes `i`-th projection from the tuple):

```
fibAlg ◄ AlgM NF (Nat × Nat) = Λ R. λ rec. λ fr.
 case fr (λ _.  pair (suc zero) zero)      % zero case
         (λ r. let p = rec r in           % suc case
              pair (add (π₁ p) (π₂ p)) (π₁ p)).
fibTup ◄ Nat → Nat = λ n. π₁ (foldM fibAlg n).
```

In this example, the `rec` function allows recursive calls to be made explicitly on elements of type `R` (which is `Nat` in disguise). This approach requires error-prone bookkeeping. Additionally, observe that the defining equation of the Fibonacci numbers (`fibTup (suc (suc n)) = fibTup n + fibTup (suc n)`) is true propositionally, but not definitionally (i.e., it does not follow by β-reduction).

The alternative solution to tupling is course-of-value recursion (also known as histomorphism), which makes it possible to express nested recursive calls directly. The central concept of this approach is course-of-value F-algebras, which are similar to usual Mendler-style F-algebras, except that they take an abstract destructor (of type `R → F R`) as yet another additional argument. The abstract destructor is a fixed-point unrolling (or, abstract inverse of the initial algebra), and intuitively allows for "pattern-matching" on constructors for the scheme `F`.

```
AlgCV ◄ (⋆ → ⋆) → ⋆ → ⋆ = λ F. λ X.
 ∀ R : ⋆. (R → F R) → (R → X) → F R → X.
```

For illustration purposes, assume that `F` is a functor and that `FixCV F` is its least fixed point. Also, assume that `inCV` and `outCV` are mutual inverses and represent a collection of constructors and destructors, respectively.

```
inCV  ◄ F (FixCV F) → FixCV F = <..>
outCV ◄ FixCV F → F (FixCV F) = <..>
```

Then, course-of-value recursion is characterized by the function `foldCV`, and its reduction behaviour is characterized by the cancellation property (`cancel`):

```
foldCV ◄ ∀ X : ⋆. AlgCV F X → FixCV F → X = <..>
cancel ◄ ∀ X : ⋆. ∀ alg : AlgCV F X. ∀ fx : F (FixCV F).
  foldCV alg (inCV fx) ≃ alg outCV (foldCV alg) fx = β.
```

Notice that after unfolding, the first argument of `alg` is instantiated with `outCV` (the destructor), and the second argument is instantiated with a partially applied `foldCV` (the recursive call).

To illustrate the nested recursive calls, we define course-of-value naturals (`NatCV`) as the course-of-value least fixed point of the functor `NF (FixCV NF)`. Then, the Fibonacci function can be implemented very close to the conventional "pattern-matching" style:

```
fibCV ◀ NatCV → NatCV = foldCV (Λ R. λ out. λ rec. λ nf.
 case nf (λ _. zero)                  % zero case
  (λ r. case (out r) (λ _. suc zero)  % (suc zero) case
           (λ r'. rec r + rec r'))).  % (suc (suc n)) case
```

Here, `out` provides an additional layer of pattern-matching on the arguments of the function. Finally, it is important to observe that given that `cancel` is true by β-reduction, then `fibCV (suc (suc n)) ≃ fibCV (suc n) + fibCV n` is also true by β-reduction.

The remaining questions is how to define the least fixed point `FixCV F` for any positive scheme `F`, and how to derive the corresponding introduction and elimination principles. We might try a usual construction in terms of universal quantification and `AlgCV`:

```
FixCV ◀ (⋆ → ⋆) → ⋆ = λ F. ∀ X : ⋆. AlgCV F X → X.
```

This definition fails since `AlgCV F X` is isomorphic to `AlgM (Enr' F) X` where

```
Enr' ◀ (⋆ → ⋆) → ⋆ → ⋆ = λ F. λ X. F X × (X → F X).
```

`Enr' F` is a *negative* scheme and (in general) the least fixed points of negative schemes are undefined in a consistent type theory. As a result, it is common to implement `foldCV` in terms of general recursion from the host language or to add it as a primitive construction (see related work in Section 6).

The **main contribution** of this paper is the derivation of course-of-value datatypes in the Calculus of Dependent Lambda Eliminations (CDLE). The key inspiration for our work comes from the categorical construction known as *restricted existentials* (Section 3). We prove that the least fixed point of the restricted existential of scheme `Enr' F` exists, and that it contains course-of-value datatypes as a subset (Section 4.1). Next, we employ heterogeneous equality from CDLE to define the datatype `FixCV` as this subset (Section 4.2). We also give (according to our best knowledge) the first generic formulation and derivation of a course-of-value induction principle in a pure type theory. Finally, we show examples of functions and proofs defined over course-of-value natural numbers (Section 5).

The CDLE type theory is implemented in Cedille, which we use to type-check the formalized development of this paper.[1] We emphasize that part of the significance of our work is *deriving* course-of-value induction within the small core calculus of CDLE. While the Calculus of Inductive Constructions (CIC) [9] directly adds support for inductive datatypes to the Calculus of Constructions (CC) in an involved way, CDLE is a minor extension of CC that makes inductive datatypes derivable rather than built-in.

2 Background

2.1 The CDLE Type Theory

CDLE [13, 15] is an extrinsically typed (or, Curry-style) version of the Calculus of Constructions (CC), extended with a heterogeneous equality type $(\mathtt{t}_1 \simeq \mathtt{t}_2)$[2], Kopylov's [7] dependent intersection type (ι x : T. T'), and Miquel's [10] implicit product type (∀ x : T. T', for erased arguments). To make type checking algorithmic, Cedille terms have typing annotations,

---

and definitional equality of terms is modulo erasure of these annotations. The target of erasure in Cedille is the pure untyped lambda calculus with no additional constructs. Due to space constraints, we omit a more detailed summary of CDLE. However, this work is a direct continuation of our previous work [3], which includes a detailed explanation of all of the constructs of Cedille.

2.2 Identity Functions and Identity Mappings

Stump showed how to derive induction for natural numbers in CDLE [14]. This result was generically extended to achieve induction for arbitrary datatypes that arise as a least fixed point of a functor [4]. One final generalization restricted `fmap` to be defined over functions that extensionally behave like identity functions (the type `Id`). Least fixed points of such "identity mapping" (the type `IdMapping`) schemes can be defined for a larger collection of datatypes, compared to functors. We omit the implementations below (indicated by `<..>`), but a detailed description of these constructs can be found in [3].

2.2.0.1 Identity Functions

We define the type `Id X Y` as the collection of all functions from `X` to `Y` that erase to the term $(\lambda\ x.\ x)$:

`Id ◀ ⋆ → ⋆ → ⋆ = λ X. λ Y. Σ f : X → Y. f ≃ (λ x. x).`

Because Cedille is extrinsically typed, the domain (`X`) and codomain (`Y`) of an identity function need not be the same. An identity function `Id X Y` can be eliminated to "cast" values of type `X` to values of type `Y` without changing the values themselves:

`elimId ◀ ∀ X Y : ⋆. ∀ c : Id X Y. X → Y = <..>`

Note that argument `c` of `elimId` is quantified using ∀ (rather than Π), indicating that it is an implicit (or, erased) argument. Significantly, the erasure `|elimId -c|` (the dash syntactically indicates that this an implicit, or erased, application) results in the identity function $(\lambda\ x.\ x)$[3].

2.2.0.2 Identity Mappings

We say that scheme `F` is an identity mapping if it is equipped with a function that lifts identity functions:

`IdMapping ◀ (⋆ → ⋆) → ⋆ = λ F. ∀ X Y: ⋆. Id X Y → Id (F X) (F Y).`

Intuitively, `IdMapping F` is similar to a functor's `fmap`, but it only needs to be defined on identity functions, and no additional laws are required. Every functor induces an identity mapping, but not vice versa [3].

---

[3] Types, as opposed to values, are always erased in terms. Hence, using (∀ X : ⋆) in the classifier of a term is sensible but using (Π X : ⋆) is not. Additionally, we omit type applications in terms because they are inferred by Cedille.

2.3 Inductive Datatypes in Cedille

Next, we review the generic datatype constructions (from [3]) definable using the constructions above. To start with, we specify a scheme `F` and its identity mapping as module-level parameters (such that every definition in this section begins with these parameters).

```
module (F : ⋆ → ⋆){imap : IdMapping F}.
```

Curly braces around the `imap` variable indicate that it is quantified implicitly (or, as an erased parameter). Another way of saying this is that none of the definitions should depend on the computational behaviour of `imap`.

The fixpoint type (`FixIndM ◀ ⋆`) is defined as an intersection of `FixM` and a proof of its inductivity (see [3] for details). `FixIndM` comes with a constructor (`inFixIndM ◀ F FixIndM → FixIndM`), and its mutually inverse destructor (`outFixIndM ◀ FixIndM → F FixIndM`). The induction principle for `FixIndM` takes a "dependent" counterpart to Mendler-style F-algebras (`AlgM`), which we call Q-proof-algebras (`PrfAlgM`):

```
induction ◀ ∀ Q : FixIndM → ⋆. PrfAlgM Q → Π e : FixIndM. Q e = <..>
```

A value of type `PrfAlgM Q` should be understood as an inductive proof that predicate `Q` holds for every `FixIndM` built by constructors `inFixIndM`. Just like F-algebras, Q-proof-algebras allow users to invoke inductive hypotheses only on direct subdata of a given argument. The rest of the paper is devoted to the formulation and derivation of a generic course-of-value induction principle that allows users to invoke inductive hypotheses on subdata at arbitrary depths (realized as `inductionCV` for any `PrfAlgCV` in Section 4.2).

3 Restricted Existentials

Uustalu and Vene defined a construction called the restricted existential to demonstrate an isomorphism between Church-style and Mendler-style initial algebras [17]. The importance of this is that for any difunctor (or, mixed variant functorial scheme) `F`, the restricted existential of `F` is an isomorphic covariant functor.

In this section, we define a variation that we call an *identity restricted existential*. We also derive its dependent elimination principle, and prove that the identity restricted existential of any scheme `F` (including negative and non-functorial ones) is an *identity mapping*. Later in the paper, the restricted existential will be the main tool for deriving course-of-value datatypes.

3.1 Restricted Coends

In the categorical setting, the restricted existential arises as a restricted coend. Our subsequent development requires existentials where the quantifier ranges over types. This can be provided by a restricted coend (`RCoend H F`), which is isomorphic to the existential type `∃ R. H R × F R` (where `H` is what we are restricting by). Our development defines `RCoend` by taking advantage of the isomorphism between the universal type `∀ R. H R → F R → Q` and the existential type `(∃ R. H R × F R) → Q` (for any `Q`) that we have in mind. Now, let us formalize the notion of restricted coend.

Let $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ be an endodifunctor and $H : \mathcal{C}^{op} \times \mathcal{C} \to Set$ be a difunctor to *Set*. An H-restricted F-coend is an initial object in the category of H-restricted F-cowedges. An H-restricted F-cowedge is a pair (C, Φ) where C (the carrier) is an object in $\mathcal{C}$ and $\{\Phi_R\}_{R \in \mathcal{C}}$ is a family of functions (dinatural transformations) between sets $H\ R\ R$ and $\mathcal{C}(F\ R\ R, C)$.

We translate this definition to Cedille, where an *H*-restricted *F*-cowedge (C, Φ) corresponds to a type (C) and a polymorphic function (RCowedge H F C):

RCowedge ◀ (⋆ → ⋆) → (⋆ → ⋆) → ⋆ → ⋆
= λ H. λ F. λ C. ∀ R : ⋆. H R ⇒ F R → C.

To simplify the subsequent development, we render difunctors as schemes with a single parameter, and the restriction H R is made implicit (denoted by ⇒, which is a non-dependent version of ∀). The simplification of making the restriction parameter erased allows us to avoid needing function extensionally to achieve our encoding. The carrier of the initial cowedge can be implemented in terms of universal quantification:

RCoend ◀ (⋆ → ⋆) → (⋆ → ⋆) → ⋆ = λ H. λ F. ∀ C : ⋆. RCowedge H F C → C.

The second component of initial cowedges is a polymorphic function, (intrRCoend), which plays the role of the constructor of its carrier (RCoend H F), and is implemented as follows:

intrRCoend ◀ ∀ H F : ⋆ → ⋆. RCowedge H F (RCoend H F)
= Λ H. Λ F. Λ R. Λ ac. λ ga. (Λ Y. λ q. q R -ac ga).

The (weak) initiality can be proved by showing that for any cowedge RCowedge H F C, there is a homomorphism from RCoend H F to C:

elimRCoend ◀ ∀ H F: ⋆ → ⋆.∀ C: ⋆. RCowedge H F C → RCoend H F → C
= Λ F. Λ A. Λ C. λ phi. λ e. e phi.

3.2 Dependent Elimination for Restricted Coends

In this section, we utilize the intersection type (denoted by ι in Cedille) to define a restricted coend type for which the induction principle is provable. To do this, we follow the original recipe described by Stump to derive natural-number induction in Cedille. First, we define a predicate expressing that an H-restricted F-coend is inductive.

RCoendInductive ◀ Π H F : ⋆ → ⋆. RCoend H F → ⋆
= λ H. λ F. λ e. ∀ Q : RCoend H F → ⋆.
(∀ R : ⋆. ∀ hr : H R. Π fr : F R. Q (intrRCoend -hr fr)) → Q e.

Second, we define the "true" inductive restricted coend as an intersection of the previously defined RCoend and the predicate RCoendInductive. In essence, this says that RCoendInd is the subset of RCoend carved out by the RCoendInductive predicate.

RCoendInd ◀ (⋆ → ⋆) → (⋆ → ⋆) → ⋆
= λ H. λ F. ι x : RCoend H F. RCoendInductive H F x.

This definition builds on an observation by Leivant that under the Curry-Howard isomorphism, proofs in second-order logic that data satisfy their type laws can be seen as isomorphic to the Church-encodings of those data [8]. Next, we define the constructor for the inductive coend:

intrRCoendInd ◀ ∀ H F : ⋆ → ⋆. RCowedge H F (RCoendInd H F)
= Λ H. Λ F. Λ R. Λ hr. λ fr.
[intrRCoend -hr fr , Λ Q. λ q. q R -hr fr].

In Cedille, the term [t , t'] introduces the intersection type ι x : T. T' x, where t has type T and t' has type [t/x]T'. Definitionally, values of intersection types reduce (via erasure) to their first components (i.e., [t , t'] is definitionally equal to t). See [3] for more information on intersection types in Cedille. The induction principle is now derivable and has the following type:

```
indRCoend ◄ ∀ H F : ⋆ → ⋆. ∀ Q : RCoendInd H F → ⋆.
  (∀ R : ⋆. ∀ hr : H R. Π fr : F R. Q (intrRCoendInd  -hr fr))
   Π e : RCoendInd H F. Q e = <..>
```

3.3   Identity Restricted Existentials

We define the identity restricted existential of `F` and the object `C` as an `F`-coend restricted by
a family of identity functions `λ X. Id X C`:

```
RExtInd ◄ (⋆ → ⋆) → ⋆ → ⋆ = λ F. λ X. RCoendInd (λ R : ⋆. Id R X) F.
```

Next, we prove that the restricted existential of any `F` is an identity mapping:

```
imapRExt ◄ ∀ F : ⋆ → ⋆. IdMapping (RExtInd F)
 = Λ F. Λ A. Λ B. Λ f. λ c. indRCoend c
   (Λ R. Λ i. λ gr. pair (intrRExtInd -(compose i f) gr) β).
```

Intuitively, `RExtInd F X` corresponds to the type $\exists$ `R. Id R X × F R`. Notice that `RExtInd F X`
is positive because `X` occurs positively in `Id`, and that positivity does not depend on `F`. With
the definition of identity restricted existentials in place, we can now move on towards using
them to derive course-of-value induction.

4   Course-of-Value Datatypes

In this section we review why the naive scheme for an inductive datatype with a destructor
does not work out due to negativity (as mentioned in the introduction, Section 1). Then,
we demonstrate how *identity restricted existentials* (`RExtInd` of Section 3) can be used in
our novel encoding to overcome this limitation, allowing us to define datatypes supporting
course-of-value induction. The development in this section is parameterized by an identity
mapping:

```
module (F : ⋆ → ⋆){imap : IdMapping F}.
```

4.1   Precursor

In [17], Uustalu and Vene showed that it is possible to use restricted existentials to derive
a superset of course-of-value natural numbers. We start by generalizing their construction
to arbitrary inductive types, in terms of least fixed points of identity mappings. The main
idea is to define a combinator that pairs the value `F X` with the destructor function (of type
`X → F X`):

```
Enr' ◄ ⋆ → ⋆ = λ X. F X × (X → F X).
```

Intuitively, we wish to construct a least fixed point of `F` and its destructor simultaneously.
The resulting scheme `Enr' F` is not positive and therefore it cannot be a functor nor an
identity mapping. This implies that we cannot take a least fixed point of it directly. Instead,
we define `CVF' F` as a restricted existential of `Enr' F`. Hence, the scheme `CVF' F` is an
identity mapping by the property of restricted existentials:

```
CVF' ◄ ⋆ → ⋆ = RExtInd (Enr' F).
imCVF' ◄ IdMapping (CVF' F) = imapRExt (Enr' F).
```

It is natural to ask what the relationship between the least fixed point of `F` and least fixed
point of `CVF' F` is.

```
FixCV' ◀ ⋆ = FixIndM (CVF' F) -(imCVF' F).
```

It turns out that `FixCV'` is not a least fixed point of `F`, because value `F FixCV'` could be paired with any function of type `FixCV' → F FixCV'`. We will provide more intuition by describing the destructor and constructor functions of `FixCV'`.

4.1.0.1 Destructor

The generic development from Section 2.3 allows us to unroll `FixCV'` into a value of `CVF' FixCV'` (which it was made from). Because `CVF' F` is a restricted existential, we can use its dependent elimination to "project out" the value `F FixCV'`:

```
outCV' ◀ FixCV' → F FixCV' = λ x. indRExt (outFixIndM -imapRExt x)
    (Λ R. Λ c. λ v. elimId -(imap -c) (π₁ v)).
```

In the definition above, the variable `v` has type `F R × (R → F R)`. Because `F` is an identity mapping, we can cast the first projection of `v` to `F FixCV'` and return it. On the other hand, the function `R → F R` cannot be casted to type `FixCV' → F FixCV'`, because the abstract type `R` appears both positively and negatively.

4.1.0.2 Constructor

Similarly, the generic development gives us the function `inFixIndM`, which constructs a `FixCV'` value from the given `CVF' FixCV'`. The latter must be built from a pair of `F FixCV'` and a function of type `FixCV' → F FixCV'`. This observation gives rise to the following specialized constructor of `FixCV'`:

```
inCV' ◀ (FixCV' → F FixCV') → F FixCV' → FixCV' = <..>
```

This constructor indicates that `FixCV'` represents the superset of course-of-value datatypes, because the function `FixCV' → F FixCV'` is not restricted to the destructor `outCV'`, and the inductive value might contain a different function of that type at every construtor. We address this issue in the next section.

4.2 Course-of-Value Datatypes with Induction

In our previous work [3], we developed a generic unrolling function for least fixed points of identity mappings:

```
outFixIndM ◀ ∀ imap : IdMapping F. FixIndM F → F (FixIndM F) = <..>
```

Observe that the only identity-mapping-specific variable is quantified implicitly. In other words, `outFixIndM` does not perform any `F`-specific computations. The same is true for the elimination principle of restricted existentials (`indRCoend`). Since `outCV'` is implemented in terms of these functions, this observation suggests that we can refer to `outCV'` as we define the subset of type `FixCV'`. In particular, we define the scheme `Enr` by pairing the value `F X` with the function `f : X → F X` and the proof that this function is equal to the previously defined `outCV'`:

```
Enr ◀ ⋆ → ⋆ = λ X. F X × Σ f : X → F X. f ≃ outCV'.
```

This constraint between terms of different types is possible due to heterogeneous equality. Just like in the previous section, we define a least fixed point of the restricted existential of `Enr F` and its least fixed point:

```
CVF ◀ ⋆ → ⋆ = λ X. RExtInd (Enr F) X.
FixCV ◀ ⋆ = FixIndM CVF (imapRExt Enr).
```

4.2.0.1 Destructor

The destructor of `FixCV` is represented by exactly the same lambda-term as the destructor
(`outCV'`) of FixCV':

```
outCV ◄ FixCV → F FixCV = λ v. indRExt
 (outFixIndM -imapRExt v) (Λ R. Λ c. λ v. elimId -(imap -c) (π₁ v)).
outCVEq ◄ outCV' ≃ outCV = β.
```

Because the only difference between `outCV'` and `outCV` is their typing annotations (which
are inferred by the typechecker), they are definitionally equal in Cedille (as witnessed by β,
the introduction rule of Cedille's equality type).

4.2.0.2 Constructor

Armed with the destructor `outCV` and the proof `outCVEq`, we can now define the constructor
of `FixCV`:

```
inCV ◄ F FixCV → FixCV = Λ F. Λ imap. λ fcv. inFixIndM -imapRExt
 (intrRExtInd -trivIdExt) (pair fcv (pair (outCV -imap) outCVEq)).
```

4.2.0.3 Lambek's Lemma

As expected, `inCV` and `outCV` are mutual inverses, which establishes that `FixCV` is a fixed
point of `F`.

```
lambekCV1 ◄ ∀ x : F FixCV. outCV (inCV x) ≃ x = β.
lambekCV2 ◄ ∀ x : FixCV.   inCV (outCV x) ≃ x = <..>
```

Note that `lambekCV1` holds definitionally, while `lambekCV2` is provable by straightforward
induction (actually, the proof only uses dependent case analysis, and ignores the inductive
hypothesis).

4.2.0.4 Induction

Recall that the induction principle for the least fixed point `FixIndM` is stated in terms of
proof-algebras (`PrfAlgM` in Section 2.3). Now, let us define proof-algebras for course-of-value
datatypes:

```
PrfAlgCV ◄ (FixCV → ⋆) → ⋆ = ∀ R : ⋆. ∀ c : Id R FixCV.
 Π out : R → F R. out ≃ outCV ⇒ Π ih : Π r : R. Q (elimId -c r).
 Π fr : F R.  Q (inCV (elimId -(imap -c) fr))
```

Compared to `PrfAlgM` [3], `PrfAlgCV` has an extra argument (`out : R → F R`), which rep-
resents an unrolling (or, abstract predecessor) function for abstract type `R`. Additionally, we
have a proof that the `out` function is equal to the previously discussed destructor `outCV`.
This evidence is needed when the construction of a particular proof-algebra depends on
the exact definition of the predecessor function. Besides those two extra arguments, the
identity function (`c`) from abstract `R` to concrete `FixCV`, the inductive hypothesis (`ih`), and
the subdata (`fr`) are the same as in `PrfAlgM` [3].

Course-of-value induction is expressible in terms of course-of-value proof-algebras and
is proved by combining the induction principle of `FixIndM` with the dependent elimination
principle of restricted existentials.

```
inductionCV ◀ ∀ Q : FixCV → ⋆. PrfAlgCV Q → Π x : FixCV. Q x = <..>
```

It is important to establish the computational behaviour of this proof-principle:

```
indCancel ◀ ∀ Q : FixCV → ⋆. ∀ palg : PrfAlgCV Q. ∀ x : F FixCV.
 inductionCV palg (inCV x) ≃ palg outCV (inductionCV palg) x = β.
```

Above, notice how the abstract unrolling function `out : R → F R` is being instantiated with the actual unrolling function `outCV`. Finally, implementing course-of-value recursion (`foldCV`) from Section 1 in terms of course-of-value induction (`inductionCV`) is straightforward.

Finally, note that while `FixIndM` and `FixCV` values are not definitionally equal, their types are isomorphic, and they have the same asymptotics. Hence, in practice high-level datatype declarations can always elaborated to their course-of-value encodings, rather than maintaining two encodings for every type.

5 Examples

We now demonstrate the utility of our results with example functions and proofs on natural numbers that require course-of-value recursion and induction. Note that the `fibCV` example from the introduction (Section 1) works as described, because `foldCV` is derivable from `inductionCV`. Recall that natural numbers may be defined as the least fixed point of a functor (`NF ◀ ⋆ → ⋆ = λ X. Unit + X.`). As remarked in Section 2.2.0.2, because `NF` is a functor, it is also an identity mapping (`nfimap ◀ IdMapping NF = <..>`). We begin by defining the type of natural numbers (`NatCV`), supporting a constant-time predecessor function, as well as course-of-value induction:

```
NatCV ◀ ⋆ = FixCV F nfimap.
zero ◀ NatCV = inCV -nfimap (in1 unit).
suc ◀ NatCV → NatCV = λ n. inCV -nfimap (in2 n).
pred ◀ NatCV → NatCV = λ n. case (outCV -nfimap n) (λ u. n) (λ n'. n').
```

5.1 Division

Consider an intuitive definition of division as iterated subtraction:

```
div : Nat → Nat → Nat
div 0 m = 0
div n m = if (n < m) then 0 else (suc (div (n - m) m))
```

Such a definition is rejected by Agda (and many languages like it), because Agda requires that recursive calls are made on arguments its termination checker can guarantee are structurally smaller, which it cannot do for an arbitrary expression (like `n - m`). With our development, the problematic recursive call (on `n - m`) is an instance of course-of-value recursion because we can define subtraction by iterating the predecessor function, and we have access to recursive results for every predecessor.

For convenience, we define the conventional `foldNat` as a specialized version of our generic development. Then, `minus n m` is definable as the `m` number of predecessors of `n`.

```
foldNat ◀ ∀ R : ⋆. (R → R) → R → NatCV → R
 = Λ R. λ rstep. λ rbase. foldCV (Λ R'. λ out. λ rec. λ nf.
 case nf (λ _. rbase) (λ r'. rstep (rec r'))).
minus' ◀ ∀ R : ⋆. (R → NF R) → R → NatCV → R
```

```
  = Λ R. λ pr. foldNat (λ r. case (pr r) (λ _. r) (λ r'. r'))
minus ◄ NatCV → NatCV → NatCV = minus' (outCV -nfimap).
```

Above, we first define an abstract operation `minus' n m`, where the type of `n` is polymorphic and where that type comes with an abstract predecessor `pr`. Then, the usual concrete `minus n m` is recovered by using `NatCV` for the polymorphic type and the destructor `outCV -nfimap` for the predecessor.

Now we can use `minus'` to define division naturally, returning zero in the base case, and iterating subtraction in the step case. This definition below is accepted purely through type-checking and without any machinery for termination-checking.

```
div ◄ NatCV → NatCV → NatCV
  = λ n. λ m. inductionCV (Λ R. Λ c. λ pr. Λ preq. λ ih. λ nf.
  case nf
    (λ x. zero)                    % div 0 m
    (λ r. if (suc (elimId -c r) < m)    % div (suc n) m
      then zero
      else (suc (ih (minus' pr r (pred m)))))) n
```

Notice that in the conditional statement, we use `elimId -c` to convert the abstract predecessor `r` to a concrete natural number, allowing us to apply `suc` to check if `suc n` is less than `m`. In the intuitive definition of `div`, we match on `0` in the first case, and on any wildcard pattern `n` in the second case. In contrast, when using `inductionCV` and `case` in our example, we must explicitly handle the `zero` and `suc r` cases. Consequently, while the intuitive definition recurses on `div (n - m) m`, we recurse on the predecessors `ih (minus' pr r (pred m))`. This is equivalent because `minus (suc n) (suc m)` is equal to `minus n m`, for all numbers `n` and `m`. We can also prove properties about our development, such as the aforementioned equivalence (`minSucSuc` below, proven by ordinary induction). By direct consequence, we can also prove that the defining equation (`divSucSuc` below, for the successor case) of the intuitive definition of division holds (by ordinary induction and rewriting by `minSucSuc`):

```
minSucSuc ◄ Π n m : NatCV. minus (suc n) (suc m) ≃ minus n m = <..>
divSucSuc ◄ Π n m : NatCV. (suc n < suc m) ≃ ff →
  div (suc n) (suc m) ≃ suc (div (minus (suc n) (suc m)) (suc m)) = <..>
```

While the propositions are stated in terms of concrete `minus`, the `div` function is defined in terms of abstract `minus'`. Nonetheless, the propositions are provable due to the computational behavior of `inductionCV`, which instantiates the bound `pr` with `outCV`, allowing us to identify `minus' pr` and `minus`.

5.2   Property of Division

Besides this section, this paper contains (mostly omitted) proofs by definitional equality, rewriting, dependent case-analysis, and/or ordinary induction. In this section we prove a property of division that takes full advantage of course-of-value induction (our primary contribution).

We assume the existence of a less-than-or-equal relation (whose definition is omitted for space reasons) on course-of-value naturals (`LE ◄ NatCV → NatCV → ⋆`), with two constructors for evidence in the zero case (`leZ ◄ Π n : NatCV. LE zero n`), and in the successor case (`leS ◄ ∀ n m : NatCV. LE n m → LE (suc n) (suc m)`). Additionally,

we will need a lemma that LE is transitive (`leTrans ◄ ∀ x y z : NatCV. LE x y → LE y z → LE x z`), and a lemma that subtraction decreases a number or keeps it the same (`leMinus ◄ Π n m : NatCV. LE (minus n m) n`). Both of these lemmas are provable by ordinary induction. Now, let's prove our property of interest by *course-of-value induction*, namely that division also decreases a number or keeps it the same:[4]

```
leDiv ◄ Π n m : NatCV. LE (div n m) n
  = λ n. λ m. inductionCV (Λ R. Λ c. λ pr. Λ preq. λ ih. λ nf.
  case nf
    (λ u. ρ (etaUnit u) - leZ zero)  % Goal: LE (div zero m) zero
    (λ r. let n1 = elimId -c r in    % Goal: LE (div (suc n) m) (suc n)
     if (suc n1 < m)
     then (leZ (suc n1)) % Goal: LE zero (suc n)
     else let            % Goal: LE (suc (div (minus n (pred m)) m)) (suc n)
       n2 = minus n1 (pred m)
       n3 = div n2 m
       n3LEn2 = ρ (sym preq) - ih (minus' pr r (pred m))
       n2LEn1 = leMinus n1 (pred m)
       in leS -n3 -n1 (leTrans -n3 -n2 -n1 n3LEn2 n2LEn1))).
```

5.2.0.1 Zero Case

The expression `div zero m` in the goal reduces to `zero`, allowing us to conclude `LE zero zero` by using the constructor `leZ`. The only caveat is that the reduction requires rewriting (using Cedille's ρ primitive) by the uniqueness of unit (`etaUnit`), because our generic encoding of `NatCV` uses the unit type in the left part of the sum (`NF X = 1 + X`).

5.2.0.2 Successor Case (When the Conditional is True)

At first, the successor case of division is prevented from reducing further because it is branching on a conditional statement (`suc (elimId -c r) < m`). In the true branch of this conditional, the goal reduces and is immediately solvable using `leZ`. Note that we use a let statement to name the result (variable `n1`) of converting the abstract predecessor `r : R` to a concrete `NatCV` via the expression `elimId -c r`.

5.2.0.3 Successor Case (When the Conditional is False)

The goal in the false branch of the conditional is solvable by using `leS`, leaving us with the subgoal `LE (div (minus n (pred m)) m) (suc n)`. We name the inner subtraction-expression `n2`, and the outer division-expression `n3`. We solve the subgoal (`LE n3 n1`) using transitivity (`leTrans`) by showing that `n3 ≤ n2 ≤ n1`, leaving us with two final subsubgoals. The second subsubgoal (`LE n2 n1`) is provable (`n2LEn1`) using our lemma about subtraction getting smaller or staying the same (`leMinus`).

Finally, the first subsubgoal (`LE n3 n2`) is the interesting case (`n3LEn2`). First, we rewrite using `preq` to change our goal from requiring the concrete natural number predecessor (`outCV -nfimap`) to instead require the abstract predecessor (`pr`). We can use `ih` to get an inductive hypothesis, of type `LE (div (elimId -c r) m) (elimId -c r)`, for any *abstract*

---

[4] To make the proof easier to read, we use non-depenent `if` and `case` statements. Our actual code requires the dependent eliminator counterparts of these statements, along with the appropriate motives.

natural number `r`. Thus, our final subsubgoal is solvable by the inductive hypothesis where
`r` is the result of the abstract subtraction `minus' pr r (pred m)`. Because `elimId -c`
is definitionally equal to the identity function, and because we already rewrote by `preq`,
this gives us exactly what we want. This is despite the fact that our original subsubgoal
(`LE n3 n2`) is stated in terms of `n3` and `n2`, which use concrete `div` and `minus`, respectively!
Importantly, the inductive hypothesis we use requires course-of-value induction, obtained
by an expression that iterates the predecessor function (`ih (minus' pr r (pred m))`). In
contrast, ordinary induction corresponds to using the inductive hypothesis `ih r`.

5.3   Catalan Numbers

Many solutions to counting problems in combinatorics can be given in terms of Catalan
numbers. The Catalan numbers are definable as the solution to the recurrence $C_0 = 1$ and
$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$. This translates to an intuitive functional definition of the Catalan
numbers:

```
cat : Nat → Nat
cat 0 = 1
cat (suc n) = sum (λ i → cat i * cat (n - i)) n
```

The `sum` function has type $(\texttt{Nat} \rightarrow \texttt{Nat}) \rightarrow \texttt{Nat} \rightarrow \texttt{Nat}$, where the lower bound of the
sum is always zero (`i=0`), the second argument is the upper bound of the sum (`n`), and
the first argument is the body of the sum (parameterized by `i`). Once again, this is not a
structurally terminating function recognizable by Agda. While `fib` and `div` have a static
number of course-of-value recursions (two and one, respectively), the number of recursions
made by `cat` is determined by its input. Nonetheless, we are able to define `cat` using our
development.

```
cat ◄ NatCV → NatCV
  = inductionCV (Λ R. Λ c. λ pf. Λ pfeq. λ ih . λ nf.
  case nf
    (λ _. suc zero)  % cat 0
    (λ r. sum        % cat (suc n)
      (λ i. mult
        (ih (minus' pf r (minus (elimId -c r) i)))
        (ih (minus' pf r i)))
      (elimId -c r))).
```

As with `div`, above `r` has abstract type `R`, so we convert it to a `NatCV` where necessary
by applying `elimId -c`. The intuitive right factor `cat (n - i)` is directly encoded as
`ih (minus' pf r i))`. However, we cannot directly encode the intuitive left factor `cat i`,
because `i` is a natural number and we only have inductive hypotheses for values of abstract
type `R`. However, `i` is equivalent to `n - (n - i)` for all `i` where `i ≤ n`. We use the abstract
`minus'` function for the outer subtraction, whose first numeric argument is an abstract `r`
but whose second numeric argument expects a concrete `NatCV`. Hence, the inner subtraction
is a concrete `minus`, whose first argument is the concrete version of `r` (converted via identity
function `c`) and whose second argument is the concrete `i` (of type `NatCV`). Because the
outer subtraction (`minus'`) returns an abstract `R`, we can get an inductive hypothesis for
an expression equivalent to `i`. Finally, we can prove the aforementioned equivalence for
`minus` (by rewriting), and as a consequence the defining equation (for the successor case) of

the intuitive definition of Catalan numbers (by dependent case-analysis and rewriting by `minusId`):

```
minusId ◀ Π n i : NatCV. (i ≤ n) ≃ tt → minus n (minus n i) ≃ i = <..>
catSuc ◀ Π n : NatCV.
  cat (suc n) ≃ sum (λ i. mult (cat i) (cat (minus n i))) n = <..>
```

Once again, the discrepancy between abstract `minus'` and concrete `minus` is resolved in the proofs thanks to the computational behavior of `inductionCV` instantiating `pr` to `outCV`.

6 Conclusions and Related Work

Ahn et al. [2] describe a hierarchy of Mendler-style recursion combinators. They implement generic course-of-value recursion in terms of Haskell's general recursion. Then, they prove that course-of-value recursion for arbitrary "negative" inductive datatypes implies non-termination.

Miranda-Perea [11] describes extensions of System F with primitive course-of-value iteration schemes. He explains that the resulting systems lose strong normalization if they are combined with negative datatypes.

Uustalu et al. [18] show that natural-deduction proof systems for intuitionistic logics can be safely *extended* with a course-of-value induction operator in a proof-theoretically defensible way. Their requirement of monotonicity corresponds to our notion of `IdMapping`.

In contrast to the work described above, we have now shown that course-of-value induction can be *derived* within type theory (specifically, within CDLE). We will end with comparing our approach to alternative approaches to handling complex termination arguments within type theory. [5]

6.1 The Below Way

Goguen et al. [5] define the induction principle `recNat` (generalizing to all inductive types), which they use to elaborate dependent pattern matching to eliminators. In the step case, `recNat` receives `BelowNat P n`, which is a large tuple consisting of the motive `P` for every predecessor of `n`. Simple functions performing nested pattern matching (e.g., `fib`) can be written using `recNat` and nested case-analysis, by projecting out inductive hypotheses from `BelowNat P n`. However, functions with more complex termination arguments (e.g., `div` and `cat`) require proving extra lemmas (e.g., `recMinus` in our accompanying code) to dynamically extract inductive hypotheses from `BelowNat P n` evidence. In our approach, such lemmas are unnecessary.

6.2 Sized Types

Abel [1] extends type theory with a notion of *sized types*, which allows intuitive function definitions to be accepted by termination checking. Course-of-value induction (CoVI) and sized types (ST) have trade-offs. ST requires defining size-indexed versions of the datatypes, which necessitates altering conventional type signatures of functions to include size information. While CoVI is derivable within CDLE, ST extends the underlying type theory. On the other hand, CoVI is restricted to functions that recurse strictly on previous values. Hence, a function like merge sort can be written using ST but not with CoVI. As future work, we

---

[5] For comparison, our accompanying code includes Agda formalizations of `fib`, `div`, and `cat` in the "Below" style of Section 6.1 and the sized types style of Section 6.2.

would like to investigate datatype encodings with a restricted version of abstract constructors (in addition to abstract destructors) for defining functions like merge sort.

─── **References** ───

1 Andreas Abel. MiniAgda: Integrating sized and dependent types. In *Workshop on Partiality And Recursion in Interactive Theorem Provers (PAR)*, July 2010.

2 Ki Yung Ahn and Tim Sheard. A hierarchy of mendler style recursion combinators: Taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 234–246, New York, NY, USA, 2011. ACM.

3 Denis Firsov, Richard Blair, and Aaron Stump. Efficient mendler-style lambda-encodings in cedille. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 235–252, 2018.

4 Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in cedille. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 215–227, New York, NY, USA, 2018. ACM.

5 Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, pages 521–540. Springer, 2006.

6 Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, July 1999.

7 Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *18th IEEE Symposium on Logic in Computer Science (LICS)*, pages 86–95, 2003.

8 Daniel Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 460–469. IEEE Computer Society, 1983.

9 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2016. Version 8.5. URL: `http://coq.inria.fr`.

10 Alexandre Miquel. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications (TLCA)*, pages 344–359. 2001.

11 Favio Ezequiel Miranda-Perea. Some remarks on type systems for course-of-value recursion. *Electronic Notes in Theoretical Computer Science*, 247:103 – 121, 2009. Proceedings of the Third Workshop on Logical and Semantic Frameworks with Applications (LSFA 2008).

12 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

13 Aaron Stump. The Calculus of Dependent Lambda Eliminations. *Journal of Functional Programming*, 27:e14.

14 Aaron Stump. From Realizability to Induction via Dependent Intersection, 2017. Under consideration for Annals of Pure and Applied Logic.

15 Aaron Stump. From Realizability to Induction via Dependent Intersection. *Ann. Pure Appl. Logic*, 2018. to appear.

16 Aaron Stump. Syntax and Semantics of Cedille. 2018. `arXiv:1806.04709`.

17 Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic J. of Computing*, 6(3):343–361, September 1999.

18 Tarmo Uustalu and Varmo Vene. Least and greatest fixed points in intuitionistic natural deduction. *Theoretical Computer Science*, 272(1):315 – 339, 2002. Theories of Types and Proofs 1997.