A Formalized Proof of Strong Normalization for Guarded Recursive Types (Long Version)

Andreas Abel and Andrea Vezzosi

Computer Science and Engineering, Chalmers and Gothenburg University,
Rännvägen 6, 41296 Göteborg, Sweden
andreas.abel@gu.se,vezzosi@chalmers.se

Abstract. We consider a simplified version of Nakano's guarded fixed-point types in a representation by infinite type expressions, defined coinductively. Small-step reduction is parametrized by a natural number "depth" that expresses under how many guards we may step during evaluation. We prove that reduction is strongly normalizing for any depth. The proof involves a typed inductive notion of strong normalization and a Kripke model of types in two dimensions: depth and typing context. Our results have been formalized in Agda and serve as a case study of reasoning about a language with coinductive type expressions.

1 Introduction

In untyped lambda calculus, fixed-point combinators can be defined using self-application. Such combinators can be assigned recursive types, albeit only negative ones. Since such types introduce logical inconsistency, they are ruled out in Martin-Löf Type Theory and other systems based on the Curry-Howard isomorphism. Nakano (2000) introduced *a modality for recursion* that allows a stratification of negative recursive types to recover consistency. In essence, each negative recursive occurrence needs to be *guarded* by the modality; this coined the term *guarded recursive types* (Birkedal and Møgelberg, 2013).[1] Nakano's modality has found applications in functional reactive programming (Krishnaswami and Benton, 2011b) where it is referred to as *later* modality.

While Nakano showed that every typed term has a weak head normal form, in this paper we prove *strong normalization* for our variant $\lambda^{\blacktriangleright}$ of Nakano's calculus. To this end, we make the introduction rule for the later modality explicit in the terms by a constructor next, following Birkedal and Møgelberg (2013) and Atkey and McBride (2013). By allowing reduction under finitely many nexts, we establish termination irrespective of the reduction strategy. Showing strong normalization of $\lambda^{\blacktriangleright}$ is a first step towards an operationally well-behaved type theory with guarded recursive types, for which Birkedal and Møgelberg (2013) have given a categorical model.

Our proof is fully formalized in the proof assistant Agda (2014) which is based on intensional Martin-Löf Type Theory. [2] One key idea of the formalization is to represent

[1] Not to be confused with *Guarded Recursive Datatype Constructors* (Xi et al., 2003).

[2] A similar proof could be formalized in other systems supporting mixed induction-coinduction, for instance, in Coq.

the recursive types of $\lambda^{\blacktriangleright}$ as infinite type expressions in form of a coinductive definition. For this, we utilize Agda's new *copattern* feature (Abel et al., 2013). The set of strongly normalizing terms is defined inductively by distinguishing on the shape of terms, following van Raamsdonk et al. (1999) and Joachimski and Matthes (2003). The first author has formalized this technique before in Twelf (Abel, 2008); in this work we extend these results by a proof of equivalence to the standard notion of strong normalization.

Due to space constraints, we can only give a sketch of the formalization; a longer version and the full Agda proofs are available online (Abel and Vezzosi, 2014). This paper is extracted from a literate Agda file; all the colored code in displays is necessarily type-correct.

2 Guarded Recursive Types and Their Semantics

Nakano's type system (2000) is equipped with subtyping, but we stick to a simpler variant without, a simply-typed version of Birkedal and Møgelberg (2013), which we shall call $\lambda^{\blacktriangleright}$. Our rather minimal grammar of types includes product $A \times B$ and function types $A \to B$, delayed computations $\blacktriangleright A$, variables X and explicit fixed-points μXA.

$$A, B, C ::= A \times B \mid A \to B \mid \blacktriangleright A \mid X \mid \mu XA$$

Base types and disjoint sum types could be added, but would only give breadth rather than depth to our formalization. As usual, a dot after a bound variable shall denote an opening parenthesis that closes as far to the right as syntactically possible. Thus, $\mu X. X \to X$ denotes $\mu X\,(X \to X)$, while $\mu XX \to X$ denotes $(\mu X.X) \to X$ (with a free variable X).

Formation of fixed-points μXA is subject to the side condition that X is guarded in A, i. e., X appears in A only under a *later* modality $\blacktriangleright$. This rules out all unguarded recursive types like $\mu X.A \times X$ or $\mu X.X \to A$, but allows their variants $\mu X.\blacktriangleright (A \times X)$ and $\mu X.A \times \blacktriangleright X$, and $\mu X.\blacktriangleright (X \to A)$ and $\mu X.\blacktriangleright X \to A$. Further, fixed-points give rise to an equality relation on types induced by $\mu XA = A[\mu XA/X]$.

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}$$

$$\frac{\Gamma \vdash t_1 : A_1 \qquad \Gamma \vdash t_2 : A_2}{\Gamma \vdash (t_1, t_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \mathsf{fst}\, t : A_1} \qquad \frac{\Gamma \vdash t : A_1 \times A_2}{\Gamma \vdash \mathsf{snd}\, t : A_2}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{next}\, t : \blacktriangleright A} \qquad \frac{\Gamma \vdash t : \blacktriangleright (A \to B) \qquad \Gamma \vdash u : \blacktriangleright A}{\Gamma \vdash t * u : \blacktriangleright B} \qquad \frac{\Gamma \vdash t : A \qquad A = B}{\Gamma \vdash t : B}$$

Fig. 1. Typing rules.

Terms are lambda-terms with pairing and projection plus operations that witness *applicative functoriality* of the later modality (Atkey and McBride, 2013).

$$t,u ::= x \mid \lambda x t \mid t\,u \mid (t_1,t_2) \mid \mathsf{fst}\,t \mid \mathsf{snd}\,t \mid \mathsf{next}\,t \mid t * u$$

Figure 1 recapitulates the static semantics. The dynamic semantics is induced by the following *contractions*:

$$\begin{array}{lll} (\lambda x.\,t)\,u & \mapsto & t[u/x] \\ \mathsf{fst}\,(t_1,t_2) & \mapsto & t_1 \\ \mathsf{snd}\,(t_1,t_2) & \mapsto & t_2 \\ (\mathsf{next}\,t) * (\mathsf{next}\,u) & \mapsto & \mathsf{next}\,(t\,u) \end{array}$$

If we conceive our small-step reduction relation $\longrightarrow$ as the compatible closure of $\mapsto$, we obtain a non-normalizing calculus, since terms like $\Omega = \omega\,(\mathsf{next}\,\omega)$ with $\omega = (\lambda x.\,x * (\mathsf{next}\,x))$ are typeable.[3] Unrestricted reduction of Ω is non-terminating: $\Omega \longrightarrow \mathsf{next}\,\Omega \longrightarrow \mathsf{next}\,(\mathsf{next}\,\Omega) \longrightarrow \ldots$ If we let next act as delay operator that blocks reduction inside, we regain termination. In general, we preserve termination if we only look under delay operators up to a certain depth. This can be made precise by a family $\longrightarrow_n$ of reduction relations indexed by a depth $n \in \mathbb{N}$, see Figure 2.

$$\frac{t \mapsto t'}{t \longrightarrow_n t'} \qquad \frac{t \longrightarrow_n t'}{\lambda x.\,t \longrightarrow_n \lambda x.\,t'} \qquad \frac{t \longrightarrow_n t'}{t\,u \longrightarrow_n t'\,u} \qquad \frac{u \longrightarrow_n u'}{t\,u \longrightarrow_n t\,u'}$$

$$\frac{t \longrightarrow_n t'}{(t,u) \longrightarrow_n (t',u)} \qquad \frac{u \longrightarrow_n u'}{(t,u) \longrightarrow_n (t,u')} \qquad \frac{t \longrightarrow_n t'}{\mathsf{fst}\,t \longrightarrow_n \mathsf{fst}\,t'} \qquad \frac{t \longrightarrow_n t'}{\mathsf{snd}\,t \longrightarrow_n \mathsf{snd}\,t'}$$

$$\boxed{\frac{t \longrightarrow_n t'}{\mathsf{next}\,t \longrightarrow_{n+1} \mathsf{next}\,t'}} \qquad \frac{t \longrightarrow_n t'}{t * u \longrightarrow_n t' * u} \qquad \frac{u \longrightarrow_n u'}{t * u \longrightarrow_n t * u'}$$

Fig. 2. Reduction

We should note that for a fixed depth n the relation $\longrightarrow_n$ is not confluent. In fact the term $(\lambda z.\,\mathsf{next}^{n+1}\,z)(\mathsf{fst}\,(u,t))$ reduces to two different normal forms, $\mathsf{next}^{n+1}\,(\mathsf{fst}\,(u,t))$ and $\mathsf{next}^{n+1}\,u$. We could remedy this situation by making sure we never hide redexes under too many applications of next and instead store them in an explicit substitution where they would still be accessible to $\longrightarrow_n$. Our problematic terms would then look like $\mathsf{next}^n\,((\mathsf{next}\,z)[\mathsf{fst}\,(u,t)/z])$ and $\mathsf{next}^n\,((\mathsf{next}\,z)[u/z])$ and the former would reduce to the latter. However, we are not bothered by the non-confluence since our semantics at level n (see below) does not distinguish between $\mathsf{next}^{n+1}u$ and $\mathsf{next}^{n+1}u'$ (as in $u' = \mathsf{fst}\,(u,t)$); neither u nor u' is required to terminate if buried under more than n nexts.

To show termination, we interpret types as sets $\mathscr{A},\mathscr{B},\mathscr{C}$ of depth-n strongly normalizing terms. We define semantic versions $[\![\times]\!]$, $[\![\to]\!]$, and $[\![\blacktriangleright]\!]$ of product, function

[3] $\vdash \Omega : A$ with $A = \mu X(\blacktriangleright X)$. To type ω, we use $x : \mu Y(\blacktriangleright(Y \to A))$.

space, and delay type constructor, plus a terminal (=largest) semantic type $\llbracket\top\rrbracket$. Then the interpretation $\llbracket A\rrbracket_n$ of closed type A at depth n can be given recursively as follows, using the Kripke construction at function types:

$$\begin{array}{lll@{\qquad}lll}
\llbracket A\times B\rrbracket_n & = & \llbracket A\rrbracket_n \;\llbracket\times\rrbracket\; \llbracket B\rrbracket_n & \mathscr{A}\;\llbracket\times\rrbracket\;\mathscr{B} & = & \{t \mid \mathsf{fst}\;t\in\mathscr{A} \text{ and } \mathsf{snd}\;t\in\mathscr{B}\} \\
\llbracket A\to B\rrbracket_n & = & \bigcap_{n'\leq n}(\llbracket A\rrbracket_{n'}\;\llbracket\to\rrbracket\;\llbracket B\rrbracket_{n'}) & \mathscr{A}\;\llbracket\to\rrbracket\;\mathscr{B} & = & \{t \mid t\,u\in\mathscr{B} \text{ for all } u\in\mathscr{A}\} \\
\llbracket \blacktriangleright A\rrbracket_0 & = & \llbracket\blacktriangleright\rrbracket\;\llbracket\top\rrbracket & \llbracket\top\rrbracket & = & \{t \mid t \text{ term}\} \\
\llbracket \blacktriangleright A\rrbracket_{n+1} & = & \llbracket\blacktriangleright\rrbracket\llbracket A\rrbracket_n & \llbracket\blacktriangleright\rrbracket\,\mathscr{A} & = & \overline{\{\mathsf{next}\;t \mid t\in\mathscr{A}\}} \\
\llbracket \mu XA\rrbracket_n & = & \llbracket A[\mu XA/X]\rrbracket_n & \multicolumn{3}{l}{(\overline{\mathscr{A}} \text{ is weak head expansion closure of } \mathscr{A})}
\end{array}$$

Due to the last equation (μ), the type interpretation is ill-defined for unguarded recursive types. However, for guarded types we only return to the fixed-point case after we have passed the case for $\blacktriangleright$, which decreases the index n. More precisely, $\llbracket A\rrbracket_n$ is defined by lexicographic induction on $(n, \mathsf{size}(A))$, where $\mathsf{size}(A)$ is the number of type constructor symbols ($\times$, $\to$, μ) that occur *unguarded* in A.

While all this sounds straightforward at an informal level, formalization of the described type language is quite hairy. For one, we have to enforce the restriction to well-formed (guarded) types. Secondly, our type system contains a conversion rule, getting us into the vicinity of dependent types which are still a challenge to a completely formal treatment (McBride, 2010). Our first formalization attempt used kinding rules for types to keep track of guardedness for formation of fixed-point, and a type equality relation, and building on this, inductively defined well-typed terms. However, the complexity was discouraging and lead us to a much more economic representation of types, which is described in the next section.

3 Formalized Syntax

In this section, we discuss the formalization of types, terms, and typing of $\lambda^{\blacktriangleright}$ in Agda. It will be necessary to talk about meta-level types, i. e., Agda's types, thus, we will refer to $\lambda^{\blacktriangleright}$'s type constructors as $\hat{\times}$, $\hat{\to}$, $\hat{\blacktriangleright}$, and $\hat{\mu}$.

3.1 Types Represented Coinductively

Instead of representing fixed-points as syntactic construction on types, which would require a non-trivial equality on types induced by $\hat{\mu}XA = A[\hat{\mu}XA/X]$, we use *meta-level* fixed-points, i. e., Agda's recursion mechanism.[4] Extensionally, we are implementing *infinite type expressions* over the constructors $\hat{\times}$, $\hat{\to}$, and $\hat{\blacktriangleright}$. The guard condition on recursive types then becomes an instance of Agda's "guard condition", i. e., the condition the termination checker imposes on recursive programs.

[4] An alternative to get around the type equality problem would be iso-recursive types, i. e., with term constructors for folding and unfolding of $\hat{\mu}XA$. However, we would still have to implement type variables, binding of type variables, type substitution, lemmas about type substitution etc.

Viewed as infinite expressions, guarded types are regular trees with an infinite number of $\hat{\blacktriangleright}$-nodes on each infinite path. This can be expressed as the mixed coinductive(ν)-inductive(μ) (meta-level) type

$$\nu X \mu Y.\, (Y \times Y) + (Y \times Y) + X.$$

The first summand stands for the binary constructor $\hat{\times}$, the second for $\hat{\rightarrow}$, and the third for the unary $\hat{\blacktriangleright}$. The nesting of a least-fixed point (μ) inside a greatest fixed-point (ν) ensures that on each path, we can only take alternatives $\hat{\times}$ and $\hat{\rightarrow}$ a finite number of times before we have to choose the third alternative $\hat{\blacktriangleright}$ and restart the process.

In Agda 2.4, we represent this mixed coinductive-inductive type by a datatype Ty (inductive component) mutually defined with a record ∞Ty (coinductive component).

```
mutual
  data Ty : Set where
    _×̂_ : (a b : Ty)    → Ty
    _→̂_ : (a b : Ty)    → Ty
    ▶̂_   : (a∞ : ∞Ty) → Ty

  record ∞Ty : Set where
    coinductive
    constructor  delay_
    field        force_ : Ty
```

While the arguments a and b of the infix constructors $\hat{\times}$ and $\hat{\rightarrow}$ are again in Ty, the prefix constructor $\hat{\blacktriangleright}$ expects and argument $a\infty$ in ∞Ty, which is basically a wrapping[5] of Ty. The functions delay and force convert back and forth between Ty and ∞Ty so that both types are valid representations of the set of types of $\lambda^{\blacktriangleright}$.

```
delay : Ty → ∞Ty
force : ∞Ty → Ty
```

However, since ∞Ty is declared coinductive, its inhabitants are not evaluated until forced. This allows us to represent infinite type expressions, like top $= \hat{\mu}X(\hat{\blacktriangleright}X)$.

```
top : ∞Ty
force top = ▶̂ top
```

Technically, top is defined by *copattern* matching (Abel et al., 2013); top is uniquely defined by the value of its only field, force top, which is given as $\hat{\blacktriangleright}$ top. Agda will use the given equation for its internal normalization procedure during type-checking. Alternatively, we could have tried to define top : Ty by top = $\hat{\blacktriangleright}$ delay top. However, Agda will rightfully complain here since rewriting with this equation would keep expanding top forever, thus, be non-terminating. In contrast, rewriting with the original equation is terminating since at each step, one application of force is removed.

The following two defined type constructors will prove useful in the definition of well-typed terms to follow.

[5] Similar to a `newtype` in the functional programming language Haskell.

```
▸_ : Ty → Ty
▸ a = ▸̂ delay a

_⇒_ : (a∞ b∞ : ∞Ty) → ∞Ty
force (a∞ ⇒ b∞) = force a∞ →̂ force b∞
```

3.2 Well-typed terms

Instead of a raw syntax and a typing relation, we represent well-typed terms directly by an inductive family (Dybjer, 1994). Our main motivation for this choice is the beautiful inductive definition of strongly normalizing terms to follow in Section 5. Since it relies on a classification of terms into the three shapes *introduction*, *elimination*, and *weak head redex*, it does not capture all strongly normalizing raw terms, in particular “junk” terms such as fst (λxx). Of course, statically well-typed terms come also at a cost: for almost all our predicates on terms we need to show that they are natural in the typing context, i. e., closed under well-typed renamings. This expense might be compensated by the extra assistance Agda can give us in proof construction, which is due to the strong constraints on possible solutions imposed by the rich typing.

Our encoding of well-typed terms follows closely Altenkirch and Reus (1999); McBride (2006); Benton et al. (2012). We represent typed variables x : Var Γ a by de Brujin indices, i. e., positions in a typing context Γ : Cxt, which is just a list of types.

```
Cxt = List Ty

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

Arguments enclosed in braces, such as Γ, a, and b in the types of the constructors zero and suc, are hidden and can in most cases be inferred by Agda. If needed, they can be passed in braces, either as positional arguments (e. g., $\{\Delta\}$) or as named arguments (e. g., $\{\Gamma = \Delta\}$). If ∀ prefixes bindings in a function type, the types of the bound variables may be omitted. Thus, ∀{Γ a} → A is short for {Γ : Cxt}{a : Ty} → A.

Terms t : Tm Γ a are indexed by a typing context Γ and their type a, guaranteeing well-typedness and well-scopedness. In the following data type definition, Tm (Γ : Cxt) shall mean that all constructors uniformly take Γ as their first (hidden) argument.

```
data Tm (Γ : Cxt) : (a : Ty) → Set where
  var   : ∀{a}       (x : Var Γ a)                             → Tm Γ a
  abs   : ∀{a b}     (t : Tm (a :: Γ) b)                       → Tm Γ (a →̂ b)
  app   : ∀{a b}     (t : Tm Γ (a →̂ b))  (u : Tm Γ a)          → Tm Γ b
  pair  : ∀{a b}     (t : Tm Γ a)         (u : Tm Γ b)          → Tm Γ (a ×̂ b)
  fst   : ∀{a b}     (t : Tm Γ (a ×̂ b))                        → Tm Γ a
  snd   : ∀{a b}     (t : Tm Γ (a ×̂ b))                        → Tm Γ b
  next  : ∀{a∞}      (t : Tm Γ (force a∞))                     → Tm Γ (▸̂ a∞)
  _*_   : ∀{a∞ b∞}  (t : Tm Γ (▸̂(a∞ ⇒ b∞))) (u : Tm Γ (▸̂ a∞)) → Tm Γ (▸̂ b∞)
```

The most natural typing for next and ∗ would be using the defined ▶_ : Ty → Ty:

```
next  : ∀{a}    (t : Tm Γ a)                            → Tm Γ (▶ a)
_∗_   : ∀{a b}  (t : Tm Γ (▶(a →̂ b))) (u : Tm Γ (▶ a))  → Tm Γ (▶ b)
```

However, this would lead to indices like ▶̂ delay *a* and unification problems Agda cannot solve, since matching on a coinductive constructor like delay is forbidden—it can lead to a loss of subject reduction (McBride, 2009). The chosen alternative typing, which parametrizes over *a*∞ *b*∞ : ∞Ty rather than *a b* : Ty, works better in practice.

3.3 Type Equality

Although our coinductive representation of $\lambda^{\blacktriangleright}$ types saves us from type variables, type substitution, and fixed-point unrolling, the question of type equality is not completely settled. The propositional equality ≡ of Martin-Löf Type Theory is intensional in the sense that only objects with the same *code* (modulo definitional equality) are considered equal. Thus, ≡ is adequate only for finite objects (such as natural numbers and lists) but not for infinite objects like functions, streams, or $\lambda^{\blacktriangleright}$ types.

However, we can define extensional equality or *bisimulation* on Ty as a mixed coinductive-inductive relation ≅/∞≅ that follows the structure of Ty/∞Ty (hence, we reuse the constructor names ×̂, →̂, and ▶̂).

```
mutual
  data _≅_ : (a b : Ty) → Set where
    _×̂_  : ∀{a a' b b'}  (a≅ : a ≅ a') (b≅ : b ≅ b')  → (a ×̂ b) ≅ (a' ×̂ b')
    _→̂_  : ∀{a a' b b'}  (a≅ : a' ≅ a) (b≅ : b ≅ b')  → (a →̂ b) ≅ (a' →̂ b')
    ▶̂_   : ∀{a∞ b∞}      (a≅ : a∞ ∞≅ b∞)              → ▶̂ a∞  ≅ ▶̂ b∞

  record _∞≅_ (a∞ b∞ : ∞Ty) : Set where
    coinductive
    constructor ≅delay
    field       ≅force : force a∞ ≅ force b∞
```

Ty-equality is indeed an equivalence relation (we omit the standard proof).

```
≅refl   : ∀{a}      → a ≅ a
≅sym    : ∀{a b}    → a ≅ b  → b ≅ a
≅trans  : ∀{a b c}  → a ≅ b  → b ≅ c → a ≅ c
```

However, unlike for ≡ we do not get a generic substitution principle for ≅, but have to prove it for any function and predicate on Ty. In particular, we have to show that we can cast a term in Tm Γ *a* to Tm Γ *b* if *a* ≅ *b*, which would require us to build type equality at least into Var Γ *a*. In essence, this would amount to work with setoids across all our development, which would add complexity without strengthening our result. Hence, we fall for the shortcut:

It is consistent to postulate that bisimulation implies equality, similarly to the functional extensionality principle for function types. This lets us define the function cast to convert terms between bisimilar types.

```
postulate ≅-to-≡ : ∀ {a b} → a ≅ b → a ≡ b

cast : ∀{Γ a b} (eq : a ≅ b) (t : Tm Γ a) → Tm Γ b
```

We shall require cast in uses of functorial application, to convert a type c∞ : ∞Ty into something that can be forced into a function type.

```
▶app : ∀{Γ c∞ b∞ a}  (eq : c∞ ∞≅ (delay a ⇒ b∞))
                     (t : Tm Γ (▶̂ c∞)) (u : Tm Γ (▶ a)) → Tm Γ (▶̂ b∞)
▶app eq t u = cast (▶̂ eq) t * u
```

3.4 Examples

Following Nakano (2000), we can adapt the Y combinator from the untyped lambda calculus to define a guarded fixed point combinator:

$$\mathsf{fix} = \lambda f.\ (\lambda x.\ f\ (x * \mathsf{next}\ x))\ (\mathsf{next}\ (\lambda x.\ f\ (x * \mathsf{next}\ x))).$$

We construct an auxiliary type Fix a that allows safe self application, since the argument will only be available "later". This fits with the type we want for the fix combinator, which makes the recursive instance y in fix $(\lambda y.t)$ available only at the next time slot.

```
fix : ∀{Γ a} → Tm Γ ((▶ a →̂ a) →̂ a)

Fix_ : Ty → ∞Ty
force (Fix a) = ▶̂ Fix a →̂ a

selfApp : ∀{Γ a} → Tm Γ (▶̂ Fix a) → Tm Γ (▶ a)
selfApp x = ▶app (≅delay ≅refl) x (next x)

fix = abs (app L (next L))
  where
    f = var (suc zero)
    x = var zero
    L = abs (app f (selfApp x))
```

Another standard example is the type of streams, which we can also define through corecursion.

```
mutual
  Stream : Ty → Ty
  Stream a = a ×̂ ▶̂ Stream∞ a

  Stream∞ : Ty → ∞Ty
  force (Stream∞ a) = Stream a

cons : ∀{Γ a} → Tm Γ a → Tm Γ (▶ Stream a) → Tm Γ (Stream a)
cons a s = pair a (cast (▶̂ (≅delay ≅refl)) s)
```

```
head : ∀{Γ a} → Tm Γ (Stream a) → Tm Γ a
head s = fst s

tail : ∀{Γ a} → Tm Γ (Stream a) → Tm Γ (▶ Stream a)
tail s = cast (▶̂ (≅delay ≅refl)) (snd s)
```

Note that tail returns a stream inside the later modality. This ensures that functions that transform streams have to be causal, i. e., can only have access to the first n elements of the input when producing the nth element of the output. A simple example is mapping a function over a stream.

```
mapS : ∀{Γ a b} → Tm Γ ((a →̂ b) →̂ (Stream a →̂ Stream b))
```

Which is also better read with named variables.

```
mapS = λf. fix (λmapS. λs. (f s, mapS ∗ tail s))
```

4 Reduction

In this section, we describe the implementation of parametrized reduction $\longrightarrow_n$ in Agda. As a prerequisite, we need to define substitution, which in turn depends on renaming (Benton et al., 2012).

A *renaming* from context Γ to context Δ, written $\Delta \leq \Gamma$, is a mapping from variables of Γ to those of Δ of the same type a. The function rename lifts such a mapping to terms.

```
_≤_  : (Δ Γ : Cxt) → Set
_≤_ Δ Γ = ∀ {a} → Var Γ a → Var Δ a

rename : ∀ {Γ Δ : Cxt} {a : Ty} (η : Δ ≤ Γ) (x : Tm Γ a) → Tm Δ a
```

Building on renaming, we define well-typed parallel substitution. From this, we get the special case of substituting de Bruijn index 0.

```
subst0 : ∀ {Γ a b} → Tm Γ a → Tm (a :: Γ) b → Tm Γ b
```

Reduction $t \longrightarrow_n t'$ is formalized as the inductive family t ⟨n⟩⇒β t' with four constructors β... representing the contraction rules and one congruence rule cong to reduce in subterms.

```
data _⟨_⟩⇒β_ {Γ} : ∀ {a} → Tm Γ a → ℕ → Tm Γ a → Set where

  β    : ∀ {n a b}{t : Tm (a :: Γ) b}{u}
       → app (abs t) u ⟨ n ⟩⇒β subst0 u t

  βfst : ∀ {n a b}{t : Tm Γ a}{u : Tm Γ b}
       → fst (pair t u) ⟨ n ⟩⇒β t

  βsnd : ∀ {n a b}{t : Tm Γ a}{u : Tm Γ b}
       → snd (pair t u) ⟨ n ⟩⇒β u
```

```
β▸    : ∀ {n a∞ b∞}{t : Tm Γ (force a∞ →̂  force b∞)}{u : Tm Γ (force a∞)}
      → (next t ∗ next {a∞ = a∞} u) ⟨ n ⟩⇒β (next {a∞ = b∞} (app t u))

cong  : ∀ {n n' Δ a b t t' Ct Ct'}{C : NβCxt Δ Γ a b n n'}
      → (Ct   : Ct ≡ C [ t ])
      → (Ct'  : Ct' ≡ C [ t' ])
      → (t⇒β  : t ⟨ n ⟩⇒β t')
      → Ct ⟨ n' ⟩⇒β Ct'
```

The congruence rule makes use of shallow one hole contexts C, which are given by the following grammar

$$C ::= \lambda x\_ \mid \_u \mid t\ \_ \mid (t,\_) \mid (\_,u) \mid \mathsf{fst}\ \_ \mid \mathsf{snd}\ \_ \mid \mathsf{next}\ \_ \mid \_{*}u \mid t{*}\_.$$

cong says that we can reduce a term, suggestively called Ct, to a term Ct', if (1) Ct decomposes into $C[t]$, a context C filled by t, and (2) Ct' into $C[t']$, and (3) t reduces to t'. As witnessed by relation $Ct{\equiv}C[t]$, context C : NβCxt Γ Δ a b n n' produces a term Ct : Tm Γ b of depth n' if filled with a term t : Tm Δ a of depth n. The depth is unchanged except for the case next, which increases the depth by 1. Thus, t ⟨n⟩⇒β t' can contract every subterm that is under at most n many nexts.

```
data NβCxt : (Δ Γ : Cxt) (a b : Ty) (n n' : ℕ) → Set where
  abs   : ∀{Γ n a b}                           → NβCxt (a :: Γ) Γ b (a →̂  b) n n
  appl  : ∀{Γ n a b} (u  : Tm Γ a)             → NβCxt Γ Γ (a →̂ b) b n n
  appr  : ∀{Γ n a b} (t  : Tm Γ (a →̂ b))      → NβCxt Γ Γ a b n n
  pairl : ∀{Γ n a b} (u  : Tm Γ b)             → NβCxt Γ Γ a (a ×̂ b) n n
  pairr : ∀{Γ n a b} (t  : Tm Γ a)             → NβCxt Γ Γ b (a ×̂ b) n n
  fst   : ∀{Γ n a b}                           → NβCxt Γ Γ (a ×̂ b) a n n
  snd   : ∀{Γ n a b}                           → NβCxt Γ Γ (a ×̂ b) b n n
  next  : ∀{Γ n a∞}                            → NβCxt Γ Γ (force a∞) (▸̂ a∞) n (1 + n)
  ∗l_   : ∀{Γ n a∞ b∞} (u : Tm Γ (▸̂ a∞))     → NβCxt Γ Γ (▸̂ (a∞ ⇒ b∞)) (▸̂ b∞) n n
  ∗r_   : ∀{Γ n a∞ b∞}
          (t : Tm Γ (▸̂ (a∞ ⇒ b∞)))            → NβCxt Γ Γ (▸̂ a∞) (▸̂ b∞) n n


data _≡_[_] {n : ℕ} {Γ : Cxt} : {n' : ℕ} {Δ : Cxt} {b a : Ty} →
            Tm Γ b → NβCxt Δ Γ a b n n' → Tm Δ a → Set
```

5 Strong Normalization

Classically, a term is *strongly normalizing* (sn) if there's no infinite reduction sequence starting from it. Constructively, the tree of all the possible reductions from an sn term must be well-founded, or, equivalently, an sn term must be in the accessible part of the reduction relation. In our case, reduction t ⟨n⟩⇒β t' is parametrized by a depth n, thus, we get the following family of sn-predicates.

```
data sn (n : ℕ) {a Γ} (t : Tm Γ a) : Set where
  acc : (∀ {t'} → t ⟨ n ⟩⇒β t' → sn n t') → sn n t
```

Van Raamsdonk et al. (1999) pioneered a more explicit characterization of strongly normalizing terms SN, namely the least set closed under introductions, formation of neutral (=stuck) terms, and weak head expansion. We adapt their technique from lambda-calculus to $\lambda^{\blacktriangleright}$; herein, it is crucial to work with well-typed terms to avoid junk like fst $(\lambda x. x)$ which does not exist in pure lambda-calculus. To formulate a deterministic weak head evaluation, we make use of the *evaluation contexts* E : ECxt

$$E ::= \_\ u \mid \mathsf{fst}\ \_ \mid \mathsf{snd}\ \_ \mid \_ * u \mid (\mathsf{next}\ t) * \_.$$

Since weak head reduction does not go into introductions which include λ-abstraction, it does not go under binders, leaving typing context Γ fixed.

```
data ECxt (Γ : Cxt) : (a b : Ty) → Set
data _≅_[_] {Γ : Cxt} : {a b : Ty} → Tm Γ b → ECxt Γ a b → Tm Γ a → Set
```

$Et{\cong}E[t]$ witnesses the splitting of a term Et into evaluation context E and hole content t. A generalization of _≅_[_] is PCxt P which additionally requires that all terms contained in the evaluation context (that is one or zero terms) satisfy predicate P. This allows us the formulation of P-neutrals as terms of the form $\vec{E}[x]$ for some $\vec{E}[\_] = E_1[\ldots E_n[\_]]$ and a variable x where all immediate subterms satisfy P.

```
data PCxt  {Γ} (P : ∀{c} → Tm Γ c → Set) :
           ∀ {a b} → Tm Γ b → ECxt Γ a b → Tm Γ a → Set where
  appl  :  ∀ {a b t u}       (u : P u)  → PCxt P (app t u)  (appl u)  (t : (a →̂ b))
  fst   :  ∀ {a b t}                    → PCxt P (fst t)     fst       (t : (a ×̂ b))
  snd   :  ∀ {a b t}                    → PCxt P (snd t)     snd       (t : (a ×̂ b))
  *l_   :  ∀ {a∞ b∞ t u}  (u : P u)  → PCxt P (t * (u : ▶̂ a∞) : ▶̂ b∞) (*l u) t
  *r_   :  ∀ {a∞ b∞ t u}  (t : P (next {a∞ = a∞ ⇒ b∞} t))
                                        → PCxt P ((next t) * (u : ▶̂ a∞) : ▶̂ b∞) (*r t) u
```

```
data PNe   {Γ} (P : ∀{c} → Tm Γ c → Set) {b} : Tm Γ b → Set where
  var   :  ∀  x                                   → PNe P (var x)
  elim  :  ∀  {a} {t : Tm Γ a} {E Et}
           →  (n : PNe P t) (Et : PCxt P Et E t)  → PNe P Et
```

Weak head reduction (whr) is a reduction of the form $\vec{E}[t] \longrightarrow \vec{E}[t']$ where $t \mapsto t'$. It is well-known that weak head expansion (whe) does not preserve sn, e.g., $(\lambda x. y)\Omega$ is not sn even though it contracts to y. In this case, Ω is a *vanishing term* lost by reduction. If we require that all vanishing terms in a reduction are sn, weak head expansion preserves sn. In the following, we define P-whr where all vanishing terms must satisfy P.

```
data _/_⇒_  {Γ} (P : ∀{c} → Tm Γ c → Set) :
            ∀ {a} → Tm Γ a → Tm Γ a → Set where

  β      : ∀ {a b}{t : Tm (a :: Γ) b}{u}
           → (u : P u)
           → P / (app (abs t) u) ⇒ subst0 u t
```

```
βfst  : ∀ {a b}{t : Tm Γ a}{u : Tm Γ b}
        → (u : P u)
        → P / fst (pair t u) ⇒ t

βsnd  : ∀ {a b}{t : Tm Γ a}{u : Tm Γ b}
        → (t : P t)
        → P / snd (pair t u) ⇒ u

β▸    : ∀ {a∞ b∞}{t : Tm Γ (force (a∞ ⇒ b∞))}{u : Tm Γ (force a∞)}
        → P / (next t ∗ next {a∞ = a∞} u) ⇒ (next {a∞ = b∞} (app t u))

cong  : ∀ {a b t t’ Et Et’}{E : ECxt Γ a b}
        → (Et   : Et ≅ E [ t ])
        → (Et’  : Et’ ≅ E [ t’ ])
        → (t⇒   : P / t ⇒ t’)
        → P / Et ⇒ Et’
```

The family of predicates SN n is defined inductively by the following rules—we allow ourselves set-notation at this semi-formal level:

$$\frac{t \in \mathsf{SN}\ n}{\lambda x t \in \mathsf{SN}\ n} \qquad \frac{t_1, t_2 \in \mathsf{SN}\ n}{(t_1, t_2) \in \mathsf{SN}\ n} \qquad \frac{}{\mathsf{next}\ t \in \mathsf{SN}\ 0} \qquad \frac{t \in \mathsf{SN}\ n}{\mathsf{next}\ t \in \mathsf{SN}\ (1+n)}$$

$$\frac{t \in \mathsf{SNe}\ n}{t \in \mathsf{SN}\ n} \qquad \frac{t' \in \mathsf{SN}\ n \qquad t\ \langle n \rangle\!\Rightarrow t'}{t \in \mathsf{SN}\ n}$$

The last two rules close SN under neutrals SNe, which is an instance of PNe with $P = \mathsf{SN}\ n$, and level-n *strong head expansion* $t\ \langle n \rangle\!\Rightarrow t'$, which is an instance of P-whe with also $P = \mathsf{SN}\ n$. We represent the inductive SN in Agda as a sized type (Hughes et al., 1996; Abel and Pientka, 2013) for the purpose of termination checking certain inductions on SN later. The assignment of sizes follows the principle that recursive invokations of SN within a constructor of SN $\{i\}$ must carry a strictly smaller size j : Size< i. The mutually defined relations SNe $n\ t$ (instance of PNe) and strong head reduction (shr) $t\ \langle n \rangle\!\Rightarrow t'$ just thread the size argument through. Note that there is a version i size $t\ \langle n \rangle\!\Rightarrow t'$ of shr that makes the size argument visible, to be supplied in case exp.

```
mutual
  data SN {i : Size}{Γ} : (n : ℕ) → ∀ {a} → Tm Γ a → Set where

    abs    : ∀ {j : Size< i} {a b n}{t : Tm (a :: Γ) b}
             → (t : SN {j} n t)
             → SN n (abs t)

    pair   : ∀ {j₁ j₂ : Size< i} {a b n t u}
             → (t : SN {j₁} n t) (u : SN {j₂} n u)
             → SN n {a ×̂ b} (pair t u)

    next0  : ∀ {a∞} {t : Tm Γ (force a∞)}
```

```
           → SN 0 {▶̂ a∞} (next t)

  next   : ∀ {j : Size< i} {a∞ n} {t : Tm Γ (force a∞)}
           → (t : SN {j} n t)
           → SN (1 + n) {▶̂ a∞} (next t)

  ne     : ∀ {j : Size< i} {a n t}
           → (n : SNe {j} n t)
           → SN n {a} t

  exp    : ∀ {j₁ j₂ : Size< i} {a n t t′}
           → (t⇒ : j₁ size t ⟨ n ⟩⇒ t′) (t′ : SN {j₂} n t′)
           → SN n {a} t

SNe              : ∀ {i : Size} {Γ a} (n : ℕ) → Tm Γ a → Set
SNe {i} n = PNe (SN {i} n)

_size_⟨_⟩⇒_  : ∀ (i : Size) {Γ a} → Tm Γ a → ℕ → Tm Γ a → Set
i size t ⟨ n ⟩⇒ t′ = SN {i} n / t ⇒ t′

_⟨_⟩⇒_          : ∀ {i : Size} {Γ a} → Tm Γ a → ℕ → Tm Γ a → Set
_⟨_⟩⇒_ {i} t n t’ = SN {i} n / t ⇒ t’
```

The SN-relations are antitone in the level n. This is one dimension of the Kripke worlds in our model (see next section).

```
mapSN  : ∀ {m n} → m ≤ℕ n → ∀ {Γ a}{t : Tm Γ a} → SN n t → SN m t


mapSNe : ∀ {m n} → m ≤ℕ n → ∀ {Γ a}{t    : Tm Γ a} → SNe n t       → SNe m t
map⇒   : ∀ {m n} → m ≤ℕ n → ∀ {Γ a}{t t’ : Tm Γ a} → t ⟨ n ⟩⇒ t’ → t ⟨ m ⟩⇒ t’
```

The other dimension of the Kripke worlds is the typing context; our notions are also closed under renaming (and even undoing of renaming). Besides renameSN, we have analogous lemmata renameSNe and rename⇒.

```
renameSN  : ∀ {n a Δ Γ} (ρ : Δ ≤ Γ) {t : Tm Γ a} →
            SN n t → SN n (rename ρ t)

fromRenameSN  : ∀{n a Γ Δ} (ρ : Δ ≤ Γ) {t : Tm Γ a} →
                SN n (rename ρ t) → SN n t
```

A consequence of fromRenameSN is that $t \in$ SN n iff $t\ x \in$ SN n for some variable x. (Consider $t = \lambda y.t'$ and $t\ x\ \langle n\rangle\!\Rightarrow t'[y/x]$.) This property is essential for the construction of the function space on sn sets (see next section).

```
absVarSN  : ∀{Γ a b n}{t : Tm (a :: Γ) (a →̂ b)} →
            app t (var zero) ∈ SN n → t ∈ SN n
```

6 Soundness

A well-established technique (Tait, 1967) to prove strong normalization is to model each type a as a set $\mathscr{A} = [\![a]\!]$ of sn terms. Each so-called semantic type $\mathscr{A}$ should contain the variables in order to interpret open terms by themselves (using the identity valuation). To establish the conditions of semantic types compositionally, the set $\mathscr{A}$ needs to be *saturated*, i. e., contain SNe (rather than just the variables) and be closed under strong head expansion (to entertain introductions).

As a preliminary step towards saturated sets we define sets of well-typed terms in an arbitrary typing context but fixed type, TmSet a. We also define shorthands for the largest set, set inclusion and closure under expansion.

```
TmSet : (a : Ty) → Set₁
TmSet a = {Γ : Cxt} (t : Tm Γ a) → Set

[⊤] : ∀{a} → TmSet a
[⊤] t = ⊤

_⊆_ : ∀{a} (A A′ : TmSet a) → Set
A ⊆ A′ = ∀{Γ}{t : Tm Γ _} → A t → A′ t

Closed : ∀ (n : ℕ) {a} (A : TmSet a) → Set
Closed n A = ∀{Γ}{t t’ : Tm Γ _} → t ⟨ n ⟩⇒ t’ → A t’ → A t
```

For each type constructor we define a corresponding operation on TmSets. The product is simply pointwise through the use of the projections.

```
_[×]_ :  ∀{a b} → TmSet a → TmSet b → TmSet (a ×̂ b)
(𝒜 [×] ℬ) t = 𝒜 (fst t) × ℬ (snd t)
```

For function types we are forced to use a Kripke-style definition, quantifying over all possible extended contexts Δ makes $\mathscr{A}$ [→] $\mathscr{B}$ closed under renamings.

```
_[→]_ : ∀{a b} → TmSet a → TmSet b → TmSet (a →̂ b)
(𝒜 [→] ℬ) {Γ} t = ∀{Δ} (ρ : Δ ≤ Γ) → ∀ {u} → 𝒜 u → ℬ (app (rename ρ t) u)
```

The TmSet for the later modality is indexed by the depth. The first two constructors are for terms in the canonical form next t, at depth zero we impose no restriction on t, otherwise we use the given set $\boldsymbol{A}$. The other two constructors are needed to satisfy the properties we require of our saturated sets.

```
data [▸] {a∞} (A : TmSet (force a∞)) {Γ} : (n : ℕ) → Tm Γ (▸̂ a∞) → Set where
  next0  :  ∀  {t : Tm Γ (force a∞)}                          → [▸] A  zero     (next t)
  next   :  ∀  {n}{t : Tm Γ (force a∞)}  (t : A t)            → [▸] A  (suc n)  (next t)
  ne     :  ∀  {n}{t : Tm Γ (▸̂ a∞)}      (n : SNe n t)        → [▸] A  n        t
  exp    :  ∀  {n}{t t’  : Tm Γ (▸̂ a∞)}
               (t⇒ : t ⟨ n ⟩⇒ t’)          (t : [▸] A n t’)   → [▸] A  n        t
```

The particularity of our saturated sets is that they are indexed by the depth, which in our case is needed to state the usual properties. In particular if a term belongs to a saturated set it is also a member of SN, which is what we need for strong normalization. In addition we require them to be closed under renaming, since we are dealing with terms in a context.

```
record IsSAT (n : ℕ) {a} (𝑨 : TmSet a) : Set where
  field
    satSNe    : SNe n ⊆ 𝑨
    satSN     : 𝑨     ⊆ SN n
    satExp    : Closed n 𝑨
    satRename : ∀ {Γ Δ} (ρ : Δ ≤ Γ) → ∀ {t} → 𝑨 t → 𝑨 (rename ρ t)

record SAT (a : Ty) (n : ℕ) : Set₁ where
  field
    satSet  : TmSet a
    satProp : IsSAT n satSet
```

For function types we will also need a notion of a sequence of saturated sets up to a specified maximum depth n.

```
SAT≤ : (a : Ty) (n : ℕ) → Set₁
SAT≤ a n = ∀ {m} → m ≤ℕ n → SAT a m
```

To help Agda's type inference, we also define a record type for membership of a term into a saturated set.

```
record _∈_ {a n Γ} (t : Tm Γ a) (𝒜 : SAT a n) : Set where
  constructor ↿_
  field ⇃_ : satSet 𝒜 t

_∈⟨_⟩_ : ∀ {a n Γ} (t : Tm Γ a) {m} (m≤n : m ≤ℕ n) (𝒜 : SAT≤ a n) → Set
t ∈⟨ m≤n ⟩ 𝒜 = t ∈ 𝒜 m≤n
```

Given the lemmas about SN shown so far we can lift our operations on TmSet to saturated sets and give the semantic version of our term constructors.

For function types we need another level of Kripke-style generalization to smaller depths, so that we can maintain antitonicity.

```
_⟦→⟧_ : ∀ {n a b} (𝒜 : SAT≤ a n) (ℬ : SAT≤ b n) → SAT (a →̂ b) n
𝒜 ⟦→⟧ ℬ = record
  { satSet  = λ t → ∀ m (m≤n : m ≤ℕ _) → (𝑨 m≤n [→] 𝑩 m≤n) t
  ; satProp = record
    { satSN  = CSN
  ; satSNe     = CSNe
  ; satExp     = CExp
  ; satRename  = CRename
    }
  }
  where
    module 𝒜 = SAT≤ 𝒜
    module ℬ = SAT≤ ℬ
    A   = 𝒜.satSet
    B   = ℬ.satSet

    C   :  TmSet (_ ⇾ _)
    C t = ∀ m (m≤n : m ≤ℕ _) → (A m≤n [→] B m≤n) t

    CSN   :  C ⊆ SN _
    CSN t  =  fromRenameSN suc (absVarSN
      (ℬ.satSN ≤ℕ.refl (t _ ≤ℕ.refl suc (𝒜.satSNe ≤ℕ.refl (var zero)))))
  CSNe : SNe _ ⊆ C
  CSNe n m m≤n ρ u =
    ℬ.satSNe m≤n (sneApp (mapSNe m≤n (renameSNe ρ n)) (𝒜.satSN m≤n u))

  CExp  : ∀{Γ}{t t' : Tm Γ _} → t ⟨ _ ⟩⇒ t' → C t' → C t
  CExp t⇒ t m m≤n ρ u =
    ℬ.satExp m≤n ((cong (appl _) (appl _) (map⇒ m≤n (rename⇒ ρ t⇒)))) (t m m≤n ρ u)

  CRename : {Γ Δ : List Ty} (ρ : Δ ≤ Γ) {t : Tm Γ _} → C t → C (rename ρ t)
  CRename =  λ ρ {t} t m m≤n ρ' {u} u →
    ≡.subst (λ t₁ → B {m} m≤n (app t₁ u)) (subst-• ρ' ρ t) (t m m≤n (ρ' •s ρ) u)
```

The proof of inclusion into SN first derives that app (rename suc t) (var zero) is in SN through the inclusion of neutral terms into 𝒜 and the inclusion of ℬ into SN, then proceeds to strip away first (var zero) and then (rename suc), so that we are left with the original goal SN n t. Renaming t with suc is necessary to be able to introduce the fresh variable zero of type a.

The types of semantic abstraction and application are somewhat obfuscated because they need to mention the upper bounds and the renamings.

```
⟦abs⟧  :  ∀   {n a b} {𝒜 : SAT≤ a n} {ℬ : SAT≤ b n} {Γ} {t : Tm (a :: Γ) b} →
              (∀  {m} (m≤n : m ≤ℕ n) {Δ} (ρ : Δ ≤ Γ) {u : Tm Δ a} →
                 u ∈⟨ m≤n ⟩ 𝒜 → (subst0 u (subst (lifts ρ) t)) ∈⟨ m≤n ⟩ ℬ)
          →  abs t ∈  (𝒜 ⟦→⟧ ℬ)
(⇃ ⟦abs⟧ {𝒜 = 𝒜}{ℬ = ℬ} t) m m≤n ρ u =
  SAT≤.satExp ℬ m≤n (β (SAT≤.satSN 𝒜 m≤n u)) (⇃ t m≤n ρ (↿ u))

⟦app⟧  :  ∀ {n a b}{𝒜 : SAT≤ a n}{ℬ : SAT≤ b n}{Γ}{t : Tm Γ (a ⇾ b)}{u : Tm Γ a}
          → t ∈  (𝒜 ⟦→⟧ ℬ) → u ∈⟨ ≤ℕ.refl ⟩ 𝒜 → app t u ∈⟨ ≤ℕ.refl ⟩ ℬ
⟦app⟧ {ℬ = ℬ} {u = u} (↿ t) (↿ u) =  ≡.subst (λ t → app t u ∈⟨ ≤ℕ.refl ⟩ ℬ) renId
                                         (↿ t _ ≤ℕ.refl id u)
```

The TmSet for product types is directly saturated, inclusion into SN uses a lemma to derive SN n t from SN n (fst t), which follows from 𝒜 ⊆ SN.

```
_⟦×⟧_ : ∀ {n a b} (𝒜 : SAT a n) (ℬ : SAT b n) → SAT (a ×̂ b) n
𝒜 ⟦×⟧ ℬ = record
  { satSet  = satSet 𝒜 [×] satSet ℬ
  ; satProp  = record
    {  satSNe      = CSNe
    ;   satSN       = CSN
```

```
      ; satExp    = CExp
      ; satRename = λ ρ x → satRename 𝒜 ρ (proj₁ x) , satRename ℬ ρ (proj₂ x)
      }
    }
    where
      A = satSet 𝒜
      B = satSet ℬ
      C :  TmSet _
      C = A [×] B

      CSNe          :  SNe _ ⊆ C
      CSNe n        =  satSNe 𝒜 (elim n fst)
                    ,  satSNe ℬ (elim n snd)

      CSN           :  C ⊆ SN _
      CSN (t , u)   =  bothProjSN (satSN 𝒜 t) (satSN ℬ u)

      CExp          :  ∀{Γ}{t t' : Tm Γ _} → t ⟨ _ ⟩⇒ t' → C t' → C t
      CExp t⇒ (t , u) = satExp 𝒜 (cong fst fst t⇒) t
                    ,  satExp ℬ (cong snd snd t⇒) u
```

Semantic introduction ⟦pair⟧ : $t_1 \in \mathscr{A} \to t_2 \in \mathscr{B} \to$ pair $t_1\ t_2 \in (\mathscr{A}$ ⟦×⟧ $\mathscr{B})$ and eliminations ⟦fst⟧ : $t \in (\mathscr{A}$ ⟦×⟧ $\mathscr{B}) \to$ fst $t \in \mathscr{A}$ and ⟦snd⟧ : $t \in (\mathscr{A}$ ⟦×⟧ $\mathscr{B}) \to$ snd $t \in \mathscr{B}$ for pairs are straightforward.

```
    ⟦pair⟧  :  ∀ {n a b} {𝒜 : SAT a n} {ℬ : SAT b n} {Γ} {t₁ : Tm Γ a} {t₂ : Tm Γ b}
               → t₁ ∈ 𝒜 → t₂ ∈ ℬ → pair t₁ t₂ ∈ (𝒜 ⟦×⟧ ℬ)
    ⇃ ⟦pair⟧ {𝒜 = 𝒜} {ℬ = ℬ} (↿ t) (↿ u)  =  satExp 𝒜 (βfst (satSN ℬ u)) t
                                           ,  satExp ℬ (βsnd (satSN 𝒜 t)) u

    ⟦fst⟧   :  ∀ {n a b} {𝒜 : SAT a n} {ℬ : SAT b n} {Γ} {t : Tm Γ (a ×̂ b)}
               → t ∈ (𝒜 ⟦×⟧ ℬ) → fst t ∈ 𝒜
    ⟦fst⟧ t  = ↿ (proj₁ (⇃ t))

    ⟦snd⟧   :  ∀ {n a b} {𝒜 : SAT a n} {ℬ : SAT b n} {Γ} {t : Tm Γ (a ×̂ b)}
               → t ∈ (𝒜 ⟦×⟧ ℬ) → snd t ∈ ℬ
    ⟦snd⟧ t = ↿ (proj₂ (⇃ t))
```

The later modality is going to use the saturated set for its type argument at the preceeding depth, we encode this fact through the type SATpred.

```
    SATpred : (a : Ty) (n : ℕ) → Set₁
    SATpred a zero    = ⊤
    SATpred a (suc n) = SAT a n

    SATpredSet : {n : ℕ}{a : Ty} → SATpred a n → TmSet a
    SATpredSet {zero}   𝒜  = [⊤]
    SATpredSet {suc n}  𝒜  = satSet 𝒜
```

Since the cases for [▶]_ are essentially a subset of those for SN, the proof of inclusion into SN goes by induction and the inclusion of 𝒜 into SN.

```
⟦▶⟧_ : ∀{n a∞} (𝒜 : SATpred (force a∞) n) → SAT (▶̂ a∞) n
⟦▶⟧_ {n} {a∞} 𝒜 = record
  { satSet = [▶] (SATpredSet 𝒜) n

  ; satProp = record
    { satSNe     = ne
    ; satSN      = CSN 𝒜
    ; satExp     = exp
    ; satRename  = CRen 𝒜
    }
  }
  where
    𝑪 : ∀ {n} (𝒜 : SATpred (force a∞) n) → TmSet (▶̂ a∞)
    𝑪 {n} 𝒜 = [▶] (SATpredSet 𝒜) n

    CSN : ∀ {n} (𝒜 : SATpred (force a∞) n) → 𝑪 {n} 𝒜 ⊆ SN n
    CSN 𝒜 next0        = next0
    CSN 𝒜 (next 𝒕)     = next (satSN 𝒜 𝒕)
    CSN 𝒜 (ne 𝒏)       = ne 𝒏
    CSN 𝒜 (exp t⇒ 𝒕)  = exp t⇒ (CSN 𝒜 𝒕)

    CRen :  ∀ {n} (𝒜 : SATpred (force a∞) n) → ∀ {Γ Δ} (ρ : Γ ≤ Δ) →
            ∀ {t} → 𝑪 {n} 𝒜 t → 𝑪 {n} 𝒜 (subst ρ t)
    CRen 𝒜 ρ next0        = next0
    CRen 𝒜 ρ (next 𝒕)     = next (satRename 𝒜 ρ 𝒕)
    CRen 𝒜 ρ (ne 𝒏)       = ne (renameSNe ρ 𝒏)
    CRen 𝒜 ρ (exp t⇒ 𝒕)  = exp (rename⇒ ρ t⇒) (CRen 𝒜 ρ 𝒕)
```

Following Section 3 we can assemble the combinators for saturated sets into a semantics for the types of $\lambda^{\blacktriangleright}$. The definition of ⟦_⟧_ proceeds by recursion on the inductive part of the type, and otherwise by well-founded recursion on the depth. Crucially the interpretation of the later modality only needs the interpretation of its type parameter at a smaller depth, which is then decreasing exactly when the representation of types becomes coinductive and would no longer support recursion.

```
⟦_⟧≤  : (a : Ty) {n : ℕ} → ∀ {m} → m ≤ℕ n → SAT a m

⟦_⟧_  : (a : Ty) (n : ℕ) → SAT a n
⟦ a →̂ b ⟧ n = ⟦ a ⟧≤ {n}  ⟦→⟧  ⟦ b ⟧≤ {n}
⟦ a ×̂ b ⟧ n = ⟦ a ⟧ n      ⟦×⟧  ⟦ b ⟧ n
⟦ ▶̂ a∞   ⟧ n = ⟦▶⟧ P n
  where
    P : ∀ n → SATpred (force a∞) n
    P zero     = _
    P (suc n)  = ⟦ force a∞ ⟧ n
```

Well-founded recursion on the depth is accomplished through the auxiliary definition ⟦_⟧≤ which recurses on the inequality proof. It is however straightforward to convert in and out of the original interpretation, or between different upper bounds.

```
in≤   : ∀ a {n m} (m≤n : m ≤ℕ n) → satSet (⟦ a ⟧   m)    ⊆ satSet (⟦ a ⟧≤ m≤n)
out≤  : ∀ a {n m} (m≤n : m ≤ℕ n) → satSet (⟦ a ⟧≤ m≤n) ⊆ satSet (⟦ a ⟧ m)

coerce≤ :  ∀ a {n n' m} (m≤n : m ≤ℕ n) (m≤n' : m ≤ℕ n')
           → satSet (⟦ a ⟧≤ m≤n) ⊆ satSet (⟦ a ⟧≤ m≤n')
```

As will be necessary later for the interpretation of next, the interpretation of types is also antitone. For most types this follows by recursion, while for function types antitonicity is embedded in their semantics and we only need to convert between different upper bounds.

```
map⟦_⟧ : ∀ a {m n} → m ≤ℕ n → satSet (⟦ a ⟧ n) ⊆ satSet (⟦ a ⟧ m)


map⟦ a →̂ b  ⟧  m≤n  t                = λ l l≤m ρ u → let l≤n = ≤ℕ.trans l≤m m≤n in
                coerce≤ b l≤n l≤m (t l l≤n ρ (coerce≤ a l≤m l≤n u))
map⟦ a ×̂ b  ⟧  m≤n  (t , u)          = map⟦ a ⟧ m≤n t , map⟦ b ⟧ m≤n u
map⟦ ▸̂ a∞  ⟧  m≤n  (ne n)           = ne (mapSNe m≤n n)
map⟦ ▸̂ a∞  ⟧  m≤n  (exp t⇒ t)       = exp (map⇒ m≤n t⇒) (map⟦ ▸̂ a∞ ⟧ m≤n t)
map⟦ ▸̂ a∞  ⟧ {m = zero}    m≤n  next0      = next0
map⟦ ▸̂ a∞  ⟧ {m = suc m}  ()   next0
map⟦ ▸̂ a∞  ⟧ {m = zero}    m≤n  (next _)  = next0
map⟦ ▸̂ a∞  ⟧ {m = suc m}  m≤n  (next t)   = next (map⟦ force a∞ ⟧ (pred≤ℕ m≤n) t)
```

Typing contexts are interpreted as predicates on substitutions. These predicates inherit antitonicity and closure under renaming. Semantically sound substitutions act as environments θ. We will need Ext to extend the environment for the interpretation of lambda abstractions.

```
⟦_⟧C : ∀ Γ {n} → ∀ {Δ} (σ : Subst Γ Δ) → Set
⟦ Γ ⟧C {n} σ = ∀ {a} (x : Var Γ a) → σ x ∈ ⟦ a ⟧ n

Map :  ∀ {m n} → (m≤n : m ≤ℕ n) →
       ∀ {Γ Δ} {σ : Subst Γ Δ} (θ : ⟦ Γ ⟧C {n} σ) → ⟦ Γ ⟧C {m} σ
Map m≤n θ {a} x = map⟦ a ⟧∈ m≤n (θ x)

Rename :  ∀ {n Δ Δ'} → (ρ : Ren Δ Δ') →
          ∀ {Γ}{σ : Subst Γ Δ} (θ : ⟦ Γ ⟧C {n} σ) →
          ⟦ Γ ⟧C (ρ •s σ)
Rename ρ θ {a} x = ↾ satRename (⟦ a ⟧ _) ρ (⇃ θ x)

Ext :  ∀ {a n Δ Γ} {t : Tm Δ a} → (t : t ∈ ⟦ a ⟧ n) →
       ∀ {σ : Subst Γ Δ} (θ : ⟦ Γ ⟧C σ) → ⟦ a :: Γ ⟧C (t ::s σ)
Ext t θ  (zero)   = t
Ext t θ  (suc x)  = θ x
```

The soundness proof, showing that every term of $\lambda^{\blacktriangleright}$ is a member of our saturated sets and so a member of SN, is now a simple matter of interpreting each operation in the language to its equivalent in the semantics that we have defined so far.

```
sound :  ∀ {n a Γ} (t : Tm Γ a) {Δ} {σ : Subst Γ Δ} →
           (θ : ⟦ Γ ⟧C {n} σ) → subst σ t ∈ ⟦ a ⟧ n
sound (var x) θ = θ x
sound (abs t) θ = ⟦abs⟧ {t = t} λ m≤n ρ u →
   ↿ in≤ _ m≤n (⇃ sound t (Ext (↿ out≤ _ m≤n (⇃ u)) (Rename ρ (Map m≤n θ))))
sound (app t u)  θ  = ⟦app⟧  (sound t θ)  (sound u θ)
sound (pair t u)  θ  = ⟦pair⟧  (sound t θ)  (sound u θ)
sound (fst t)      θ  = ⟦fst⟧   (sound t θ)
sound (snd t)      θ  = ⟦snd⟧   (sound t θ)
sound (t ∗ u)      θ  = ⟦∗⟧      (sound t θ)  (sound u θ)
sound {zero}  (next t)  θ  = ↿ next0
sound {suc n} (next t)  θ  = ↿ (next (⇃ sound t (Map n≤sn θ)))
```

The interpretation of next depends on the depth, at zero we are done, at suc n we recurse on the subterm at depth n, using antitonicity to Map the current environment to depth n as well. In fact without next we would not have needed antitonicity at all since there would have been no way to embed a term from a smaller depth into a larger one.

7 SN correctness

To complete our strong normalization proof we need to show that SN is included in the characterization of strong normalization as a well-founded predicate sn.

```
fromSN  :  ∀ {i} {Γ} {n : ℕ} {a} {t : Tm Γ a} →
             SN {i} n t → sn n t
```

The cases for canonical and neutral forms are straightforward, since no reduction can happen at the top of the expression and we cover the others through the induction hypotheses.

```
fromSNe  :  ∀ {i Γ n a} {t : Tm Γ a} →
              SNe {i} n t → sn n t

fromSN (ne n)         = fromSNe n
fromSN (abs t)        = abssn (fromSN t)
fromSN (pair t u)     = pairsn (fromSN t) (fromSN u)
fromSN next0          = next0sn
fromSN (next t)       = nextsn (fromSN t)
fromSN (exp t⇒ t₁)  = acc (expsn t⇒ t₁ (fromSN t₁))
```

The expansion case is more challenging instead, we can not in fact prove expsn by induction directly.

```
expsn  :  ∀ {i j Γ n a} {t th to : Tm Γ a} →
          i size t ⟨ n ⟩⇒ th → SN {j} n th → sn n th →
          t ⟨ n ⟩⇒β to → sn n to
```

We can see the problem by looking at one of the congruence cases, in particular reduction on the left of an application. There we would have $t\,u \in sn$, tht_1 and $t\beta t_2$, and need to prove $t_2\,u \in sn$. By induction we could obtain $t_2 \in sn$ but then there would be no easy way to obtain $t_2\,u \in sn$, since strong normalization is not closed under application.

The solution is to instead generalize the statement to work under a sequence of head reduction evaluation contexts. We represent such sequences with the type ECxt*, and denote their application to a term with the operator _[_]*.

```
expsnCxt  :  ∀ {i j Γ n a b} {t th to : Tm Γ a} →
             (Es : ECxt* Γ a b) → i size t ⟨ n ⟩⇒ th →
             SN {j} n (Es [ th ]*) → sn n (Es [ th ]*) →
             t ⟨ n ⟩⇒β to → sn n (Es [ to ]*)
expsn t⇒ t t t⇒β = expsnCxt [] t⇒ t t t⇒β
```

In this way the congruence cases are solved just by induction with a larger context.

```
expsnCxt E (cong (appl u) (appl .u) th⇒) th th (cong (appl .u) (appl .u) t⇒)
  = expsnCxt (appl u :: E) th⇒ th th t⇒
```

This generalization however affects the lemmata that handle the reduction cases, which also need to work under a sequence of evaluation contexts. Fortunately the addition of a premise $E[z] \in sn$, about an unrelated term z, allows to conveniently handle all the reductions that target the context.

```
β▶sn :  ∀ {n Γ b} {a∞ b∞} {z} {t : Tm Γ (force (a∞ ⇒ b∞))} {u : Tm Γ (force a∞)}
        (E : ECxt* Γ (▶̂ b∞) b) → sn (suc n) (E [ z ]*) →
        sn n t → sn n u → sn (suc n) (E [ next (app t u) ]*) →
        sn (suc n) (E [ next t ∗ next {a∞ = a∞} u ]*)


βfstsn :  ∀ {n Γ b} {a c} {z} {t : Tm Γ b} {u : Tm Γ a}
          (E : ECxt* Γ b c) → sn n (E [ z ]*) →
          sn n t → sn n u → sn n (E [ t ]*) →
          sn n (E [ fst (pair t u) ]*)


βsndsn :  ∀ {n Γ b} {a c} {z} {t : Tm Γ b} {u : Tm Γ a}
          (E : ECxt* Γ b c) → sn n (E [ z ]*) →
          sn n t → sn n u → sn n (E [ t ]*) →
          sn n (E [ snd (pair u t) ]*)


βsn :  ∀ {i n a b c Γ} {u : Tm Γ a} {t : Tm (a :: Γ) b}{z}
       (Es : ECxt* Γ b c) → sn n (Es [ z ]*) →
       sn n t → SN {i} n (Es [ subst0 u t ]*) → sn n u →
       sn n (Es [ app (abs t) u ]*)
```

8 Conclusions

In this paper, we presented a family of strongly-normalizing reduction relations for simply-typed lambda calculus with Nakano's modality for recursion. Using a similar stratification, Krishnaswami and Benton (2011a) have shown weak normalization using hereditary substitutions, albeit for a system without recursive types.

Our Agda formalization uses a saturated sets semantics based on an inductive notion of strong normalization. Herein, we represented recursive types as infinite type expressions and terms as intrinsically well-typed ones.

Our treatment of infinite type expressions was greatly simplified by adding an extensionality axiom for the underlying coinductive type to Agda's type theory. This would not have been necessary in a more extensional theory such as *Observational Type Theory* (Altenkirch et al., 2007) as shown in (McBride, 2009). Possibly *Homotopy Type Theory* (UnivalentFoundations, 2013) would also address this problem, but there the status of coinductive types is yet unclear.

For the future, we would like to investigate how to incorporate guarded recursive types into a dependently-typed language, and how they relate to other approaches like coinduction with sized types, for instance.

Acknowledgments. Thanks to Lars Birkedal, Ranald Clouston, and Rasmus Møgelberg for fruitful discussions on guarded recursive types, and Hans Bugge Grathwohl, Fabien Renaud, and some anonymous referees for useful feedback on the Agda development and a draft version of this paper. The first author acknowledges support by Vetenskapsrådet framework grant 254820104 (Thierry Coquand). This paper has been prepared with Stevan Andjelkovic's Agda-to-LaTeX converter.

References

Agda Wiki. Chalmers and Gothenburg University, 2.4 edn. (2014), http://wiki.portal.chalmers.se/agda

Abel, A.: Normalization for the simply-typed lambda-calculus in Twelf. In: Logical Frameworks and Metalanguages (LFM 04). Electronic Notes in Theoretical Computer Science, vol. 199C, pp. 3–16. Elsevier (2008)

Abel, A., Pientka, B.: Wellfounded recursion with copatterns: A unified approach to termination and productivity. In: Proc. of the 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'13. pp. 185–196. ACM Press (2013)

Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: 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. pp. 27–38. ACM Press (2013)

Abel, A., Vezzosi, A.: A formalized proof of strong normalization for guarded recursive types (long version and Agda sources) (Aug 2014), http://www.cse.chalmers.se/~abela/publications.html#aplas14

Altenkirch, T., McBride, C., Swierstra, W.: Observational equality, now! In: Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007. pp. 57–68. ACM Press (2007)

Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1683, pp. 453–468. Springer-Verlag (1999)

Atkey, R., McBride, C.: Productive coprogramming with guarded recursion. In: Proc. of the 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'13. pp. 197–208. ACM Press (2013)

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

Birkedal, L., Møgelberg, R.E.: Intensional type theory with guarded recursive types qua fixed points on universes. In: 28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013. pp. 213–222. IEEE Computer Society Press (2013)

Dybjer, P.: Inductive families. Formal Aspects of Computing 6(4), 440–465 (1994)

Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996. pp. 410–423 (1996)

Joachimski, F., Matthes, R.: Short proofs of normalization. Archive of Mathematical Logic 42(1), 59–87 (2003)

Krishnaswami, N.R., Benton, N.: A semantic model for graphical user interfaces. In: Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 45–57. ACM Press (2011a)

Krishnaswami, N.R., Benton, N.: Ultrametric semantics of reactive programs. In: Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada. pp. 257–266. IEEE Computer Society Press (2011b)

McBride, C.: Type-preserving renaming and substitution (2006), `http://strictlypositive.org/ren-sub.pdf`, unpublished draft

McBride, C.: Let's see how things unfold: Reconciling the infinite with the intensional. In: Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5728, pp. 113–126. Springer-Verlag (2009)

McBride, C.: Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In: Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010. pp. 1–12. ACM Press (2010)

Nakano, H.: A modality for recursion. In: 15th Annual IEEE Symposium on Logic in Computer Science (LICS 2000), 26-29 June 2000, Santa Barbara, California, USA, Proceedings. pp. 255–266. IEEE Computer Society Press (2000)

van Raamsdonk, F., Severi, P., Sørensen, M.H., Xi, H.: Perpetual reductions in lambda calculus. Information and Computation 149(2), 173–225 (1999)

Tait, W.W.: Intensional interpretations of functionals of finite type I. The Journal of Symbolic Logic 32(2), 198–212 (1967)

UnivalentFoundations: Homotopy type theory: Univalent foundations of mathematics. Tech. rep., Institute for Advanced Study (2013), `http://homotopytypetheory.org/book/`

Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 224–235. New Orleans (2003)