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 and Gothenburg University
Sweden
andreas.abel@gu.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 normalizer consists of two coinductively defined functions in the delay monad: One is a standard evaluator of lambda terms to closures, the other a type-directed reifier from values to η-long β-normal forms. Their composition, normalization-by-evaluation, is shown to be a total function *a posteriori*, using a standard logical-relations argument.

The successful formalization serves as a proof-of-concept for coinductive programming and reasoning 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 closures 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 [15] and Abel, Coquand, and Dybjer [3]: 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 [19]) as terms in normal form.[1] 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* [7, 12]; 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 [10], but not here.

N.R. Krishnaswami and P.B. Levy (Eds.):
Mathematically Structured Functional Programming 2014 (MSFP 2014).
EPTCS 153, 2014, pp. 51–67, doi:10.4204/EPTCS.153.4

The delay monad has been used to implement evaluators before: Danielsson's *Operational Semantics Using the Partiality Monad* [14] 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 [6]. 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 [22]. In the type-theoretic community, the delay monad has been popularized by Capretta [11], and it is isomorphic to the trampolin type [17]. Escardo [16] describes a delay monad in the context of a (ultra)metric model for PCF which allows *intensional functions* that can measure the termination speed of their arguments. Indeed, the coinductive delay monad is intensional in the same sense as it makes the speed of convergence observable.

Using hereditary substitutions [24], 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 [21]. The idea of normalization by induction on types is very old, see, e.g., Prawitz [23]. 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 our 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 [2].

2 Delay Monad

The Delay type is used to represent computations of type A whose result may be returned with some delay or never be returned at all. 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 an element of ∞Delay

which we can force again, and so forth.

```
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., a lower bound on the number of times we can iteratively force the delayed computation. More precisely, forcing $a\infty : \infty\text{Delay } i\ A$ will result in a value $a\infty : \text{Delay } j\ A$ of strictly smaller observation depth $j < i$. An exception is a delayed value $a\infty : \infty\text{Delay } \infty\ A$ of infinite observation depth, whose forcing force $a\infty : \text{Delay } \infty\ A$ 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 $a? : \text{Delay } \infty\ A$ 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 hidden 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* [5]. 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 [4]. Agda can reconstruct size arguments in programs if the sizes are declared in their type signature. Thus, we omit the hidden size arguments in the following.

At each observation depth i, the functor Delay i forms a monad. The return of the monad is given by now, and bind $\_ \gg\!= \_$ 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 $a? \gg\!= f$ 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 (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 laters). 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 [6, Sec. 2.2]. Those 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

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

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 $\_\sim\langle\ i\ \rangle\sim\_$ and $\_\infty\sim\langle\ i\ \rangle\sim\_$ 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?
∞~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 $\_ \infty>>= \_$) 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 $a? \Downarrow a$, 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 → (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}{b : B} →
          ?a ⇓ a → f a ⇓ b → (?a >>= f) ⇓ b
```

That completes our discussion of 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 [7] without explicit substitutions. First we introduce simple types Ty with one base type $\star$ and function types $a \Rightarrow b$.

```
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 that only non-empty contexts can have variables, since none of the constructors targets the empty context. The zeroth variable has 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 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. Further, 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 Ξ 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 (Ξ : Cxt → Ty → Set)(Γ : Cxt) : Ty → Set where
  var  : ∀{a}   → Var Γ a                → Ne Ξ Γ a
  app  : ∀{a b} → Ne Ξ Γ (a ⇒ b) → Ξ Γ a → Ne Ξ 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 that looks variables up in the environment. Notice that the typing ensures that lookup never tries to access a variable that is out of scope 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, and a function beta that reduces a β-redex, i.e., a closure applied to a value. While eval and beta are recursively invoked only on subterms, apply is called with arguments f and v which are results of evaluating terms t and u and not structurally smaller than the arguments of caller eval. Thus, the three functions are not defined by structural induction but by mutual *coinduction*.

```
eval  : ∀{i Γ Δ b}   → Tm Γ b       → Env Δ Γ              → Delay i (Val Δ b)
apply : ∀{i Δ a b}   → Val Δ (a ⇒ b)                 → Val Δ a → Delay i (Val Δ b)
beta  : ∀{i Γ Δ a b} → Tm (Γ , a) b → Env Δ Γ → Val Δ a → ∞Delay i (Val Δ b)

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 >>= λ v → apply f v

apply (ne w)     v = now (ne (app w v))
apply (lam t ρ)  v = later (beta t ρ v)

force (beta t ρ v)  = eval t (ρ , v)
```

To justify the coinductive definition, the recursive calls must be *guarded*. Immediately guarded is only beta which only unfolds if forced. The apply function only calls beta, and this call is under constructor later, i.e., not under any elimination, thus, the code for apply is also not endangering productivity. Yet eval makes three recursive calls as arguments to the elimination _>>=_, violating the syntactic guardedness condition [13, 18] as implemented, e.g., in Coq [20] and previous Agda [8]. Sized types come to the rescue here! The typing of bind

```
_>>=_ : ∀ {i A B} → Delay i A → (A → Delay i B) → Delay i B
```

guarantees that its two arguments are observed no deeper than its result; thus, guardedness is not destroyed by a use of bind. Finally, eval calls itself only on subterms, thus, these two recursive calls, while not guarded by explicit delays, can be justified by a local structural induction on Tm. Agda's termination checker is able to recognize lexicographic termination measures [1], in this case it is a lexicographic recursion first on observation depth in the Delay monad and second on the height of Tm trees.

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 operation 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 is defined by induction on the types. If the value is of 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 of function type then we perform eta expansion; we know the result is a lam, but the lambda body cannot be immediately returned, since function values may be unevaluated closures; hence, its given later by eta. The function eta takes the function value, weakens it, then applies it to the fresh variable var zero yielding a delayed value at range type, which is read back recursively.

```
readback    : ∀{i Γ a}   → Val Γ a        → Delay i (Nf Γ a)
nereadback  : ∀{i Γ a}   → Ne Val Γ a     → Delay i (Ne Nf Γ a)
eta         : ∀{i Γ a b} → Val Γ (a ⇒ b)  → ∞Delay i (Nf (Γ , a) b)


readback {a = ⋆}       (ne w)  = ne   <$> nereadback w
readback {a = _ ⇒ _} v         = lam  <$> later (eta v)

force (eta v)                  = readback =≪ apply (weakVal v) (ne (var zero))

nereadback (var x)             = now (var x)
nereadback (app w v)           = nereadback w >>= λ m → app m <$> readback v
```

The three functions are defined by an outer coinduction into the Delay monad and an inner local induction on neutral values in nereadback. Again, the sized typing of bind and map are crucial to communicate the termination argument to Agda.

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

While we have managed to define the normalizer in a way acceptable to Agda's termination checker, we have not established that simply-typed lambda calculus is actually normalizing, i.e., that each well-typed

term reaches its normal form after a only final number of delays have been issued. To this end, we define a logical predicate $\mathsf{V}[\![\_]\!]\_$, 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, 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 $\mathsf{C}[\![\_]\!]\_$ on delayed values $v?$ is shorthand for a triple $(v, v{\Downarrow}, [\![v]\!])$ of a value v, a proof $v{\Downarrow}$ that the delayed value converges to the value and a proof $[\![v]\!]$ of strong computability.

$$
\begin{array}{lll}
\mathsf{V}[\![\_]\!]\_ & : \forall\{\Gamma\}\ (a : \mathsf{Ty}) \to \mathsf{Val}\ \Gamma\ a & \to \mathsf{Set} \\
\mathsf{C}[\![\_]\!]\_ & : \forall\{\Gamma\}\ (a : \mathsf{Ty}) \to \mathsf{Delay}\ \infty\ (\mathsf{Val}\ \Gamma\ a) & \to \mathsf{Set}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{V}[\![\ \star\ ]\!] & (\mathsf{ne}\ w) & = \mathsf{nereadback}\ w \Downarrow \\
\mathsf{V}[\![\ a \Rightarrow b\ ]\!] & f & = \forall\{\Delta\}(\eta : \Delta \le \_)(u : \mathsf{Val}\ \Delta\ a) \to \mathsf{V}[\![\ a\ ]\!]\ u \to \mathsf{C}[\![\ b\ ]\!]\ (\mathsf{apply}\ (\mathsf{val}{\le}\ \eta\ f)\ u)
\end{array}
$$

$$
\mathsf{C}[\![\ a\ ]\!] \qquad v? \qquad = \exists\, \lambda\, v \to v? \Downarrow v \times \mathsf{V}[\![\ a\ ]\!]\ v
$$

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

$$
\begin{array}{lll}
\mathsf{E}[\![\_]\!]\_ & & : \quad \forall\{\Delta\}(\Gamma : \mathsf{Cxt}) \to \mathsf{Env}\ \Delta\ \Gamma \to \mathsf{Set} \\
\mathsf{E}[\![\ \varepsilon\ ]\!] & \varepsilon & = \top \\
\mathsf{E}[\![\ \Gamma\ ,\ a\ ]\!] & (\rho\ ,\ v) & = \mathsf{E}[\![\ \Gamma\ ]\!]\ \rho \times \mathsf{V}[\![\ a\ ]\!]\ v
\end{array}
$$

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.

$$
\mathsf{val}{\le}\text{-}\mathsf{id} \ : \forall\{\Delta\ a\}\ \ (v : \mathsf{Val}\ \Delta\ a) \qquad \to \mathsf{val}{\le}\ \mathsf{id}\ v \equiv v
$$

$$
\mathsf{env}{\le}\text{-}\mathsf{id} \ : \forall\{\Gamma\ \Delta\}\ \ (\rho : \mathsf{Env}\ \Delta\ \Gamma) \qquad \to \mathsf{env}{\le}\ \mathsf{id}\ \rho \equiv \rho
$$

$$
\mathsf{nev}{\le}\text{-}\mathsf{id} \ : \forall\{\Delta\ a\}\ \ (t : \mathsf{Ne}\ \mathsf{Val}\ \Delta\ a) \ \to \mathsf{nev}{\le}\ \mathsf{id}\ t \equiv t
$$

$$
\begin{array}{ll}
\mathsf{var}{\le}\text{-}\bullet & : \forall\{\Delta\ \Delta'\ \Delta''\ a\}\ (\eta : \Delta \le \Delta')\ (\eta' : \Delta' \le \Delta'')\ (x : \mathsf{Var}\ \Delta''\ a) \to \\
& \quad \mathsf{var}{\le}\ \eta\ (\mathsf{var}{\le}\ \eta'\ x) \equiv \mathsf{var}{\le}\ (\eta \bullet \eta')\ x
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{val}{\le}\text{-}\bullet & : \forall\{\Delta\ \Delta'\ \Delta''\ a\}\ (\eta : \Delta \le \Delta')\ (\eta' : \Delta' \le \Delta'')\ (v : \mathsf{Val}\ \Delta''\ a) \to \\
& \quad \mathsf{val}{\le}\ \eta\ (\mathsf{val}{\le}\ \eta'\ v) \equiv \mathsf{val}{\le}\ (\eta \bullet \eta')\ v
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{env}{\le}\text{-}\bullet & : \forall\{\Gamma\ \Delta\ \Delta'\ \Delta''\}\ (\eta : \Delta \le \Delta')\ (\eta' : \Delta' \le \Delta'')\ (\rho : \mathsf{Env}\ \Delta''\ \Gamma) \to \\
& \quad \mathsf{env}{\le}\ \eta\ (\mathsf{env}{\le}\ \eta'\ \rho) \equiv \mathsf{env}{\le}\ (\eta \bullet \eta')\ \rho
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{nev}{\le}\text{-}\bullet & : \forall\{\Delta\ \Delta'\ \Delta''\ a\}\ (\eta : \Delta \le \Delta')\ (\eta' : \Delta' \le \Delta'')\ (t : \mathsf{Ne}\ \mathsf{Val}\ \Delta''\ a) \to \\
& \quad \mathsf{nev}{\le}\ \eta\ (\mathsf{nev}{\le}\ \eta'\ t) \equiv \mathsf{nev}{\le}\ (\eta \bullet \eta')\ t
\end{array}
$$

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 $\star$) 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 [9]. 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⟦ a ⟧ v  → V⟦ a ⟧ (val≤ η v)
E⟦⟧≤         : ∀{Γ Δ Δ′}  (η : Δ′ ≤ Δ)  (ρ : Env Δ Γ)  → 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) → 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 f′ → apply f′ 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. They 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 semantics $[\![u]\!]$. 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 $a \Rightarrow b$ 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 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. We also thank the anonymous referees for their helpful comments.

Andreas Abel has been supported by framework grant 254820104 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 grant no. 9219.

This article has been type set with the Stevan Andjelkovic's LaTeX backend for Agda.

References

[1] Andreas Abel & Thorsten Altenkirch (2002): *A Predicative Analysis of Structural Recursion*. *Journal of Functional Programming* 12(1), pp. 1–41. Available at `http://dx.doi.org/10.1017/S0956796801004191`.

[2] Andreas Abel & James Chapman (2014): *Normalization by Evaluation in the Delay Monad: Literate Agda Code*. Available at `http://www.cse.chalmers.se/~abela/eptcs14.lagda`. Tested with Agda development version and standard library of the date of publication.

[3] Andreas Abel, Thierry Coquand & Peter Dybjer (2008): *Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory*. In: Philippe Audebaud & Christine Paulin-Mohring, editors: *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings, Lecture Notes in Computer Science* 5133, Springer-Verlag, pp. 29–56. Available at `http://dx.doi.org/10.1007/978-3-540-70594-9_4`.

[4] Andreas Abel & Brigitte Pientka (2013): *Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity*. In: Greg Morrisett & Tarmo Uustalu, editors: *Proceedings of the Eighteenth ACM*

SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA, September 25-27, 2013, ACM Press, pp. 185–196. Available at http://doi.acm.org/10.1145/2500365.2500591.

[5] Andreas Abel, Brigitte Pientka, David Thibodeau & Anton Setzer (2013): *Copatterns: Programming Infinite Structures by Observations*. In: Roberto Giacobazzi & Radhia Cousot, editors: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23 - 25, 2013*, ACM Press, pp. 27–38. Available at http://doi.acm.org/10.1145/2429069.2429075.

[6] Klaus Aehlig & Felix Joachimski (2005): *Continuous Normalization for the Lambda-Calculus and Gödel's T*. *Annals of Pure and Applied Logic* 133, pp. 39–71. Available at http://dx.doi.org/10.1016/j.apal.2004.10.003.

[7] Thorsten Altenkirch & James Chapman (2009): *Big-step normalisation*. *Journal of Functional Programming* 19(3-4), pp. 311–333. Available at http://dx.doi.org/10.1017/S0956796809007278.

[8] Thorsten Altenkirch & Nils Anders Danielsson (2010). *Termination Checking in the Presence of Nested Inductive and Coinductive Types*. Short note supporting a talk given at PAR 2010, Workshop on Partiality and Recursion in Interactive Theorem Provers, FLoC 2010. Available at http://www.cse.chalmers.se/~nad/publications/altenkirch-danielsson-par2010.pdf.

[9] Lennart Augustsson (1999): *Equality proofs in Cayenne*. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.9415. Unpublished note, TeX source see http://www.augustsson.net/Darcs/Cayenne/doc/eqproof.tex.

[10] Ulrich Berger & Helmut Schwichtenberg (1991): *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*, IEEE Computer Society Press, pp. 203–211. Available at http://dx.doi.org/10.1109/LICS.1991.151645.

[11] Venanzio Capretta (2005): *General Recursion via Coinductive Types*. *Logical Methods in Computer Science* 1(2). Available at http://dx.doi.org/10.2168/LMCS-1(2:1)2005.

[12] James Chapman (2009): *Type Checking and Normalization*. Ph.D. thesis, School of Computer Science, University of Nottingham.

[13] Thierry Coquand (1993): *Infinite Objects in Type Theory*. In: H. Barendregt & T. Nipkow, editors: *Types for Proofs and Programs (TYPES '93), Lecture Notes in Computer Science* 806, Springer-Verlag, pp. 62–78. Available at http://dx.doi.org/10.1007/3-540-58085-9_72.

[14] Nils Anders Danielsson (2012): *Operational semantics using the partiality monad*. In: Peter Thiemann & Robby Bruce Findler, editors: *Proceedings of the Seventeenth ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, ACM Press, pp. 127–138. Available at http://doi.acm.org/10.1145/2364527.2364546.

[15] Olivier Danvy (1999): *Type-Directed Partial Evaluation*. In: John Hatcliff, Torben Æ. Mogensen & Peter Thiemann, editors: *Partial Evaluation – Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998, Lecture Notes in Computer Science* 1706, Springer-Verlag, pp. 367–411. Available at http://doi.acm.org/10.1145/237721.237784.

[16] Martin H. Escardo (1999). *A metric model of PCF*. Presented at the Workshop on Realizability Semantics and Applications, June 30-July 1, 1999 (associated to the Federated Logic Conference, held in Trento, June 29-July 12, 1999).

[17] Steven E. Ganz, Daniel P. Friedman & Mitchell Wand (1999): *Trampolined Style*. In: Didier Rémi & Peter Lee, editors: *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France*, ACM Press, pp. 18–27. Available at http://doi.acm.org/10.1145/317636.317779.

[18] Eduardo Giménez (1995): *Codifying Guarded Definitions with Recursive Schemes*. In: Peter Dybjer, Bengt Nordström & Jan Smith, editors: *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers, Lecture Notes in Computer Science* 996, Springer-Verlag, pp. 39–59. Available at http://dx.doi.org/10.1007/3-540-60579-7_3.

[19] Benjamin Grégoire & Xavier Leroy (2002): *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*, *SIGPLAN Notices* 37, ACM Press, pp. 235–246. Available at `http://doi.acm.org/10.1145/581478.581501`.

[20] INRIA (2012): *The Coq Proof Assistant Reference Manual*. INRIA, version 8.4 edition. Available at `http://coq.inria.fr/`.

[21] Chantal Keller & Thorsten Altenkirch (2010): *Hereditary Substitutions for Simple Types, Formalized*. In: V. Capretta & J. Chapman, editors: *Third Workshop on Mathematically Structured Functional Programming, MSFP 2010, Baltimore, USA, September 25, 2010*, ACM Press, Baltimore, USA. Available at `http://dx.doi.org/10.1145/1863597.1863601`.

[22] Grigori Mints (1978): *Finite Investigations of Transfinite Derivations*. *Journal of Soviet Mathematics* 10, pp. 548–596. Available at `http://dx.doi.org/10.1007/BF01091743`. Translated from: Zap. Nauchn. Semin. LOMI 49 (1975).

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

[24] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2003): *A concurrent logical framework I: Judgements and properties*. Technical Report, School of Computer Science, Carnegie Mellon University, Pittsburgh.