Normalization by Evaluation in the Delay Monad

A Case Study for Coinduction via Copatterns and Sized Types

Andreas Abel
Department of Computer Science and Engineering
Chalmers University of Technology
Sweden
abela@chalmers.se

James Chapman
Institute of Cybernetics
Tallinn University of Technology
Estonia
james@cs.ioc.ee

In this paper we present an Agda formalization of a normalizer for simply-typed lambda terms. The first step is to write a coinductive evaluator using the delay monad. The other component of the normalizer, a type-directed reifier from values to η-long β-normal forms, resides in the delay monad as well. Their composition, normalization-by-evaluation, is shown to be a total function, using a standard logical-relations argument.

The successful formalization serves as a proof-of-concept for coinductive programming and proving using sized types and copatterns, a new and presently experimental feature of Agda.

1 Introduction and Related Work

It would be a great shame if dependently-typed programming (DTP) restricted us to only writing very clever programs that were a priori structurally recursive and hence obviously terminating. Put another way, it is a lot to ask of the programmer to provide the program and its termination proof in one go, programmers should also be supported in working step-by-step. This paper champions a technique that lowers the barrier of entry, from showing termination to only showing productivity up front, and then later providing the opportunity to show termination (convergence). In this paper, we write a simple recursive normalizer for simply-typed lambda calculus which as an intermediate step constructs first-order weak head normal forms and finally constructs full η-long β-normal forms. The normalizer is not structurally recursive and we represent it in Agda as a potentially non-terminating but nonetheless productive corecursive function targeting the coinductive delay monad. Later we show that the function is indeed terminating as all such delayed computations converge (are only finitely delayed) by a quite traditional strong computability argument. The coinductive normalizer, when combined with the termination proof, yields a terminating function returning undelayed normal forms.

Our normalizer is an instance of *normalization by evaluation* as conceived by Danvy [Dan99] and Abel, Coquand, and Dybjer [ACD08]:[1] Terms are first evaluated into an applicative structure of values; herein, we realize function values by closures, which can be seen as weak head normal forms under explicit substitution. The second phase goes in the other direction: values are *read back* (terminology by Grégoire and Leroy [GL02]) as terms in normal form. In contrast to the cited works, we employ intrinsically well-typed representations of terms and values. In fact, our approach is closest to Altenkirch and Chapman's *big-step normalization* [AC09, Cha09]; this work can be consulted for more detailed descriptions of well-typed terms and values. Where Altenkirch and Chapman represent partial functions via their inductively defined graphs, we take the more direct route via the coinductive delay monad. This is the essential difference and contribution of the present work.

[1] In a more strict terminology, normalization by evaluation must evaluate object-level functions as meta-level functions; such is happening in Berger and Schwichtenberg's original work [BS91], but not here.

Submitted to:
MSFP 2014

© A. Abel & J. Chapman
This work is licensed under the
Creative Commons Attribution License.

The delay monad has been used implement evaluators before: Danielsson's *Operational Semantics Using the Partiality Monad* [Dan12] for untyped lambda-terms is the model for our evaluator. However, we use a *sized* delay monad, which allows us to use the bind operation of the monad directly; Danielsson, working with the previous version of Agda and its coinduction, has to use a workaround to please Agda's guardedness checker.

In spirit, evaluation into the delay monad is closely related to *continuous normalization* as implemented by Aehlig and Joachimski [AJ05]. Since they compute possibly infinitely deep normal forms (from untyped lambda-terms), their type of terms is coinductive; further, our `later` constructor of the delay monad is one of their constructors of lambda-terms, called *repetition constructor*. They attribute this idea to Mints [Min78]. In the type-theoretic community, the delay monad has been popularized by Capretta [Cap05].

Using hereditary substitutions [WCPW03], a normalization function for the simply-typed lambda calculus can be defined directly, by structural recursion on types. This normalizer has been formalized in Agda by Altenkirch and Keller [KA10]. The idea of normalization by induction on types is very old, see, e.g., Prawitz [Pra65]. Note however, that normalization via hereditary substitution implements a fixed strategy, bottom-up normalization, which cannot be changed without losing the inductive structure of the algorithm. Our strategy, normalization via closures, cannot be implemented directly by induction on types. Further, the simple induction on types also breaks down when switching to more powerful lambda-calculi like Gödel's T, while out approach scales without effort.

To save paper and preserve precious forests, we have only included the essential parts of the Agda development; the full code is available online under `http://www.tcs.ifi.lmu.de/~abel/msfp14.lagda`. It runs under the development version of Agda from December 2013 or later plus the latest standard library.

2 Delay Monad

The `Delay` type is used to represent computations of type `A` whose result will be returned after a potentially infinite delay. A value available immediately is wrapped in the `now` constructor. A delayed value is wrapped in at least one `later` constructor. Each `later` represents a single delay and an infinite number of `later` constructors wrapping a value represents an infinite delay, i.e., a non-terminating computation.

It is interesting to compare the `Delay` type with the `Maybe` type familiar from Haskell. Both are used to represent partial values, but differ in the nature of partiality. Pattern matching on an element of the `Maybe` type immediately yields either success (returning a value) or failure (returning no value) whereas pattern matching on an element of the `Delay` type either yields success (returning a value) or a delay after which one can pattern match again. While `Maybe` lets us represent computation with error, possible *non-termination* is elegantly modeled by the `Delay` type. A definitely non-terminating value is represented by an infinite succession of `later` constructors, thus, `Delay` must be a coinductive type. When analyzing a delayed value, we never know whether after an initial succession of `later` constructors we will finally get a `now` with a proper value—this reflects the undecidability of termination in general.

In Agda, the `Delay` type can be represented as a mutual definition of an inductive datatype and a coinductive record. The record `∞Delay` is a coalgebra and one interacts with it by using its single observation (copattern) `force`. Once forced we get an element of the `Delay` datatype which we can pattern match on to see if the value is available `now` or `later`. If it is `later` then we get a element of `∞Delay` which we can `force` again, etc.

```
mutual
```

```
  data Delay (i : Size) (A : Set) : Set where
    now   : A           → Delay i A
    later : ∞Delay i A → Delay i A

  record ∞Delay (i : Size) (A : Set) : Set where
    coinductive
    field
      force : {j : Size< i} → Delay j A
```

Both types (`Delay` and `∞Delay`) are indexed by a size `i`. This should be understood as *observation depth*, i.e., at least the number of times we can iteratively `force` the delayed computation. More precisely, forcing `a∞ : ∞Delay i A` will result in a value `force a∞ : Delay j A` of strictly smaller observation depth `j < i`. An exception is a delayed value `a∞ : ∞Delay ∞ A` of infinite observation depth, whose forcing `force a∞ : Delay ∞ A` again has infinite observation depth. The sizes (observation depths) are merely a means to establish productivity of recursive definitions, in the end, we are only interested in values `a? : Delay ∞ A` of infinite depth.

If a corecursive function into `Delay i A` only calls itself at smaller depths `j < i` it is guaranteed to be *productive*, i.e., well-defined. In the following definition of the non-terminating value `never`, we make the size arguments explicit to demonstrate how they ensure productivity:

```
never : ∀{i A} → ∞Delay i A
force (never {i}) {j} = later (never {j})
```

The value `never` is defined to be the thing that, if forced, returns a postponed version of itself. Formally, we have defined a member of the record type `∞Delay i A` by giving the contents of all of its fields, here only `force`. The use of a projection like `force` on the left hand side of a defining equation is called a *copattern* [APTS13]. Corecursive definitions by copatterns are the latest addition to Agda, and can be activated since version 2.3.2 via the flag `--copatterns`. The use of copatterns reduces productivity checking to termination checking. Agda simply checks that the size argument `j` given in the recursive call to `never` is smaller than the original function parameter `i`. Indeed, `j < i` is ensured by the typing of projection `force`. A more detailed explanation and theoretical foundations can be found in previous work of the first author [AP13].

At each observation depth `i`, the functor `Delay i` forms a monad. The `return` of the monad is given by `now`, and bind `_>>=_` is implemented below. Notice that bind is size (observation depth) preserving; in other words, its modulus of continuity is the identity. The number of safe observations on `a? >>= f` is no less than those on both `a?` and `f a` for any `a`. The implementation of bind follows a common scheme when working with `Delay`: we define two mutually recursive functions, the first by pattern matching on `Delay` and the second by copattern matching on `∞Delay`.

```
module Bind where
  mutual
    _>>=_ : ∀ {i A B} → Delay i A → (A → Delay i B) → Delay i B
    now   a  >>= f = f a
    later a∞ >>= f = later (a∞ ∞>>= f)

    _∞>>=_ : ∀ {i A B} → ∞Delay i A → (A → Delay i B) → ∞Delay i B
    force (a∞ ∞>>= f) = force a∞ >>= f
```

We make `Delay i` an instance of `RawMonad` (it is called ‘raw’ as it does not enforce the laws) as defined in the Agda standard library. This provides us automatically with a `RawFunctor` instance, with map function `_<$>_` written infix as in Haskell’s base library.

```
delayMonad : ∀ {i} → RawMonad {f = lzero} (Delay i)
delayMonad {i} = record
  { return = now
  ; _>>=_  = _>>=_ {i}
  } where open Bind
```

2.1 Strong Bisimilarity

We can define the coinductive strong bisimilarity relation `_~_` for `Delay ∞ A` following the same pattern as for `Delay` itself. Two finite computations are *strongly bisimilar* if they contain the same value and the same amount of delay (number of `later`s). Non-terminating computations are also identified.[2]

```
mutual
  data _~_ {i : Size} {A : Set} : (a? b? : Delay ∞ A) → Set where
    ~now   : ∀ a → now a ~ now a
    ~later : ∀ {a∞ b∞} (eq : a∞ ∞~⟨ i ⟩~ b∞) → later a∞ ~ later b∞

  _~⟨_⟩~_ = λ {A} a? i b? → _~_ {i}{A} a? b?

  record _∞~⟨_⟩~_ {A} (a∞ : ∞Delay ∞ A) i (b∞ : ∞Delay ∞ A) : Set where
    coinductive
    field
      ~force : {j : Size< i} → force a∞ ~⟨ j ⟩~ force b∞

_∞~_ = λ {i} {A} a∞ b∞ → _∞~⟨_⟩~_ {A} a∞ i b∞
```

The definition includes the two sized relations `_~⟨ i ⟩~` on `Delay ∞ A` and `_∞~⟨ i ⟩~` on `∞Delay ∞ A` that exist for the purpose of recursively constructing derivations (proofs) of bisimilarity in a way that convinces Agda of their productivity. These are approximations of bisimilarity in the sense that they are intermediate, partially defined relations needed for the construction of the fully defined relations `_~⟨ ∞ ⟩~` and `_∞~⟨ ∞ ⟩~`. They are subtly different to the approximations $\cong_n$ of strong bisimilarity $\cong$ in the context of ultrametric spaces [AJ05, Sec. 2.2]. These approximations are fully defined relations that approximate the concept of equality, for instance at stage $n = 0$ all values are equal, at $n = 1$ they are equal if observations of depth one coincide, until at stage $n = \omega$ observation of arbitrary depth must yield the same result.

All bisimilarity relations `_~⟨ i ⟩~` and `_∞~⟨ i ⟩~` are equivalences. The proofs by coinduction are straightforward and omitted here.

```
~refl  : ∀{i A}(a? : Delay ∞ A)  → a? ~⟨ i ⟩~ a?
∞~refl : ∀{i A}(a∞ : ∞Delay ∞ A) → a∞ ∞~⟨ i ⟩~ a∞

~sym :  ∀{i A}{a? b? : Delay ∞ A}  → a? ~⟨ i ⟩~ b? → b? ~⟨ i ⟩~ a?
```

[2]One could also consider other relations such as *weak bisimilarity* which identifies finite computations containing the same value but different numbers of `later`s.

```
∞~sym : ∀{i A}{a? b? : ∞Delay ∞ A} → a? ∞~⟨ i ⟩~ b? → b? ∞~⟨ i ⟩~ a?

~trans : ∀{i A}{a? b? c? : Delay ∞ A} →
         a? ~⟨ i ⟩~ b? →  b? ~⟨ i ⟩~ c? → a? ~⟨ i ⟩~ c?
∞~trans : ∀{i A}{a∞ b∞ c∞ : ∞Delay ∞ A} →
          a∞ ∞~⟨ i ⟩~ b∞ →  b∞ ∞~⟨ i ⟩~ c∞ → a∞ ∞~⟨ i ⟩~ c∞
```

The associativity law of the delay monad holds up to strong bisimilarity. Here, we spell out the proof by coinduction:

```
mutual
  bind-assoc : ∀{i A B C} (m : Delay ∞ A)
               {k : A → Delay ∞ B} {l : B → Delay ∞ C} →
               ((m >>= k) >>= l) ~⟨ i ⟩~ (m >>= λ a → (k a >>= l))
  bind-assoc (now a)    = ~refl _
  bind-assoc (later a∞) = ~later (∞bind-assoc a∞)

  ∞bind-assoc : ∀{i A B C} (a∞ : ∞Delay ∞ A)
                {k : A → Delay ∞ B} {l : B → Delay ∞ C} →
                ((a∞ ∞>>= k) ∞>>= l) ∞~⟨ i ⟩~ (a∞ ∞>>= λ a → (k a >>= l))
  ~force (∞bind-assoc a∞) = bind-assoc (force a∞)
```

Further, bind (_>>=_ and _∞>>=_) and is a congruence in both arguments (proofs omitted here).

```
bind-cong-l  : ∀{i A B}{a? b? : Delay ∞ A} →  a? ~⟨ i ⟩~ b? →
               (k : A → Delay ∞ B) → (a? >>= k) ~⟨ i ⟩~ (b? >>= k)
∞bind-cong-l : ∀{i A B}{a∞ b∞ : ∞Delay ∞ A} → a∞ ∞~⟨ i ⟩~ b∞ →
               (k : A → Delay ∞ B) → (a∞ ∞>>= k) ∞~⟨ i ⟩~ (b∞ ∞>>= k)
bind-cong-r  : ∀{i A B}(a? : Delay ∞ A){k l : A → Delay ∞ B} →
               (∀ a → (k a) ~⟨ i ⟩~ (l a)) → (a? >>= k) ~⟨ i ⟩~ (a? >>= l)
∞bind-cong-r : ∀{i A B}(a∞ : ∞Delay ∞ A){k l : A → Delay ∞ B} →
               (∀ a → (k a) ~⟨ i ⟩~ (l a)) → (a∞ ∞>>= k) ∞~⟨ i ⟩~ (a∞ ∞>>= l)
```

As map (_<$>_) is defined in terms of bind and return, laws for map are instances of the monad laws:

```
map-compose : ∀{i A B C} (a? : Delay ∞ A) {f : A → B} {g : B → C} →
  (g <$> (f <$> a?)) ~⟨ i ⟩~ ((g ∘ f) <$> a?)
map-compose a? = bind-assoc a?

map-cong : ∀{i A B}{a? b? : Delay ∞ A} (f : A → B) →
  a? ~⟨ i ⟩~ b? → (f <$> a?) ~⟨ i ⟩~ (f <$> b?)
map-cong f eq = bind-cong-l eq (now ∘ f)
```

2.2 Convergence

We define convergence as a relation between delayed computations of type `Delay ∞ A` and values of type `A`. If `a? ⇓ a`, then the delayed computation `a?` eventually yields the value `a`. This is a central concept in this paper as we will write a (productive) normalizer that produces delayed normal forms and then prove that all such delayed normal forms converge to a value yielding termination of the normalizer. Notice that convergence is an *inductive* relation defined on coinductive data.

```
data _⇓_ {A : Set} : (a? : Delay ∞ A) (a : A) → Set where
  now⇓   : ∀ {a} → now a ⇓ a
  later⇓ : ∀ {a} {a∞ : ∞Delay ∞ A} → force a∞ ⇓ a → later a∞ ⇓ a

_⇓ : {A : Set} (x : Delay ∞ A) → Set
x ⇓ = ∃ λ a → x ⇓ a
```

We define some useful utilities about convergence: We can map functions on values over a convergence relation (see `map⇓`). If a delayed computation `a?` converges to a value `a` then so does any strongly bisimilar computation `a?′` (see `subst~⇓`). If we apply a function `f` to a delayed value `a?` using bind and we know that the delayed value converges to a value `a` then we can replace the bind with an ordinary application `f a` (see `bind⇓`).

```
map⇓ : ∀ {A B} {a : A} {a? : Delay ∞ A}
  (f : A → B) (a⇓ : a? ⇓ a) → (f <$> a?) ⇓ f a

subst~⇓ : ∀{A}{a? a?′ : Delay ∞ A}{a : A} → a? ⇓ a → a? ~ a?′ → a?′ ⇓ a

bind⇓ : ∀{A B}(f : A → Delay ∞ B)
       {?a : Delay ∞ A}{a : A} → ?a ⇓ a →
       {b : B} → f a ⇓ b → (?a >>= f) ⇓ b
```

That completes our discussion the the delay infrastructure.

3 Well-typed terms, values, and coinductive normalization

We present the syntax of the well-typed lambda terms, which is Altenkirch and Chapman's [AC09] without explicit substitutions. First we introduce simple types `Ty` with one base type $\star$ and function types `a ⇒ b`.

```
data Ty : Set where
  ⋆   : Ty
  _⇒_ : (a b : Ty) → Ty
```

We use de Bruijn indices to represent variables, so contexts `Cxt` are just lists of (unnamed) types.

```
data Cxt : Set where
  ε   : Cxt
  _,_ : (Γ : Cxt) (a : Ty) → Cxt
```

Variables are de Bruijn indices, just natural numbers. They are indexed by context and type which guarantees that they are well-scoped and well-typed. Notice only non-empty contexts can have variables, notice that neither constructor targets the empty context. The zero variable as the same type as the type at the end of the context.

```
data Var : (Γ : Cxt) (a : Ty) → Set where
  zero : ∀ {Γ a}                 → Var (Γ , a) a
  suc  : ∀ {Γ a b} (x : Var Γ a) → Var (Γ , b) a
```

Terms are are also indexed by context and type, guaranteeing well-typedness and well-scopedness. Terms are either variables, lambda abstractions or applications. Notice that the context index in the body of the lambda tracks that one more variable has been bound and that applications are guaranteed to be well-typed.

```
data Tm (Γ : Cxt) : (a : Ty) → Set where
  var : ∀ {a}   (x : Var Γ a)                      → Tm Γ a
  abs : ∀ {a b} (t : Tm (Γ , a) b)                 → Tm Γ (a ⇒ b)
  app : ∀ {a b} (t : Tm Γ (a ⇒ b)) (u : Tm Γ a) → Tm Γ b
```

We introduce neutral terms, parametric in the argument type of application as we will need both neutral weak-head normal and beta-eta normal forms. Intuitively neutrals are *stuck*. In plain lambda-calculus, they are either variables or applications that cannot compute as there is a neutral term in the function position.

```
data Ne (T : Cxt → Ty → Set)(Γ : Cxt) : Ty → Set where
  var : ∀{a} → Var Γ a → Ne T Γ a
  app : ∀{a b} → Ne T Γ (a ⇒ b) → T Γ a → Ne T Γ b
```

Weak head normal forms (`Values`) are either neutral terms or closures of a body of a lambda and an environment containing values for the all the variables except the lambda bound variable. Once a value for the lambda bound variable is available the body of the lambda may be evaluated in the now complete environment. `Values` are defined mutually with `Environments` which are just lists of values. We also provide a `lookup` function for looking up variables in the environment. Notice that the types ensure that the lookup function never tries to lookup a variable that is out of range and, indeed, never encounters an empty environment as no variables can exist there.

```
mutual
  data Val (Δ : Cxt) : (a : Ty) → Set where
    ne  : ∀{a}     (w : Ne Val Δ a)                 → Val Δ a
    lam : ∀{Γ a b} (t : Tm (Γ , a) b) (ρ : Env Δ Γ) → Val Δ (a ⇒ b)

  data Env (Δ : Cxt) : (Γ : Cxt) → Set where
    ε   : Env Δ ε
    _,_ : ∀ {Γ a} (ρ : Env Δ Γ) (v : Val Δ a) → Env Δ (Γ , a)

lookup : ∀ {Γ Δ a} → Var Γ a → Env Δ Γ → Val Δ a
lookup zero    (ρ , v) = v
lookup (suc x) (ρ , v) = lookup x ρ
```

Evaluation `eval` takes a term and a suitable environment and returns a delayed value. It is defined mutually with an `apply` function that applies function values to argument values, returning a delayed result value.

```
mutual
  eval : ∀{i Γ Δ a} → Tm Γ a → Env Δ Γ → Delay i (Val Δ a)
  eval (var x)   ρ = now (lookup x ρ)
  eval (abs t)   ρ = now (lam t ρ)
  eval (app t u) ρ = eval t ρ >>= λ f → eval u ρ >>= apply f
  apply : ∀ {i Δ a b} → Val Δ (a ⇒ b) → Val Δ a → Delay i (Val Δ b)
  apply (ne w)    v = now (ne (app w v))
  apply (lam t ρ) v = later (beta t ρ v)

  beta : ∀ {i Γ a b} (t : Tm (Γ , a) b)
    {Δ : Cxt} (ρ : Env Δ Γ) (v : Val Δ a) → ∞Delay i (Val Δ b)
  force (beta t ρ v) = eval t (ρ , v)
```

Beta-eta normal forms are either of function type, in which case they must be a lambda term, or of base type, in which case they must be a neutral term, meaning, a variable applied to normal forms.

```
data Nf (Γ : Cxt) : Ty → Set where
  lam : ∀{a b}(n : Nf (Γ , a) b) → Nf Γ (a ⇒ b)
  ne  : (m : Ne Nf Γ ⋆) → Nf Γ ⋆
```

To turn values into normal forms we must be able to apply functional values to fresh variables. We need an operation on values that introduces a fresh variable into the context:

```
weakVal : ∀ {Δ a c} → Val Δ c → Val (Δ , a) c
```

We take the approach of implementing this operations using so-called order preserving embeddings (OPEs) which represent weakenings in arbitrary positions in the context. Order preserving embeddings can be represented in a first order way which simplifies reasoning about them.

```
data _≤_ : (Γ Δ : Cxt) → Set where
  id   : ∀ {Γ} → Γ ≤ Γ
  weak : ∀ {Γ Δ a} → Γ ≤ Δ → (Γ , a) ≤ Δ
  lift : ∀ {Γ Δ a} → Γ ≤ Δ → (Γ , a) ≤ (Δ , a)
```

We implement composition of OPEs and prove that `id` is the right unit of composition (proof suppressed). The left unit property holds definitionally. We could additionally prove associativity and observe that OPEs form a category but this is not required in this paper.

```
_•_ : ∀ {Γ Δ Δ′} (η : Γ ≤ Δ) (η′ : Δ ≤ Δ′) → Γ ≤ Δ′
id     • η′      = η′
weak η • η′      = weak (η • η′)
lift η • id      = lift η
lift η • weak η′ = weak (η • η′)
lift η • lift η′ = lift (η • η′)

η•id :  ∀ {Γ Δ} (η : Γ ≤ Δ) → η • id ≡ η
```

We define a map operation that weakens variables, values, environments, normal forms and neutral terms by OPEs.

```
var≤ : ∀{Γ Δ} → Γ ≤ Δ → ∀{a} → Var Δ a → Var Γ a
val≤ : ∀{Γ Δ} → Γ ≤ Δ → ∀{a} → Val Δ a → Val Γ a
env≤ : ∀{Γ Δ} → Γ ≤ Δ → ∀{E} → Env Δ E → Env Γ E
nev≤ : ∀{Γ Δ} → Γ ≤ Δ → ∀{a} → Ne Val Δ a → Ne Val Γ a
nf≤  : ∀{Γ Δ} → Γ ≤ Δ → ∀{a} → Nf Δ a → Nf Γ a
nen≤ : ∀{Γ Δ} → Γ ≤ Δ → ∀{a} → Ne Nf Δ a → Ne Nf Γ a
```

Having defined weakening of values by OPEs, defining the simplest form of weakening `weakVal` that just introduces a fresh variable into the context is easy to define:

```
wk : ∀{Γ a} → (Γ , a) ≤ Γ
wk = weak id

weakVal = val≤ wk
```

We can now define a function `readback` that turns values into delayed normal forms, the potential delay is due to the call to the `apply` function. The readback function calls its utility function `∞readback` which is defined by induction on the type and copattern matching. If the value is at base type then a call to `nereadback` is made which just proceeds structurally through the neutral term replacing values in the argument positions by normal forms. If the value is at function type then the function is weakened and applied to a fresh variable, yielding a delayed value at range type, then a lambda constructor is applied.

```
mutual
  readback : ∀{i Γ a} → Val Γ a → Delay i (Nf Γ a)
  readback {a = ⋆} (ne w) = ne  <$> nereadback w
  readback {a = _ ⇒ _} v = lam <$> later (eta v)

  eta : ∀{i Γ a b} → Val Γ (a ⇒ b) → ∞Delay i (Nf (Γ , a) b)
  force (eta v) = readback =<< apply (weakVal v) (ne (var zero))

  nereadback : ∀{i Γ a} → Ne Val Γ a → Delay i (Ne Nf Γ a)
  nereadback (var x)   = now (var x)
  nereadback (app w v) =
    nereadback w >>= λ m → app m <$> readback v
```

We define the identity environment by induction on the context.

```
ide : ∀ Γ → Env Γ Γ
ide ε = ε
ide (Γ , a) = env≤ wk (ide Γ) , ne (var zero)
```

Given `eval`, `ide` and `readback` we can define a normalization function `nf` that for any term returns a delayed normal form.

```
nf : ∀{Γ a}(t : Tm Γ a) → Delay ∞ (Nf Γ a)
nf {Γ} t = eval t (ide Γ) >>= readback
```

4 Termination proof

We define a logical predicate `V⟦_⟧_`, corresponding to strong computability on values. It is defined by induction on the type of the value. At base type when the value must be neutral, then the relation states that the neutral term is strongly computable if its readback converges. At function type it states that the function is strongly computable if, in any weakened context (in the general OPE sense) it takes any value which is strongly computable to a delayed value which converges to a strongly computable value. The predicate `C⟦_⟧_` on delayed values is shorthand for a triple of a value, a proof that the delayed value converges to the value and a proof of strong computability.

```
mutual
  V⟦_⟧_ : ∀{Γ}(a : Ty) → Val Γ a → Set
  V⟦ ⋆     ⟧ (ne w) = nereadback w ⇓
  V⟦ a ⇒ b ⟧ f      = ∀{Δ}(η : Δ ≤ _)(u : Val Δ a)
    (⟦u⟧ : V⟦ a ⟧ u) → C⟦ b ⟧ (apply (val≤ η f) u)

  C⟦_⟧_ : ∀{Γ}(a : Ty) → Delay ∞ (Val Γ a) → Set
  C⟦ a ⟧ v? = ∃ λ v → v? ⇓ v × V⟦ a ⟧ v
```

The notion of strongly computable value is easily extended to environments.

```
E⟦_⟧_ : ∀{Δ}(Γ : Cxt) → Env Δ Γ → Set
E⟦ ε ⟧     ε       = ⊤
E⟦ Γ , a ⟧ (ρ , v) = E⟦ Γ ⟧ ρ × V⟦ a ⟧ v
```

Later we will require weakening (applying an OPE) variables, values, environments, etc. preserve identity and composition (respect functor laws). We state these properties now but suppress the proofs.

```
val≤-id : ∀ {Δ a} (v : Val Δ a)    → val≤ id v ≡ v
env≤-id : ∀ {Γ Δ} (ρ : Env Δ Γ)    → env≤ id ρ ≡ ρ
nev≤-id : ∀ {Δ a} (t : Ne Val Δ a) → nev≤ id t ≡ t
```

```
var≤-• : ∀ {Γ₁ Γ₂ Γ₃ a} (η : Γ₁ ≤ Γ₂) (η′ : Γ₂ ≤ Γ₃) (x : Var Γ₃ a) →
  var≤ η (var≤ η′ x) ≡ var≤ (η • η′) x
val≤-• : ∀ {Δ₁ Δ₂ Δ₃ a} (η : Δ₁ ≤ Δ₂) (η′ : Δ₂ ≤ Δ₃) (v : Val Δ₃ a) →
         val≤ η (val≤ η′ v) ≡ val≤ (η • η′) v
env≤-• : ∀ {Γ Δ₁ Δ₂ Δ₃} (η : Δ₁ ≤ Δ₂) (η′ : Δ₂ ≤ Δ₃) (ρ : Env Δ₃ Γ) →
         env≤ η (env≤ η′ ρ) ≡ env≤ (η • η′) ρ
nev≤-• : ∀ {Δ₁ Δ₂ Δ₃ a} (η : Δ₁ ≤ Δ₂) (η′ : Δ₂ ≤ Δ₃) (t : Ne Val Δ₃ a) →
         nev≤ η (nev≤ η′ t) ≡ nev≤ (η • η′) t
```

We also require that the operations that we introduce such as lookup, eval, apply, readback etc. commute with weakening. We, again, state these necessary properties but suppress the proofs.

```
lookup≤  : ∀ {Γ Δ Δ′ a} (x : Var Γ a) (ρ : Env Δ Γ) (η : Δ′ ≤ Δ) →
           val≤ η (lookup x ρ) ≡ lookup x (env≤ η ρ)
eval≤    : ∀ {i Γ Δ Δ′ a} (t : Tm Γ a) (ρ : Env Δ Γ) (η : Δ′ ≤ Δ) →
           (val≤ η <$> (eval t ρ)) ~⟨ i ⟩~ (eval t (env≤ η ρ))
apply≤   : ∀{i Γ Δ a b} (f : Val Γ (a ⇒ b))(v : Val Γ a)(η : Δ ≤ Γ) →
           (val≤ η <$> apply f v) ~⟨ i ⟩~ (apply (val≤ η f) (val≤ η v))
beta≤ : ∀ {i Γ Δ E a b} (t : Tm (Γ , a) b)(ρ : Env Δ Γ) (v : Val Δ a) →
        (η : E ≤ Δ) →
        (val≤ η ∞<$> (beta t ρ v)) ∞~⟨ i ⟩~ beta t (env≤ η ρ) (val≤ η v)


nereadback≤ : ∀{i Γ Δ a}(η : Δ ≤ Γ)(t : Ne Val Γ a) →
              (nen≤ η <$> nereadback t) ~⟨ i ⟩~ (nereadback (nev≤ η t))
readback≤   : ∀{i Γ Δ} a (η : Δ ≤ Γ)(v : Val Γ a) →
              (nf≤ η <$> readback v) ~⟨ i ⟩~ (readback (val≤ η v))
eta≤  : ∀{i Γ Δ a b} (η : Δ ≤ Γ)(v : Val Γ (a ⇒ b)) →
        (nf≤ (lift η) ∞<$> eta v) ∞~⟨ i ⟩~ (eta (val≤ η v))
```

As an example of a commutivity lemma, we show the proofs of the base case (type ⋆) for `readback≤`. The proof is a chain of bisimulation equations (in relation `_⟨ i ⟩_`), and we use the preorder reasoning package of Agda's standard library which provides nice syntax for equality chains, following an idea of Augustsson [Aug99]. Justification for each step is provided in angle brackets, some steps (≡⟨⟩) hold directly by definition.

```
readback≤ ⋆ η (ne w) =
  proof
    nf≤ η  <$>  (ne  <$> nereadback w)    ~⟨ map-compose (nereadback w) ⟩
    (nf≤ η ∘ ne)     <$> nereadback w     ≡⟨⟩
    (Nf.ne ∘ nen≤ η) <$> nereadback w     ~⟨ ~sym (map-compose (nereadback w)) ⟩
    ne <$>  (nen≤ η  <$> nereadback w)    ~⟨ map-cong ne (nereadback≤ η w) ⟩
    ne <$>   nereadback (nev≤ η w)
  ■
  where open ~-Reasoning
```

We must also be able to weaken proofs of strong computability. Again we skip the proofs.

```
nereadback≤⇓ : ∀{Γ Δ a}(η : Δ ≤ Γ)(t : Ne Val Γ a){n : Ne Nf Γ a} →
              nereadback t ⇓ n → nereadback (nev≤ η t) ⇓ nen≤ η n
V⟦⟧≤ : ∀{Δ Δ′} a (η : Δ′ ≤ Δ)(v : Val Δ a)(⟦v⟧ : V⟦ a ⟧ v) → V⟦ a ⟧ (val≤ η v)
E⟦⟧≤ : ∀{Γ Δ Δ′} (η : Δ′ ≤ Δ) (ρ : Env Δ Γ) (θ : E⟦ Γ ⟧ ρ) → E⟦ Γ ⟧ (env≤ η ρ)
```

Finally, we can work our way up towards the fundamental theorem of logical relations (called `term` for *termination* below). In our case, it is just a logical predicate, namely, strong computability `C⟦_⟧`, but the proof technique is the same: induction on well-typed terms. To this end, we establish lemmas for each case, calling them ⟦var⟧, ⟦abs⟧, and ⟦app⟧. To start, soundness of variable evaluation is a consequence of a sound (θ) environment ρ:

```
⟦var⟧ : ∀{Δ Γ a} (x : Var Γ a) (ρ : Env Δ Γ) (θ : E⟦ Γ ⟧ ρ) →
          C⟦ a ⟧ (now (lookup x ρ))
⟦var⟧ zero    (_ , v) (_ , v⇓) = v , now⇓ , v⇓
⟦var⟧(suc x) (ρ , _) (θ , _ ) = ⟦var⟧ x ρ θ
```

The abstraction case requires another, albeit trivial lemma: `sound-β`, which states the semantic soundness of β-expansion.

```
sound-β : ∀ {Δ Γ a b} (t : Tm (Γ , a) b) (ρ : Env Δ Γ) (u : Val Δ a) →
          C⟦ b ⟧ (eval t  (ρ , u)) → C⟦ b ⟧ (apply (lam t ρ) u)
sound-β t ρ u (v , v⇓ , ⟦v⟧) = v , later⇓ v⇓ , ⟦v⟧

⟦abs⟧ : ∀ {Δ Γ a b} (t : Tm (Γ , a) b) (ρ : Env Δ Γ) (θ : E⟦ Γ ⟧ ρ) →
  (∀{Δ′}(η : Δ′ ≤ Δ)(u : Val Δ′ a)(u⇓ : V⟦ a ⟧ u) → C⟦ b ⟧ (eval t (env≤ η ρ , u))) →
  C⟦ a ⇒ b ⟧ (now (lam t ρ))
⟦abs⟧ t ρ θ ih = lam t ρ , now⇓ , (λ η u p → sound-β t (env≤ η ρ) u (ih η u p))
```

The lemma for application is straightforward, the proof term is just a bit bloated by the need to apply the first functor law `val≤-id` to fix the types.

```
⟦app⟧ : ∀ {Δ a b} {f? : Delay _ (Val Δ (a ⇒ b))} {u? : Delay _ (Val Δ a)} →
          C⟦ a ⇒ b ⟧ f? → C⟦ a ⟧ u? → C⟦ b ⟧ (f? >>= λ f → u? >>= apply f)
```

```
⟦app⟧ {u? = u?} (f , f⇓ , ⟦f⟧) (u , u⇓ , ⟦u⟧) =
  let v , v⇓ , ⟦v⟧ = ⟦f⟧ id u ⟦u⟧
      v⇓′          = bind⇓ (λ f′ → u? >>= apply f′)
                       f⇓
                       (bind⇓ (apply f)
                              u⇓
                              (subst (λ X → apply X u ⇓ v)
                                     (val≤-id f)
                                     v⇓))
  in  v , v⇓′ , ⟦v⟧
```

Evaluation is sound, in particular, it terminates. The proof of `term` proceeds by induction on the terms and is straightforward after our preparations.

```
term : ∀ {Δ Γ a} (t : Tm Γ a) (ρ : Env Δ Γ) (θ : E⟦ Γ ⟧ ρ) → C⟦ a ⟧ (eval t ρ)
term (var x)   ρ θ = ⟦var⟧ x ρ θ
term (abs t)   ρ θ = ⟦abs⟧ t ρ θ (λ η u p →
  term t (env≤ η ρ , u) (E⟦⟧≤ η ρ θ , p))
term (app t u) ρ θ = ⟦app⟧ (term t ρ θ) (term u ρ θ)
```

Termination of readback for strongly computable values follows from the following two mutually defined lemmas. The are proved mutually by induction on types.

To reify a functional value `f`, we need to reflect the fresh variable `(var zero)` to obtain a value `u` with sematics ⟦u⟧. We can then apply the semantic function ⟦f⟧ to `u` and recursively reify the returned value `v`.

```
mutual
  reify : ∀{Γ} a (v : Val Γ a) → V⟦ a ⟧ v → readback v ⇓
  reify ⋆        (ne _) (m , ⇓m) = ne m , map⇓ ne ⇓m
  reify (a ⇒ b)  f      ⟦f⟧      =
    let u           = ne (var zero)
        ⟦u⟧          = reflect a (var zero) (var zero , now⇓)
        v , v⇓ , ⟦v⟧ = ⟦f⟧ wk u ⟦u⟧
        n , ⇓n = reify b v ⟦v⟧
        ⇓λn    = later⇓ (bind⇓ (λ x → now (lam x))
                              (bind⇓ readback v⇓ ⇓n)
                              now⇓)
    in  lam n , ⇓λn
```

Reflecting a neutral value `w` at function type `a ⇒ b` returns a semantic function, which, if applied to a value `u` of type `a` and its semantics ⟦u⟧, in essence reflects recursively the application of `w` to `u`, which is again neutral, at type `b`. A little more has to be done, though, e.g., we also show that this application can be read back.

```
  reflect : ∀{Γ} a (w : Ne Val Γ a) → nereadback w ⇓ → V⟦ a ⟧ (ne w)
  reflect ⋆ w w⇓ = w⇓
  reflect (a ⇒ b) w (m , w⇓m) η u ⟦u⟧ =
    let n , ⇓n = reify a u ⟦u⟧
        m′     = nen≤ η m
```

```
      ⇓m      = nereadback≤⇓ η w w⇓m
      wu      = app (nev≤ η w) u
      ⟦wu⟧    = reflect b wu (app m′ n ,
                    bind⇓ (λ m → app m <$> readback u)
                          ⇓m
                          (bind⇓ (λ n → now (app m′ n)) ⇓n now⇓))
  in  ne wu , now⇓ , ⟦wu⟧
```

As immediate corollaries we get that all variables are strongly computable and that the identity environment is strongly computable.

```
var↑ : ∀{Γ a}(x : Var Γ a) → V⟦ a ⟧ ne (var x)
var↑ x = reflect _ (var x) (var x , now⇓)

⟦ide⟧ : ∀ Γ → E⟦ Γ ⟧ (ide Γ)
⟦ide⟧ ε       = _
⟦ide⟧ (Γ , a) = E⟦⟧≤ wk (ide Γ) (⟦ide⟧ Γ) , var↑ zero
```

Finally we can plug the termination of `eval` in the identity environment to yield a strongly computable value and the termination of `readback` give a strongly computable value to yield the termination of `nf`.

```
normalize : ∀ Γ a (t : Tm Γ a) → ∃ λ n → nf t ⇓ n
normalize Γ a t = let v , v⇓ , ⟦v⟧ = term t (ide Γ) (⟦ide⟧ Γ)
                      n , ⇓n      = reify a v ⟦v⟧
                  in  n , bind⇓ readback v⇓ ⇓n
```

5 Conclusions

We have presented a coinductive normalizer for simply typed lambda calculus and proved that that it terminates. The combination of the coinductive normalizer and termination proof yield a terminating normalizer function in type theory.

The successful formalization serves as a proof-of-concept for coinductive programming and proving using sized types and copatterns, a new and presently experimental feature of Agda. The approach we have taken lifts easily to extensions such as Gödel's System T.

Acknowledgments The authors are grateful to Nils Anders Danielsson for discussions and his talk at *Shonan Meeting 026: Coinduction for computation structures and programming* in October 2013 which inspired this work.

Andreas Abel has been supported by a framework grant of Vetenskapsrådet to the Chalmers ProgLog group, held by Thierry Coquand. James Chapman has been supported by the ERDF funded Estonian CoE project EXCS and ICT National Programme project "Coinduction", the Estonian Ministry of Research and Education target-financed research theme no. 0140007s12 and the Estonian Science Foundation grants no. 9219.

References

[AC09] Thorsten Altenkirch and James Chapman. Big-step normalisation. *Journal of Functional Programming*, 19(3-4):311–333, 2009.

[ACD08] Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*, volume 5133 of *Lecture Notes in Computer Science*, pages 29–56. Springer-Verlag, 2008.

[AJ05] Klaus Aehlig and Felix Joachimski. Continuous normalization for the lambda-calculus and Gödel's T. *Annals of Pure and Applied Logic*, 133:39–71, 2005.

[AP13] Andreas Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *International Conference on Functional Programming (ICFP 2013)*. ACM Press, 2013.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23 - 25, 2013*, pages 27–38. ACM Press, 2013.

[Aug99] Lennart Augustsson. Equality proofs in Cayenne. Unpublished note, TeX source see http://www.augustsson.net/Darcs/Cayenne/doc/eqproof.tex, 1999.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse to the evaluation functional for typed λ-calculus. In *Sixth Annual Symposium on Logic in Computer Science (LICS '91), July, 1991, Amsterdam, The Netherlands, Proceedings*, pages 203–211. IEEE Computer Society Press, 1991.

[Cap05] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[Cha09] James Chapman. *Type Checking and Normalization*. PhD thesis, School of Computer Science, University of Nottingham, 2009.

[Dan99] Olivier Danvy. Type-directed partial evaluation. In *Partial Evaluation – Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*, pages 367–411. Springer-Verlag, 1999.

[Dan12] Nils Anders Danielsson. Operational semantics using the partiality monad. In *Proceedings of the Seventeenth ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 127–138. ACM Press, 2012.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, volume 37 of *SIGPLAN Notices*, pages 235–246. ACM Press, 2002.

[KA10] Chantal Keller and Thorsten Altenkirch. Hereditary Substitutions for Simple Types, Formalized. In *Third Workshop on Mathematically Structured Functional Programming, MSFP 2010, Baltimore, USA, September 25, 2010*, Baltimore, USA, 2010. ACM Press.

[Min78] Grigori Mints. Finite investigations of transfinite derivations. *Journal of Soviet Mathematics*, 10:548–596, 1978. Translated from: Zap. Nauchn. Semin. LOMI 49 (1975).

[Pra65] Dag Prawitz. *Natural Deduction*. Almqvist & Wiksell, Stockholm, 1965. Republication by Dover Publications Inc., Mineola, New York, 2006.

[WCPW03] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgements and properties. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, 2003.