Thesis for the Degree of Licentiate of Philosophy

Guarded Recursive Types in Type Theory

Andrea Vezzosi

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Göteborg, Sweden 2015

Guarded Recursive Types in Type Theory

Andrea Vezzosi

Technical Report 144L
ISSN 1652-876X
Department of Computer Science and Engineering
Programming Logic Research Group

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31 772 1000

Printed at Chalmers Reproservice
Göteborg, Sweden 2015

Abstract

In total functional (co)programming valid programs are guaranteed to always produce (part of) their output in a finite number of steps.

Enforcing this property while not sacrificing expressivity has been challenging. Traditionally systems like Agda and Coq have relied on a syntactic restriction on (co)recursive calls, but this is inherently anti-modular.

Guarded recursion, first introduced by Nakano, has been recently applied in the coprogramming case to ensure totality through typing instead. The relationship between the consumption and the production of data is captured by a delay modality, which allows to give a safe type to a general fixpoint combinator.

This thesis consists of two parts. In the first we formalize, using the proof assistant Agda, a result about strong normalization for a small language extended with guarded recursive types. In the second we extend guarded recursive types to additionally ensure termination of recursive programs: the main result is a model based on relational parametricity for the dependently typed calculus we designed.

Keywords Dependent Type Theory, Guarded Recursion, Induction, Coinduction, Type-based termination

Acknowledgments

First of all I would like to thank Andreas Abel and Nils Anders Danielsson for the guidance and support during the development of the results in this thesis.

I would also like to thank Hans Bugge Grathwohl for all the time spent experimenting with implementations of guarded recursion, and Paolo Capriotti, Nicolai Kraus, Rasmus Ejlers Møgelberg, Bas Spitters and many others for all the discussions about type theory.

A special thanks goes to the Programming Logic group at Chalmers for being such a good place to conduct research, full of stimulating and welcoming people.

Introduction

Totality is a desirable property of programs: if we run a program we expect to eventually obtain a result. Totality is maybe even more important for programs that denote proofs through the Curry-Howard correspondence: we expect a proof to handle all the possible cases and to avoid unfounded reasoning.

Proof assistants based on intuitionistic type theory like Agda and Coq enforce totality by imposing operationally inspired restrictions on the programs they accept. Computation rules are kept relatively simple, and restrictions on recursive definitions ensure that reducing an expression will never end up in a loop. Reduction is kept terminating even in the presence of free variables, i.e. open computation.

The traditional checks allow definitions that roughly correspond to primitive recursion, with the addition of some heuristics like lexicographic ordering, in case of recursing over multiple arguments, or deep pattern matching. Traditional checks tend to get in the way of abstraction through e.g. higher order functions, especially when dealing with infinite data, like streams. Sized Types, as implemented in Agda, get to be more liberal by recording in the types some aspects of the computational behaviour of definitions through additional "Size" indices. A good intuition is that any form of recursion is reduced to well-founded induction on sizes [Abel and Pientka, 2013], however this intuition is impaired by additional syntactic restrictions motivated by the operational semantics: some definitions that are well-founded according to the ordering on sizes can nonetheless cause open computation to loop, so they get excluded by additional checks. On the other hand the computational behaviour is straightforward: each clause in a definition is used as a rewrite rule.

The operational nature of the restrictions described so far is not an accident: both Coq and Agda rely on normalization of terms during typechecking so open computation has to be terminating for typechecking to be decidable.

The line of research which currently goes under the name of Guarded Recursive Types started with [Nakano, 2000] by trying to characterize which functions have a fixed point even in the presence of uninhabited types, i.e., in a language which is logically consistent. He observed that the semantics of fixed points in Scott domains can be tweaked to not require a bottom element everywhere if we incorporate the notion of approximation in the types themselves rather than just their elements. In Scott domains the solution to a

recursive equation like $f = F(f)$ is obtained by taking the supremum of successive approximations $\bot, F(\bot), F(F(\bot)), \ldots$ where F is a map $S \to S$ from some domain S to itself. Such semantics require S to contain $\bot$ and the suprema of such chains, but if instead we consider a sequence $S_0 \supset S_1 \supset S_2 \supset ...$ of approximations of S and a corresponding family of functions $\{F_n : S_n \to S_{(n+1)}\}_n$ of approximations of F we can define the nth approximation of the fixed point to be $F_{(n-1)}(F_{(n-2)}(\ldots F_0(\bot)))$, assuming $\bot \in S_0$. Note that each F_n is required to go from S_n to $S_{(n+1)}$, i.e., provide more precision than it is given, by borrowing a term from corecursion we call such an F productive. In Guarded Recursive Types the productivity of F is internalized in its type $\blacktriangleright S \to S$ by the so-called later modality $\blacktriangleright$, whose semantics introduces the necessary one-step delay in the chain of approximations. We obtain then a general fixed point combinator $\mathsf{fix} : (\blacktriangleright A \to A) \to A$ which can be used to express recursion, both at the level of values and at the level of types. In particular even negative recursive types like $T \equiv (\blacktriangleright T) \to A$ have a meaning, as long as the negative occurrence is guarded by the $\blacktriangleright$ modality.

So far the guiding principle for Guarded Recursive Types has been the denotational semantics, in particular the one given by the topos of trees [Birkedal and Møgelberg] which cleanly captures the notions of approximation and well-foundedness. The resulting language accepts all the recursive definitions that can be expressed through fix and enforces totality by the fact that expressions like $\mathsf{fix}\,(\lambda\, x.\, x)$ are ill-typed. Such a close match to the denotational semantics comes with the burden of engineering an operational semantics that is suitable for normalization of expressions with free variables. In the first paper of this thesis, an extended version of [Abel and Vezzosi], we put Agda to the task of modeling reduction for a simply typed lambda calculus extended with guarded recursive types. Through the technique of saturated sets we obtain that if we replace the reduction relation with a family of approximations then each of those enjoys the strong normalization property. In particular we obtain that in case we reach a term of type $\blacktriangleright A$ we might need to block reduction, similarly to how thunking works for languages with lazy evaluation, but otherwise we can normalize our terms as usual. The paper is also a case study on conducting such formalizations in Agda.

Guarded Recursive Types have been applied to the case of corecursion and coinductive types [Atkey and McBride, 2013], i.e. generation of possibly infinite data, but a natural question is whether they can also handle the more traditional recursion on finite data, like trees. In the second paper of this thesis (previously unpublished) we find that it is indeed possible by introducing a modality that is dual to $\blacktriangleright$. There we also show how these modalities can be encoded with the language of Sized Types while still maintaining fix as the only source of recursion. We present a dependently typed calculus with these features and show its consistency through a model based on relational parametricity.

At the time of writing, ongoing work includes an implementation of guarded recursive types with decidable typechecking. The implementation is based on the cubical prototype [Cohen et al., 2013], which provides a typechecker and

interpreter for a core type theory with support for the univalence axiom [UnivalentFoundations, 2013]. There we can provide a propositional equality for the ▶ modality and coinductive types which identifies observationally equivalent values, e.g. bisimilar streams are equal. Additionally we plan to extend the result from the second paper with an operational semantics suitable for a typechecking algorithm.

Bibliography

A. Abel and B. Pientka. 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*, pages 185–196. ACM Press, 2013.

A. Abel and A. Vezzosi. A formalized proof of strong normalization for guarded recursive types. In *12th Asian Symposium on Programming Languages and Systems, APLAS 2014*.

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

L. Birkedal and R. E. Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *28th IEEE Symp. on Logic in Computer Science (LICS'13)*, pages 213–222.

C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. cubicaltt, 2013. URL `https://github.com/mortberg/cubicaltt`.

H. Nakano. A modality for recursion. In *Proc. of the 15th IEEE Symp. on Logic in Computer Science (LICS 2000)*, pages 255–266. IEEE Computer Soc. Press, 2000.

UnivalentFoundations. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013. URL `http://homotopytypetheory.org/book/`.

Chapter 1

Strong Normalization for Guarded Recursive Types

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 `next`s, 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).

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

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

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

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 $\omega = (\lambda x.\, x \circledast (\mathsf{next}\ x))$ 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.

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

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

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

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 $[\![\top]\!]$. Then the interpretation $[\![A]\!]_n$ of closed type A at depth n can be given recursively as follows, using the Kripke construction at function types:

$$
\begin{array}{lll}
[\![A \times B]\!]_n & = [\![A]\!]_n \; [\![\times]\!] \; [\![B]\!]_n & \mathscr{A} \; [\![\times]\!] \; \mathscr{B} = \{t \mid \mathsf{fst}\, t \in \mathscr{A} \text{ and } \mathsf{snd}\, t \in \mathscr{B}\} \\
[\![A \to B]\!]_n & = \bigcap_{n' \le n} ([\![A]\!]_{n'} \; [\![\to]\!] \; [\![B]\!]_{n'}) & \mathscr{A} \; [\![\to]\!] \; \mathscr{B} = \{t \mid t\, u \in \mathscr{B} \text{ for all } u \in \mathscr{A}\} \\
[\![\blacktriangleright A]\!]_0 & = [\![\blacktriangleright]\!] \, [\![\top]\!] & [\![\top]\!] \qquad = \{t \mid t \text{ term}\} \\
[\![\blacktriangleright A]\!]_{n+1} & = [\![\blacktriangleright]\!] [\![A]\!]_n & [\![\blacktriangleright]\!] \, \mathscr{A} \quad = \overline{\{\mathsf{next}\, t \mid t \in \mathscr{A}\}} \\
[\![\mu X A]\!]_n & = [\![A[\mu X A/X]]\!]_n & (\overline{\mathscr{A}} \text{ is weak head expansion closure of } \mathscr{A})
\end{array}
$$

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, $[\![A]\!]_n$ is defined by lexicographic induction on $(n, \mathsf{size}(A))$, where $\mathsf{size}(A)$ is the number of type constructor symbols $(\times, \to, \mu)$ 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 vincinity 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 $\hat{\to}$, 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 $\hat{\to}$ 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 $\hat{\to}$ 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 $\lambda^{\blacktriangleright}$.

$$\begin{aligned} \mathsf{delay} &: \mathsf{Ty} \to \infty\mathsf{Ty} \\ \mathsf{force} &: \infty\mathsf{Ty} \to \mathsf{Ty} \end{aligned}$$

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

```
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 = ▶̂ 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.

```
▶_ : Ty → Ty
▶ a = ▶̂ delay a
```

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

```
_⇒_ : (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 $\mathsf{fst}\ (\lambda x x)$. 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 $x : \mathsf{Var}\ \Gamma\ a$ by de Brujin indices, i. e., positions in a typing context $\Gamma : \mathsf{Cxt}$, 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 $\forall$ prefixes bindings in a function type, the types of the bound variables may be omitted. Thus, $\forall\{\Gamma\ a\} \to \mathrm{A}$ is short for $\{\Gamma : \mathsf{Cxt}\}\{a : \mathsf{Ty}\} \to \mathrm{A}$.

Terms $t : \mathsf{Tm}\ \Gamma\ a$ are indexed by a typing context Γ and their type a, guaranteeing well-typedness and well-scopedness. In the following data type definition, $\mathsf{Tm}\ (\Gamma : \mathsf{Cxt})$ 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 $▶\_ : \mathsf{Ty} \to \mathsf{Ty}$:

```
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 $a\infty\ b\infty : \infty\mathsf{Ty}$ rather than $a\ b : \mathsf{Ty}$, 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 $\equiv$ of Martin-Löf Type Theory is intensional in the sense that only objects with the same *code* (modulo definitional equality) are considered equal. Thus, $\equiv$ 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 $\equiv/\infty\cong$ that follows the structure of $\mathsf{Ty}/\infty\mathsf{Ty}$ (hence, we reuse the constructor names $\hat{\times}$, $\hat{\to}$, and $\hat{\blacktriangleright}$).

```
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 $\equiv$ we do not get a generic substitution principle for $\cong$, 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 $\mathsf{Tm}\ \Gamma\ a$ to $\mathsf{Tm}\ \Gamma\ b$ if $a \cong b$, which would require us to build type equality at least into $\mathsf{Var}\ \Gamma\ a$. 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 $c\infty : \infty\mathsf{Ty}$ 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:

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

We construct an auxiliary type $\mathsf{Fix}\ a$ 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 $\mathsf{fix}\ (\lambda y.t)$ 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.

$$\mathsf{mapS} = \lambda f.\ \mathsf{fix}\ (\lambda mapS.\ \lambda s.\ (f\ s, mapS \circledast \mathsf{tail}\ s))$$

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 $t \langle n \rangle \Rightarrow \beta\ t'$ 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 $\Gamma\ \Delta\ a\ b\ n\ n'$ produces a term Ct : Tm $\Gamma\ b$ of depth n' if filled with a term t : Tm $\Delta\ a$ of depth n. The depth is unchanged except for the case next, which increases the depth by 1. Thus, $t \langle n \rangle \Rightarrow \beta\ t'$ 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 $t\ \langle n\rangle{\Rightarrow}\beta\ t'$ 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 $\mathsf{fst}\,(\lambda x.x)$ 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 $j : \mathsf{Size}< i$. 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 $t \in \mathsf{SN}\ n$ iff $t\ x \in \mathsf{SN}\ n$ 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 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 $\mathscr{A}$ and the inclusion of $\mathscr{B}$ 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 $\mathscr{A} \subseteq$ 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)   = bothProj SN (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 (β fst (satSN 𝒜 u)) t
                                     , satExp ℬ (β snd (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 $\mathscr{A}$ 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 $[\![\_]\!]\leq$ 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 ∈ ⟦ 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 $t\,u \in sn$, $th t_1$ 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⇒β = expsnCxt [] 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)

Chapter 2

Total (Co)Programming with Guarded Recursion

Total (Co)Programming with Guarded Recursion

Andrea Vezzosi
Chalmers Univerisity of Technology
vezzosi@chalmers.se

December 15, 2015

Abstract

In total functional (co)programming valid programs are guaranteed to always produce (part of) their output in a finite number of steps.

Enforcing this property while not sacrificing expressivity has been challenging. Traditionally systems like Agda and Coq have relied on a syntactic restriction on (co)recursive calls, but this is inherently anti-modular.

Guarded recursion, first introduced by Nakano, has been recently applied in the coprogramming case to ensure totality through typing instead. The relationship between the consumption and the production of data is captured by a delay modality, which allows to give a safe type to a general fixpoint combinator.

Here we show how that approach can be extended to additionally ensure termination of recursive programs, through the introduction of a dual modality and the interaction between the two. We recast the fixpoint combinator as well-founded induction with additional invariance guarantees, which we justify through a parametric model.

We obtain a dependently typed calculus which modularly ensures totality for both coinductive and (finitely branching) inductive types.

1 Introduction

The standard termination checkers of systems like Coq or Agda rely on syntactic checks, and while they employ additional heuristics, they mainly ensure that the recursive calls have structurally smaller arguments. This is the case for the map function on lists:

$$\begin{aligned}&\mathsf{map} : (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B\\ &\mathsf{map}\ f\ [\,] \quad\quad = [\,]\\ &\mathsf{map}\ f\ (x :: xs) = f\ x :: \mathsf{map}\ f\ xs\end{aligned}$$

Here the argument xs of $\mathsf{map}\ f\ xs$ is manifestly smaller than the original $(x :: xs)$ because it is a direct "subtree", so the definition is accepted.

However, such a syntactic check gets in the way of code reuse and compositionality. This is especially the case when using higher-order combinators, because they can appear between a recursive call and the actual data it will process. The following definition of a mapping function for rose trees is an example of this.

```
data RoseTree A = Node A (List (RoseTree A))
mapRT : (A → B) → RoseTree A → RoseTree B
mapRT f (Node a ts)
  = Node (f a) (map (λ t. mapRT f t) ts)
```

The definition of mapRT is not accepted because syntactically t has no relation to Node a ts, we need to pay attention to the semantics of map to accept this definition as terminating, in fact a common workaround is to essentially inline the code for map[1].

```
mapRT : (A → B) → RoseTree A → RoseTree B
mapRT f (Node a ts) = Node (f a) (mapmapRT f ts)
  where
    mapmapRT f []       = []
    mapmapRT f (t :: ts) = mapRT f t :: mapmapRT f ts
```

Here we have spelled out the nested recursion as a function mapmapRT, and the checker can figure out that mapRT is only going from the call

```
mapRT f (Node a (t :: ts))
```

to mapRT f t, so it is accepted.

Such a system then actively fights abstraction, and offers few recourses other than sticking to some specific "design patterns" for (co)recursive definitions.

Recently there has been a fair amount of research into moving the information about how functions consume and produce data to the type level, so that totality checking can be more modular [Atkey and McBride, 2013, Møgelberg, Abel and Pientka]. In particular the previous work on Guarded Recursion [Atkey and McBride, 2013, Møgelberg] has handled the case of ensuring totality for corecursion, i.e. manipulating infinite data. The issue of ensuring totality in a modular way for recursion, over well-founded data, was however left open. We address that problem by presenting a calculus that supports both corecursion and recursion with a single guarded fixed point combinator.

We make the following contributions:

- We define a core type system which supports total programming by combining guarded recursion, existential and universal Time quantification and a new delay modality. From these primitives we can derive both (finitely branching) initial algebras and final coalgebras, i.e. inductive and coinductive datatypes. The type system is an extension of Martin-Löf Type Theory.

[1] If the definition of map is available, Coq will attempt this automatically

- We give an interpretation of the system into a relationally parametric model of dependent type theory in reflexive graphs [Atkey et al., a], drawing a parallel between Time quantification and parametric polymorphism.
- In particular we show how existential Time quantification supporting representation independence can be defined in a predicative system by truncating Σ types to the universe of discrete reflexive graphs.

1.1 Guarded Recursion

The basic principle of Guarded Recursion is to keep track of the termination information in the types. Going back to the rose tree example we can give the data constructor Node a more precise type.

$$\mathsf{Node} : A \to (\mathsf{List}\ (\diamond\ \mathsf{RoseTree}\ A)) \to \mathsf{RoseTree}\ A$$

We use the modality $\diamond$ to explicitly mark the recursive occurrence of RoseTree A. Because of this the subtrees have a type that differs from the root, and we will be able to keep track of which recursive calls are valid.

Once we've done that we can write mapRT by reusing map.

$$\begin{aligned}&\mathsf{mapRT} : (A \to B) \to \mathsf{RoseTree}\ A \to \mathsf{RoseTree}\ B\\ &\mathsf{mapRT}\ f\ (\mathsf{Node}\ a\ ts) = \mathsf{Node}\ a\ (\mathsf{map}\ (\lambda\, t.\, \mathsf{mapRT}^{\diamond}\ f\ t)\ ts)\end{aligned}$$

Here the call $\mathsf{mapRT}^{\diamond}$ stands for an implicit use of a guarded fixpoint combinator we will introduce in Section 1.2, so it gets type

$$(A \to B) \to \diamond\ \mathsf{RoseTree}\ A \to \diamond\ \mathsf{RoseTree}\ B$$

i.e. it can only handle smaller trees but also produces smaller trees, so the definition is well-typed. If we tried to write a circular definition like

$$\mathsf{mapRT}\ f\ ts = \mathsf{mapRT}^{\diamond}\ f\ ts$$

the types would not match because ts is of type RoseTree A rather than $\diamond$ RoseTree A.

We can explain the meaning of $\diamond\ A$ using a countdown metaphor: if we consider that we have a finite amount of time i left to complete our computation, a value of type $\diamond\ A$ tells us that there is a future time j at which we have a value of type A. So in particular, even if we take A to be the unit type $\top$, the type $\diamond\ \top$ is not necessarily inhabited, because we might have no time left. Considering that our times are counting down to 0 we say that future times are smaller.

In general we will have to deal with data on unrelated timelines, so instead of $\diamond$ we have a family of type constructors $\diamond^{\kappa}$ indexed by clock variables κ, which are introduced by the quantifiers $\forall\,\kappa$ and $\exists\,\kappa$. Often we will leave the clocks implicit at the level of terms, however where appropriate we will use

λ abstraction and application for $\forall\,\kappa$, and the constructor $(\cdot,\cdot)$ and pattern matching for $\exists\,\kappa$.

$$(\cdot,\cdot) : \forall\,\kappa.\,(A\ \to\ \exists\,\kappa.\,A)$$

The proper type of Node will then have a clock indexing the result type, which becomes $\mathsf{RoseTree}^{\kappa}\ A$ i.e. the type of trees bounded by the clock κ.

$$\mathsf{Node} : \forall\,\kappa.\,A\ \to\ \mathsf{List}\ (\diamond^{\kappa}\ \mathsf{RoseTree}^{\kappa}\ A)\ \to\ \mathsf{RoseTree}^{\kappa}\ A$$

1.2 Fixedpoint combinators

So far we have seen directly recursive definitions, here we show the fixed point combinator they are based on and how it is derived from a more general one.

$$\begin{aligned}&\mathsf{fix}_{\diamond} : (\forall\,\kappa.\,(\diamond^{\kappa}\ A\ \to\ \diamond^{\kappa}\ B)\ \to\ A\ \to\ B)\\ &\qquad\to\ \forall\,\kappa.\,A\ \to\ B\end{aligned}$$

The combinator $\mathsf{fix}_{\diamond}$ is what we have implicitly used so far, and as we have seen it is a good match for the definition of functions that recurse on one of their arguments.

More generally though we also want to support cases of productive value recursion, like Haskell's $\mathsf{ones}\ =\ 1 : \mathsf{ones}$, as in the previous works on guarded recursion. For this purpose we make use of the modality $\Box^{\kappa}$, the dual of $\diamond^{\kappa}$, to guard the recursion.

$$\mathsf{fix} : (\forall\,\kappa.\,\Box^{\kappa}\ A\ \to\ A)\ \to\ \forall\,\kappa.\,A$$

Using the time metaphor, a value of type $\Box^{\kappa}\ A$ gives us an A at every time in the future.

With a suitable type Stream^{κ} we can then define ones.

$$\begin{aligned}&\mathsf{cons} : \forall\,\kappa.\,A\ \to\ \Box^{\kappa}\ \mathsf{Stream}^{\kappa}\ A\ \to\ \mathsf{Stream}^{\kappa}\ A\\ &\mathsf{ones} : \forall\,\kappa.\,\mathsf{Stream}^{\kappa}\ \mathsf{Nat}\\ &\mathsf{ones}\ =\ \mathsf{fix}\ (\lambda\,\kappa\ xs.\,\mathsf{cons}\ 1\ xs)\end{aligned}$$

Other examples involving $\Box^{\kappa}$ can be found in previous work that focuses on coinduction [Atkey and McBride, 2013] [Møgelberg]. An important difference is that in the present work $\Box^{\kappa}$ is no longer a full applicative functor. In fact while we do still support $\circledast$,

$$\circledast : \forall\,\kappa.\,\Box^{\kappa}\ (A\ \to\ B)\ \to\ \Box^{\kappa}\ A\ \to\ \Box^{\kappa}\ B$$

we do not support pure in general, e.g. not at type $\diamond^{\kappa}\ A$, but we do for those types that do not mention κ:

$$\mathsf{pure} : \forall\,\kappa.\,A\ \to\ \Box^{\kappa}\ A \qquad \kappa \notin \mathsf{fv}(A)$$

In the language of Section 2 such an operation will be implementable for a more general class of types, which will correspond to those antitone with regard to time in the model of Section 4.

The interface of $\diamond^\kappa$ is composed of two operations, $\star$ and extract.

If we have a function at any time in the future $\square^\kappa\ (A\ \to\ B)$ and an argument at some time in the future $\diamond^\kappa\ A$, we can apply one to the other with the combinator $\star$,

$$\star : \forall\ \kappa\,.\ \square^\kappa\ (A\ \to\ B)\ \to\ \diamond^\kappa\ A\ \to\ \diamond^\kappa\ B$$

which in particular can be used to define $\mathsf{fix}_\diamond$ in terms of fix

$$\mathsf{fix}_\diamond\ f\ =\ \mathsf{fix}\ (\lambda\ \kappa\ r\,.\ f\ \kappa\ (\lambda\ x\,.\ r\ \star_\kappa\ x))$$

Lastly, as a dual of pure, we have extract,

$$\mathsf{extract} : \forall \kappa.\ \diamond^\kappa A \to A \qquad \kappa \notin \mathsf{fv}(A)$$

which allows to get values out of the $\diamond^\kappa$ modality as long as their type is independent of the clock κ.

1.3 Well-founded unfolding

There are interesting uses of $\diamond^\kappa$ even when not directly associated with a datatype, an example of this is the unfold function for List. We use $\top$ for the type with one element, and $+$ for the sum type with constructors inl and inr.

$$\begin{aligned}
&\mathsf{unfold} : (\forall\ \kappa\,.\ S\ \to\ \top + (A \times \diamond^\kappa\ S))\ \to \\
&\qquad \forall\ \kappa\,.\ S\ \to\ \mathsf{List}\ A \\
&\mathsf{unfold}\ f\ s\ =\ \mathsf{case}\ f\ s\ \mathsf{of} \\
&\quad \mathsf{inl}\ _\ \ \ \ \to\ [\,] \\
&\quad \mathsf{inr}\ (a, s')\ \to\ a\ ::\ \mathsf{extract}\ (\mathsf{unfold}^\diamond\ f\ s')
\end{aligned}$$

We can guarantee termination of unfold because the type f requires it to show the clock has not run out of time in the inr case, so that we can safely recurse on s'. Accepting unfold as terminating would be challenging for a syntactic check, since the relationship between s' and s depends on the semantics of f. In fact unfolding a constant function f that always returns $\mathsf{inr}\ (a, s')$ would never reach the base case. The $\diamond^\kappa$ modality prevents such an f because there's no way to always guarantee that the clock κ has remaining time left. The use of extract is justified if we assume that A does not mention κ.

We can make use of the S argument to acquire information about κ like in the following example. Using unfold and a type of bounded natural numbers we define the function downfrom, which returns a list counting down to 1.

$$\begin{aligned}
&\mathsf{Zero} : \forall\ \kappa\,.\ \mathsf{Nat}^\kappa \\
&\mathsf{Suc}\ \ : \forall\ \kappa\,.\ \diamond^\kappa\ \mathsf{Nat}^\kappa\ \to\ \mathsf{Nat}^\kappa
\end{aligned}$$

```
downfrom : ∀ κ. Nat^κ → List (∃ κ. Nat^κ)
downfrom =
  unfold (λ κ n → case n of
                    Zero   → inl _
                    Suc n' → inr ((κ, n), n'))
```

The existential quantification allows us to forget the time information of the naturals in the list, so that we do not have to keep it synchronized with the clock κ given to the function we unfold.

1.4 Inductive types

In previous work on guarded recursion, coinductive types were obtained by universal clock quantification over their guarded variant, e.g. $\forall\,\kappa.\,\mathsf{Stream}^\kappa\,\mathsf{Nat}$ would be the type of coinductive streams. In the present work we are able to dualize that result and obtain inductive types by existential quantification of the guarded variant, e.g. $\exists\,\kappa.\,\mathsf{Nat}^\kappa$.

For now we will assume type-level recursion, in Section 3 we will reduce it to uses of fix on a universe of types and construct initial algebras for suitable functors.

Here we show the natural numbers as the simplest example. The guarded version Nat^κ is defined using a sum type, to encode the two constructors, and inserting $\triangleright^\kappa$ in front of the recursive occurrence of Nat^κ.

```
Nat^κ  = ⊤ + ▷^κ Nat^κ
Zero^κ : Nat^κ
Zero^κ = inl tt
Suc^κ  : ▷^κ Nat^κ → Nat^κ
Suc^κ  = λ n. inr n
```

Now we can bind the clock variable with an existential to define

```
Nat = ∃ κ. Nat^κ
```

and show that Nat supports the expected iterator.

```
fold : A → (A → A) → Nat → A
fold z f (κ, n) = fix_▷ (λ κ r n →
  case n of
    inl _ → z
    inr n → f (extract (r n)))
  κ n
```

However it is not enough to package the clock to get the right type, for example we risk having too many Zeros if we can tell the difference between $(\kappa, \mathsf{inl}\ \mathsf{tt})$ and $(\kappa', \mathsf{inl}\ \mathsf{tt})$ for two different clocks κ and κ'.

The key idea is that values of type $\exists\,\kappa.\,A$ must keep abstract the specific clock they were built with, exactly like weak sums in System F. Intuitively Nat

will not be the initial algebra of $(\top +)$ unless $\mathsf{Nat} \cong \top + \mathsf{Nat}$ holds, so we must be able to support both an interface and an equational theory where clocks play no role.

In the calculus we will internalize this invariance over the packaged clock as type isomorphisms that are justified by a parametric model (Section 4). In the specific case of Nat, both $(\kappa, \mathsf{inl}\ \mathsf{tt})$ and $(\kappa', \mathsf{inl}\ \mathsf{tt})$ get sent to $\mathsf{inl}\ \mathsf{tt}$ by the isomorphism, so we can conclude they are equal.

Once we scale up to a dependently typed language we will also be able to implement an induction principle in terms of fix in Section 3. However before that we will describe our calculus, where we directly manipulate time values instead of clocks to express programs more directly, rather than introducing additional combinators.

2 From Clocks to Time

The notation we have used in the examples so far is closely modeled on the one used in [Atkey and McBride, 2013] and [Møgelberg]. In particular clocks are a convenient abstraction over explicitly handling time values, since we can use the same clock to refer to different amounts of type depending on the context.

We can think of clock variables as indices into an environment of values of type Time. This environment however is not passed untouched to every subexpression, or simply added to by binders, it also gets updated at the index κ by the modalities $\Box^\kappa$ and $\Diamond^\kappa$ for the scope of their arguments. So the same clock variable represents different time values in different parts of a type expression.

To clarify the flow of Time we instead adopt a syntax inspired by Sized Types [Abel and Pientka]. It will also offer more flexibility in the dependent case. In fact the first step is to add to the dependently typed language of Figure 1 a new type Time, together with inequality $(i\ j : \mathsf{Time}) \vdash i \leqslant j$, zero $0 : \mathsf{Time}$, successor $\uparrow : \mathsf{Time} \to \mathsf{Time}$ and a maximum operator $\sqcup : \mathsf{Time} \to \mathsf{Time} \to \mathsf{Time}$ (Figure 2).

Universal quantification over Time is simply done with a dependent function type, and the $\Box^\kappa$ modality is a corresponding bounded version. As a result the type $\forall\, \kappa.\, \Box^\kappa\, \mathsf{Nat}^\kappa$ becomes $(i : \mathsf{Time}) \to (j : \mathsf{Time}) \to \uparrow j \leqslant i \to \mathsf{Nat}^{\,j}$, where the smaller time j is explicitly mentioned and passed as the time parameter of the guarded natural numbers type.

In the following we use the shorthand $\forall i.\ A$ in place of $(i : \mathsf{Time}) \to A$ and $\forall j < i.\ A$ in place of $(j : \mathsf{Time}) \to \uparrow j \leqslant i \to A$, and so we also omit the inequality proof in abstractions and applications, writing $\lambda\, j$ in place of $\lambda\, j\ (p : \uparrow j \leqslant i)$ and $f\ j$ in place of $f\ j\ p$.

As an example the operator $\circledast$ is given the following type and implementation:

$$
\begin{aligned}
&(\circledast) : \forall\, i.\, (\forall\, j < i.\, A \to B) \\
&\qquad \to (\forall\, j < i.\, A) \to (\forall\, j < i.\, B) \\
&f \circledast_i x = \lambda\, j \to f\ j\ (x\ j)
\end{aligned}
$$

Given a smaller time j we get the values of f and x at that time and apply one to the other.

With this formulation it is also easy to see how to generalize $\circledast$ to the dependent case since we can get hold of j and pass that to x to obtain a valid argument for the dependent type B:

$$
\begin{aligned}
&(\circledast^{\Pi}) : \forall\, i\,.\, (\forall\, j < i\,.\, (x : A) \;\to\; B\; x) \\
&\qquad \to\; (x : \forall\, j < i\,.\, A) \;\to\; (\forall\, j < i\,.\, B\; (x\; j)) \\
&f \;\circledast^{\Pi}_{i}\; x \;=\; \lambda\, j \;\to\; f\; j\; (x\; j)
\end{aligned}
$$

However the existential quantification $\exists\, i.\, A$ for Time, shown in Figure 5, has to be distinct from a plain Σ type because allowing an unrestricted first projection would let us observe the specific Time contained. The result type of the case expression for $\exists\, i.\, A$ instead is restricted to belong to the universe U, which lacks a code for Time itself (Figure 4). We can specialize the case expression to the non-dependent case and implement an $\mathsf{uncurry}$ combinator:

$$
\begin{aligned}
&\mathsf{uncurry} : (\forall\, i.\, A \;\to\; \mathrm{El}\; B) \;\to\; (\exists\, i\,.\, A) \;\to\; \mathrm{El}\; B \\
&\mathsf{uncurry}\; f\; x \;=\; \mathsf{case}\; x\; \mathsf{of}\; (i, a) \;\to\; f\; i\; a
\end{aligned}
$$

where we can pattern match on (i, a) because the return type belongs to the universe U. We also add a form of η expansion for $\exists\, i$, which will be necessary to ensure the proper computational behaviour for induction.

Dually to $\Box^{\kappa}$ we have that $\Diamond^{\kappa}$ corresponds to a bounded $\exists$ and we allow similar shorthands $\exists\, j < i.\, A$ for $\exists\, j\,.\, \uparrow j \leqslant i \times A$ and (j, a) for (j, p, a). As an example we show the implementation of $\star$:

$$
\begin{aligned}
&(\star) : \forall\, i\,.\, (\forall\, j < i\,.\, A \;\to\; \mathrm{El}\; B) \\
&\qquad \to\; (\exists\, j < i\,.\, A) \;\to\; \mathrm{El}\; (\exists\, j < i\,.\, B) \\
&f \;\star_i\; (j, x) \;=\; (j, f\; j\; x)
\end{aligned}
$$

We can also implement $\mathsf{extract}$ for $A : \mathrm{U}$:

$$
\begin{aligned}
&\mathsf{extract} : (A : \mathrm{U}) \;\to\; \forall\, i\,.\, (\exists\, j < i\,.\, \mathrm{El}\; A) \;\to\; \mathrm{El}\; A \\
&\mathsf{extract}\; A\; i\; (j, a) \;=\; a
\end{aligned}
$$

The fix combinator, shown in Figure 3, is taken as a primitive principle of well-founded induction on Time. The notation $A\,[i]$ stands for a type with i occurring free, so that $A\,[j]$ has instead all those occurrences replaced by j. The equality rule for $\mathsf{fix}\; f\; i$ describes it as the unique function with this unfolding,

$$
\mathsf{fix}\; f\; i \;=\; f\; i\; (\mathsf{guard}_{\Box}\; (\mathsf{fix}\; f)\; i)
$$

however such an equality, which is justified by the model, would lead to loss of strong normalization if used unrestricted as a computation rule, making type-checking undecidable. We discuss possible solutions in Section 6.

As anticipated in Section 1.4 we internalize with isomorphisms the properties of invariance with regard to Time of our language. The isomorphisms of Figure

7 describe how the Time quantifiers commute with the other connectives and enrich the equational theory. We take as example the pair of $\mathsf{guard}_\diamond$ and $\mathsf{force}_\diamond$:

$$\begin{aligned}
&\mathsf{guard}_\diamond : (\exists\, i\,.\, A\ i) \ \to\ \exists\, i\,.\, \exists\, j < i.\ A\ j\\
&\mathsf{guard}_\diamond\ (i, a) \ =\ (\uparrow i, i, a)\\
&\mathsf{force}_\diamond : (\exists\, i\,.\, \exists\, j < i.\ A\ j) \ \to\ \exists\, i\,.\, A\ i\\
&\mathsf{force}_\diamond\ (i, j, a) \ =\ (j, a)
\end{aligned}$$

the β and η rules for $\exists\ i$ are not enough to show that they are an isomorphism, in particular they only allow us to conclude that

$$\mathsf{guard}_\diamond\ (\mathsf{force}_\diamond\ (i, j, a)) \ =\ (\uparrow j, j, a)$$

but having imposed that $\mathsf{guard}_\diamond$ and $\mathsf{force}_\diamond$ form an isomorphism we can additionally deduce that

$$(i, j, a) \ =\ (\uparrow j, j, a)$$

showing that the packaged times i and $\uparrow j$ are in fact irrelevant in that position.

In the case of $\forall\ i$ we additionally have the usual isomorphisms that can be implemented for dependent functions:

$$\begin{array}{rl}
\forall\, i.\ A \times B \cong \forall\, i.\ A \times \forall\, i.\ B & \\
\forall\, i.\ (x : A) \ \to\ B \cong (x : A) \ \to\ \forall\, i\,.\, B & i \notin \mathsf{fv}(A)\\
\forall\, i.\ \forall\, j.\ A \cong \forall\, j.\ \forall\, i.\ A &
\end{array}$$

The limitation to only finite El A in the isomorphism

$$\exists\, i\,.\, (x : \mathrm{El}\ A) \ \to\ \exists\, j < i.\ B \cong (x : \mathrm{El}\ A) \ \to\ \exists\, j\,.\, B$$

is because, to go from right to left, we need to find a i which is an upper bound for all the js returned by the function $(x : \mathrm{El}\ A) \to \exists j.\ B$ across the whole domain El A. However given only $\sqcup$ we can only compute the upper bound of finitely many time values. We did not introduce a $\mathsf{limit} : (A \to \mathsf{Time}) \to \mathsf{Time}$ operator because A might contain Time itself, and that would have led to impredicativity issues in the model of Section 4. We plan to lift this restriction in further work.

3 Inductive Types

In this section we will show how to implement the recursive type equations we have used in terms of fixed points on the universe, then the induction principle for Nat, and lastly we construct an initial algebra for any functor that properly commutes with $\exists\ i$.

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash (x : A) \to B : \mathsf{Type}} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Sigma(x : A).\, B : \mathsf{Type}}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A + B : \mathsf{Type}} \qquad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash A \times B : \mathsf{Type}}$$

$$\frac{\Gamma \vdash \qquad X = \bot, \top, \mathsf{Bool}}{\Gamma \vdash X : \mathsf{Type}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathrm{U} : \mathsf{Type}} \qquad \frac{\Gamma \vdash u : \mathrm{U}}{\Gamma \vdash \mathrm{El}\ u : \mathsf{Type}}$$

Figure 1: Dependent Type Theory with a Universe

$$\frac{}{\Gamma \vdash \mathsf{Time} : \mathsf{Type}} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Time}} \qquad \frac{\Gamma \vdash t : \mathsf{Time}}{\Gamma \vdash \uparrow t : \mathsf{Time}}$$

$$\frac{\Gamma \vdash t_0 : \mathsf{Time} \qquad \Gamma \vdash t_1 : \mathsf{Time}}{\Gamma \vdash t_0 \sqcup t_1 : \mathsf{Time}} \qquad \frac{\Gamma \vdash t_0 : \mathsf{Time} \qquad \Gamma \vdash t_1 : \mathsf{Time}}{\Gamma \vdash t_0 \leqslant t_1 : \mathsf{Type}}$$

Figure 2: The Time type

$$\frac{\Gamma, i : \mathsf{Time} \vdash A[i] : \mathsf{Type} \qquad \Gamma \vdash f : \forall i.\ (\forall j < i.\ A[j]) \to A[i]}{\Gamma \vdash \mathsf{fix}\ f : \forall i. A[i]}$$

$$\frac{f\ i\ (\mathsf{guard}_\square\ u\ i) = u\ i}{u\ i = \mathsf{fix}\ f\ i}$$

where

$$\mathsf{guard}_\square : (\forall i.\ A[i]) \to \forall i.\ \forall j < i.\ A[j]$$
$$\mathsf{guard}_\square\ f = \lambda i\ j.\ f\ j$$

Figure 3: Guarded Fixpoint

$$\frac{\Gamma, i : \mathsf{Time} \vdash A : U}{\Gamma \vdash \forall i.\, A : U} \qquad \frac{\Gamma, i : \mathsf{Time} \vdash A : U}{\Gamma \vdash \exists i.\, A : U} \qquad \frac{\Gamma \vdash t_0 : \mathsf{Time} \qquad \Gamma \vdash t_1 : \mathsf{Time}}{\Gamma \vdash t_0 \leqslant t_1 : U}$$

$$\frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash B : U}{\Gamma \vdash (x : A) \to B : U} \qquad \frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash B : U}{\Gamma \vdash \Sigma(x : A).\, B : U}$$

$$\frac{\Gamma \vdash A : U \qquad \Gamma \vdash B : U}{\Gamma \vdash A + B : U} \qquad \frac{\Gamma \vdash A : U \qquad \Gamma \vdash B : U}{\Gamma \vdash A \times B : U}$$

$$\frac{\Gamma \vdash \qquad X = \bot, \top, \mathsf{Bool}}{\Gamma \vdash X : U}$$

Figure 4: Codes for the Universe U

$$\frac{\Gamma,\, i : \mathsf{Time} \vdash A : \mathsf{Type}}{\Gamma \vdash \exists i.\, A : \mathsf{Type}}$$

$$\frac{\Gamma,\, i : \mathsf{Time} \vdash A[i] : \mathsf{Type} \qquad \Gamma \vdash t : \mathsf{Time} \qquad \Gamma \vdash e : A[t]}{\Gamma \vdash (t, e) : \exists i.\, A[i]}$$

$$\frac{\begin{array}{c}\Gamma,\, x : \exists i.\, A \vdash P[x] : \mathrm{U} \\ \Gamma \vdash e_\exists : \exists i.\, A \qquad \Gamma,\, i : \mathsf{Time},\, a : A \vdash e_p : \mathrm{El}\ (P[(i, a)])\end{array}}{\Gamma \vdash \mathsf{case}\ e_\exists\ \mathsf{of}\ (i, a)\ \to\ e_p : \mathrm{El}\ (P[e_\exists])}$$

$\exists$-β

$$\frac{\begin{array}{c}\Gamma,\, x : \exists i.\, A[i] \vdash P[x] : \mathrm{U} \\ \Gamma \vdash t : \mathsf{Time} \qquad \Gamma \vdash e_a : A[t] \qquad \Gamma,\, i : \mathsf{Time},\, x : A[i] \vdash e_p[i, x] : \mathrm{El}\ (P[(i, x)])\end{array}}{\Gamma \vdash (\mathsf{case}\ (t, e_a)\ \mathsf{of}\ (i, x)\ \to\ e_p[i, x]) = e_p[t, e_a] : \mathrm{El}\ (P[(t, e_a)])}$$

$\exists$-η

$$\frac{\begin{array}{c}\Gamma,\, x : \exists i.\, A \vdash P[x] : \mathrm{U} \qquad \Gamma,\, x : \exists i.\, A,\, p : \mathrm{El}\ (P[x]) \vdash Q[x, p] : \mathrm{U} \\ \Gamma,\, i : \mathsf{Time},\, a : A \vdash e_p : \mathrm{El}\ (P[(i, a)]) \\ \Gamma \vdash e_\exists : \exists i.\, A \qquad \Gamma,\, x : \exists i.\, A,\, p : \mathrm{El}\ (P[x]) \vdash e_q[x, p] : \mathrm{El}\ (Q[x, p])\end{array}}{\begin{array}{c}\Gamma \vdash e_q[e_\exists, \mathsf{case}\ e_\exists\ \mathsf{of} (i, a) \to e_p] = \mathsf{case}\ e_\exists\ \mathsf{of}\ (i, a)\ \to e_q[(i, a), e_p] \\ : Q[e_\exists, \mathsf{case}\ e_\exists\ \mathsf{of}\ (i, a)\ \to\ e_p]\end{array}}$$

Figure 5: Existential Time Quantifier

$$\frac{\Gamma \vdash t : \mathsf{Time}}{\Gamma \vdash \mathsf{refl}\ t : t \leqslant t} \qquad \frac{\Gamma \vdash p_0 : t_0 \leqslant t_1 \qquad \Gamma \vdash p_1 : t_1 \leqslant t_2}{\Gamma \vdash \mathsf{trans}\ p_0\ p_1 : t_0 \leqslant t_1} \qquad \frac{\Gamma \vdash t : \mathsf{Time}}{\Gamma \vdash \mathsf{zero}\ t : 0 \leqslant t}$$

$$\frac{\Gamma \vdash t : \mathsf{Time}}{\Gamma \vdash \mathsf{step}\ t : t \leqslant \uparrow t}$$

$$\frac{\Gamma \vdash t_0 : \mathsf{Time} \qquad \Gamma \vdash t_1 : \mathsf{Time}}{\Gamma \vdash \sqcup_0\ t_0\ t_1 : t_0 \leqslant t_0 \sqcup t_1} \qquad \frac{\Gamma \vdash t_0 : \mathsf{Time} \qquad \Gamma \vdash t_1 : \mathsf{Time}}{\Gamma \vdash \sqcup_1\ t_0\ t_1 : t_1 \leqslant t_0 \sqcup t_1}$$

Figure 6: Time inequalities

$$\begin{array}{rl}
\forall i.\ \mathrm{El}\ A \cong \mathrm{El}\ A & i \notin \mathsf{fv}(A) \\
\forall i.\ \mathrm{El}\ (A[i]) \cong \forall i.\ \forall j < i.\ \mathrm{El}\ (A[j]) & (\mathsf{guard}_\Box, \mathsf{force}_\Box) \\
(\forall i.\ A) + (\forall i.\ B) \cong \forall i.\ (A + B) & \\
\forall i.\ \Sigma(x : \mathrm{El}\ A).\ B \cong \Sigma(x : \mathrm{El}\ A).\ \forall i.\ B & i \notin \mathsf{fv}(A) \\
\\
\exists i.\ \mathrm{El}\ A \cong \mathrm{El}\ A & i \notin \mathsf{fv}(A) \\
\exists i.\ A[i] \cong \exists i.\ \exists j < i.\ A[j] & (\mathsf{guard}_\Diamond, \mathsf{force}_\Diamond) \\
(\exists i.\ A) + (\exists i.\ B) \cong \exists i.\ (A + B) & \\
(\exists i.\ A[i]) \times (\exists i.\ B[i]) \cong \exists i.\ (\exists j < i.\ A[j] \times \exists j < i.\ B[j]) & \\
\Sigma(x : \mathrm{El}\ A).\ \exists i.\ B \cong \exists i.\ \Sigma(x : \mathrm{El}\ A).\ B & i \notin \mathsf{fv}(A) \\
\exists i.\ (x : \mathrm{El}\ A) \to \exists j < i.\ B[j] \cong (x : \mathrm{El}\ A) \to \exists i.\ B[i] & \text{finite El } A, i \notin \mathsf{fv}(A) \\
\exists i.\ \exists j.\ A \cong \exists j.\ \exists i.\ A &
\end{array}$$

Figure 7: Type Isomorphisms

3.1 Recursive Type Equations

For any function $F : \mathrm{U} \to \mathrm{U}$ we can build $\mathsf{Fix}_\diamond\ F : \mathsf{Time} \to \mathrm{U}$ such that $\mathsf{Fix}_\diamond\ F\ i = F\ (\exists\, j < i.\ \mathsf{Fix}_\diamond\ F\ j)$.

The first step is to recognize that $\exists\, j < i.$ can take an element of $\forall\, j < i.\ \mathrm{U}$ as input rather than U or $\mathsf{Time} \to \mathrm{U}$, defining the combinator $\hat{\diamond}^i$.

$$\begin{aligned}&\hat{\diamond}^i : (\forall\, j < i.\ \mathrm{U}) \to \mathrm{U}\\ &\hat{\diamond}^i\ X = \exists\, j < i.\ X\ j\end{aligned}$$

Using the time analogy for intuition, in a case where we have no time left, so there's no $j < i$, we already know that $\exists\, j < i.\ A$ is equivalent to $\bot$ without having to know what A is, only if we actually have time to spare we will look into it.

Given $\hat{\diamond}^i$ we can turn $F : \mathrm{U} \to \mathrm{U}$ into $\forall i.\ (\forall j < i.\ \mathrm{U}) \to \mathrm{U}$ by precomposition and define $\mathsf{Fix}_\diamond$ through fix, giving us the desired property.

$$\mathsf{Fix}_\diamond = \mathsf{fix}\ (\lambda\, i\ X.\ F\ (\hat{\diamond}^i\ X))$$

In the same way we define $\mathsf{Fix}_\Box$ by $\hat{\Box}^i$:

$$\begin{aligned}&\hat{\Box}^i : (\forall\, j < i.\ \mathrm{U}) \to \mathrm{U}\\ &\hat{\Box}^i\ X = \forall\, j < i.\ X\ j\end{aligned}$$

3.2 Induction on Nat

To get a concrete feel for how we can reduce induction to fix we show induction for the natural numbers.

First we redefine Nat, and show the definition of its constructors Zero and Suc:

$$\begin{aligned}&\mathsf{Nat} = \mathrm{El}\ (\exists\, i.\ \mathsf{Fix}_\diamond\ (\lambda\, X.\ \top + X)\ i)\\ &\mathsf{Zero} : \mathsf{Nat}\\ &\mathsf{Zero} = (0, \mathsf{inl}\ \mathsf{tt})\\ &\mathsf{Suc} : \mathsf{Nat} \to \mathsf{Nat}\\ &\mathsf{Suc}\ (i, n) = (\uparrow i, \mathsf{inr}\ (i, n))\end{aligned}$$

Then we can define an induction principle for the type families $P : \mathsf{Nat} \to \mathrm{U}$ which live in the universe U, since we are restricted by the elimination rule of $\exists\, i$.

$$\begin{aligned}&\mathsf{indNat} : (P : \mathsf{Nat} \to \mathrm{U})\\ &\qquad \to P\ \mathsf{Zero}\\ &\qquad \to (\forall\, n \to \mathrm{El}\ (P\ n) \to \mathrm{El}\ (P\ (\mathsf{Suc}\ n)))\\ &\qquad \to (n : \mathsf{Nat}) \to \mathrm{El}\ (P\ n)\\ &\mathsf{indNat}\ P\ z\ s\ (i, n) =\end{aligned}$$

$$
\begin{array}{l}
\mathsf{fix}\ (\lambda\ i\ rec\ n\ \to\ \mathsf{case}\ n\ \mathsf{of} \\
\qquad\qquad\qquad\qquad \mathsf{inl}\ \mathsf{tt}\ \ \ \to\ z \\
\qquad\qquad\qquad\qquad \mathsf{inr}\ (j,n)\ \to\ s\ (j,n)\ (rec\ j\ n) \\
\quad) \\
\quad i\ n
\end{array}
$$

Reading carefully we see that $s\ (j,n)\ (rec\ j\ n)$ has type $\mathrm{El}\ (P\ (\mathsf{Suc}\ (j,n)))$ which we can reduce to $\mathrm{El}\ (P\ (\uparrow\ j, \mathsf{inr}\ (j,n)))$ while the expected type is $\mathrm{El}\ (P\ (i, \mathsf{inr}\ (j,n)))$. We can however conclude that $(\uparrow j, \mathsf{inr}\ (j,n))$ and $(i, \mathsf{inr}\ (j,n))$ are equal since they both get sent to (j,n) by the

$$\exists\ i\,.\ \top + (\exists\ j < i\,.\ A) \cong \top + \exists\ i.\ A$$

isomorphism.

For an induction principle we also want the right computational behaviour when applied to the constructors for the datatype. We have that $\mathsf{indNat}\ P\ z\ s\ \mathsf{Zero} = z$ simply by unfolding fix and reducing the case distinction. However for

$$\mathsf{indNat}\ P\ z\ s\ (\mathsf{Suc}\ n)\ =\ s\ n\ (\mathsf{indNat}\ P\ z\ s\ n)$$

we also need to apply the η rule for $\exists$.

$$
\begin{array}{l}
\mathsf{indNat}\ P\ z\ s\ (\mathsf{Suc}\ n) \\
=\ \mathsf{indNat}\ P\ z\ s\ (\mathsf{case}\ n\ \mathsf{of}\ (i,n')\ \to\ (\uparrow i, \mathsf{inr}\ (i,n'))) \\
=\ \mathsf{case}\ n\ \mathsf{of}\ (i,n')\ \to\ \mathsf{indNat}\ P\ z\ s\ (\uparrow i, \mathsf{inr}\ (i,n')) \\
=\ \mathsf{case}\ n\ \mathsf{of}\ (i,n')\ \to\ s\ (i,n')\ (\mathsf{indNat}\ P\ z\ s\ (i,n')) \\
=\ s\ n\ (\mathsf{case}\ n\ \mathsf{of}\ (i,n')\ \to\ (\mathsf{indNat}\ P\ z\ s\ (i,n'))) \\
=\ s\ n\ (\mathsf{indNat}\ P\ z\ s\ n)
\end{array}
$$

3.3 Internal Indexed Functors

In order to prove the existence of initial algebras we define the category of indexed types, in the universe U, with respect to some context Γ. For each $\Gamma \vdash X : \mathit{Type}$ we define the category of X-indexed types with terms $\Gamma \vdash A : X\ \to\ \mathrm{U}$ as objects and $\Gamma \vdash f : (x : X)\ \to\ \mathrm{El}\ (A\ x)\ \to\ \mathrm{El}\ (B\ x)$ as morphisms, identities and composition are defined as expected. We also write $A \Rightarrow B$ for $(x : X)\ \to\ \mathrm{El}\ (A\ x)\ \to\ \mathrm{El}\ (B\ x)$ and leave the index x implicit when only lifted pointwise.

An X-indexed functor F is then a pair of terms

$$
\begin{array}{l}
\Gamma \vdash F_0 : (X\ \to\ \mathrm{U})\ \to\ (X\ \to\ \mathrm{U}) \\
\Gamma \vdash F_1 : (A\ B : X\ \to\ \mathrm{U})\ \to \\
\qquad\qquad (A \Rightarrow B)\ \to\ (F\ A \Rightarrow F\ B)
\end{array}
$$

such that F_1 preserves identities and composition.

As an example we can make $\exists\, j < i.$ into a $(\mathsf{Time} \times X)$-indexed functor by defining $\diamond$ like so:

$$(\diamond)_{\,0} : (\mathsf{Time} \times X \;\to\; \mathrm{U}) \;\to\; (\mathsf{Time} \times X \;\to\; \mathrm{U})$$
$$(\diamond)_{\,0}\; A\; (i, x) \;=\; \exists\, j < i.\; A\; (j, x)$$

the action on morphisms is also pointwise and the η rule for $\exists$ and $\times$ are enough to preserve identities and composition. For any X-indexed functor F we define the $\mathsf{Time} \times X$-indexed functor $F\; [\diamond\; -]$ that threads the Time component to $\diamond$.

$$F\; [\diamond\; A]_{\,0}\; (i, x) \;=\; F_0\; (\lambda\, y.\; \exists\, j < i.\; A\; y)\; x$$

3.4 Guarded Initial Algebras

For each X-indexed functor F we obtain the initial algebra of $F\; [\diamond\; -]$ by

$$\mu^{\diamond}\; F\; (i, x) \;=\; \mathsf{fix}\; (\lambda\; i\; A\; x.\; F\; (\lambda\; y.\; \exists\, j < i.\; A\; (j, x))\; i\; x$$

so that $\mu^{\diamond}\; F \;=\; F\; [\diamond\; \mu^{\diamond}\; F]$.

The algebra $F\; [\diamond\; \mu^{\diamond}\; F] \Rightarrow \mu^{\diamond}\; F$ is then simply the identity and the morphism from any other algebra $f : F\; [\diamond\; A] \Rightarrow A$ is also definable through fix, which also ensures its uniqueness.

$$\mathsf{fold}^{\diamond} : (A : \mathsf{Time} \times X \;\to\; \mathrm{U}) \;\to$$
$$\qquad (F\; [\diamond\; A] \Rightarrow A) \;\to\; \mu^{\diamond}\; F \Rightarrow A$$
$$\mathsf{fold}^{\diamond}\; A\; f\; (i, x) \;=\; \mathsf{fix}\; (\lambda\; i\; \mathsf{fold}^{\diamond}\; m.\; f\; (F\; (\mathsf{fold}^{\diamond}\; \star)\; m))\; i\; x$$

3.5 Initial Algebras

Initial algebras can then be obtained by $\lambda\, x \;\to\; \exists\, i.\; \mu^{\diamond}\; F\; (i, x)$ for those X-indexed functors F which weakly commute with $\exists\, i$ in the following sense.

Definition 3.1. Let F be an X-indexed functor, we say that F weakly commutes with $\exists\, i$ if the canonical map

$$(x : X) \;\to\; (\exists\, i.\; F\; [\diamond\; A]\; (i, x))$$
$$\qquad \to\; F\; (\lambda\, x' \;\to\; \exists\, i.\; A\; (i, x'))\; x$$

is an isomomorphism for every A.

Theorem 3.1. *Let F be an X-indexed functor that weakly commutes with $\exists\, i$, then $\mu\; F\; x \;=\; \exists\, i.\; \mu^{\diamond}\; F\; (i, x)$ is the initial algebra of F.*

Proof (Sketch). From F weakly commuting with $\exists\, i$ at type $\mu^{\diamond}\; F$ we obtain an indexed isomorphism $F\; (\mu\; F) \cong \mu\; F$ and so in particular an algebra $F\; (\mu\; F) \Rightarrow F$, the morphism from any other algebra is obtained from the one for $\mu^{\diamond}$ and inherits his uniqueness since there's a bijection between algebra morphisms like in [Møgelberg].

$$\begin{array}{l}\mathsf{fold} : (A : I \to \mathrm{U}) \to (F\ A \Rightarrow A) \to (\mu\ F \Rightarrow A) \\ \mathsf{fold}\ A\ f\ x\ (i, m) = \mathsf{fold}^{\diamond}\ (\lambda\ (i, x) \to A\ x) \\ \quad (\lambda\ j\ m \to f\ (F\ \mathsf{extract}\ m))\ i\ x\ m\end{array}$$

To determine whether a functor F weakly commutes with $\exists\ i$ we make use of the isomorphisms of Figure 7, in particular we can handle the finitary-branching strictly positive functors but not functors of the form

$$F\ X = \mathsf{Nat} \to X$$

or

$$F\ X = \mathsf{Stream}\ X$$

because of the limitations already discussed.

3.6 (Guarded) Final Coalgebras

The above result can be dualized to obtain final coalgebras of X-indexed functors F that commute with $\forall\ i$.

For each X-indexed functor F we obtain the final coalgebra of $F\ [\Box\ -]$ by

$$\nu^{\Box}\ F\ (i, x) = \mathsf{fix}\ (\lambda\ i\ A\ x.\ F\ (\lambda\ y.\ \forall\ j < i\ .\ A\ (j, x))\ i\ x$$

so that $\nu^{\Box}\ F = F\ [\Box\ \nu^{\Box}\ F]$.

Definition 3.2. Let F be an X-indexed functor, we say that F weakly commutes with $\forall\ i$ if the canonical map

$$\begin{array}{l}(x : X) \to F\ (\lambda\ x'.\ \forall\ i.\ A\ (i, x'))\ x \\ \qquad\quad \to (\forall\ i\ .\ F\ [\Box\ A]\ (i, x))\end{array}$$

is an isomomorphism for every A.

Theorem 3.2. *Let F be an X-indexed functor that weakly commutes with $\forall\ i$, then $\nu\ F\ x = \exists\ i.\ \nu^{\Box}\ F\ (i, x)$ is the final coalgebra of F.*

3.7 Mixed Recursion-Corecursion

Here we show with an example that the language can handle cases of mixed recursion-corecursion by nested uses of fix.

From [Blanchette et al.] we take the example of a function

$$\mathsf{cat} : \mathsf{Nat} \to \mathsf{Stream\ Nat}$$

such that cat 1 is the stream of Catalan numbers: $C_1, C_2, C_3, \ldots$ where $C_n = \frac{1}{1+n}\binom{2n}{n}$. The function itself is of little interest other than being a concise example of mixed recursion-corecursion.

We define $\mathsf{Stream}\ A$ as the final coalgebra of $A \times$, with $\mathsf{Stream}^{\,i}\ A$ as the guarded version.

$$\begin{aligned}&\mathsf{Stream}^{\,i}\ A\ =\ \mathrm{El}\ (\mathsf{Fix}_{\Box}\ (\lambda\ X\,.\ A \times X)\ i)\\&\mathsf{Stream}\ A\ =\ \forall\ i\,.\ \mathsf{Stream}^{\,i}\ A\end{aligned}$$

We also assume we have a function

$$\oplus : \forall\ i\,.\ \mathsf{Stream}^{\,i}\ \mathsf{Nat}\ \to\ \mathsf{Stream}^{\,i}\ \mathsf{Nat}\ \to\ \mathsf{Stream}^{\,i}\ \mathsf{Nat}$$

which sums the elements of two streams pointwise.

Given the above we nest two calls to fix, the outer one used for the corecursive calls, while the inner one for the recursive one. The order of the nesting makes so $\mathsf{cat}^{\Box}$ can accept an arbitrary natural number as argument, while $\mathsf{cat}^{\Diamond}$ only one strictly bounded by j. Conversely $\mathsf{cat}^{\Diamond}$ produces a stream bounded by i while $\mathsf{cat}^{\Box}$ only a stream at a smaller bound.

$$\begin{aligned}&\mathsf{cat}' : \forall\ i\ \to\ \mathsf{Nat}\ \to\ \mathsf{Stream}^{\,i}\ \mathsf{Nat}\\&\mathsf{cat}'\ =\ \mathsf{fix}\ (\lambda\ i\ \mathsf{cat}^{\Box}\,.\ \mathsf{uncurry}\ (\mathsf{fix}\ (\lambda\ j\ \mathsf{cat}^{\Diamond}\ n\,.\\&\quad \mathsf{case}\ n\ \mathsf{of}\\&\qquad \mathsf{inl}\ \_\ \ \to\ \mathsf{Suc}\ \mathsf{Zero}, \mathsf{cat}^{\Box}\ \circledast\ \mathsf{pure}\ (\mathsf{Suc}\ \mathsf{Zero})\\&\qquad \mathsf{inr}\ n'\ \to\ \mathsf{extract}\ (\mathsf{cat}^{\Diamond}\ \star_j\ n')\\&\qquad\qquad\quad \oplus\ (\mathsf{Zero}, \mathsf{cat}^{\Box}\ \circledast\ \mathsf{pure}\ (\mathsf{Suc}\ (j, \mathsf{inr}\ n')))\\&\quad)))\\&\mathsf{cat}\ n\ =\ \lambda\ i\,.\ \mathsf{cat}'\ i\ n\end{aligned}$$

4 A Parametric Model

The type isomorphisms involving $\forall\ i$ of Figure 7 express the intuition that values are constructed in a Time invariant way, dually the isomorphisms involving $\exists\ i$ express the intuition that the only elimination principle for $\exists\ i$ is Time invariant.

Since [Reynolds], Relational Parametricity has been shown to be a good tool to capture invariance properties of programs, initially applied to invariance of polymorphic types under change of data representation, but also invariance under change of units of measure [Kennedy] or manipulations of vectors under translations [Atkey et al., b].

The basic principle is that types get modeled as a pair of a set of values and a relation over that set. The relation describes a property that has to be preserved when manipulating such values.

To model our language then we make use of the relationally parametric model of dependent type theory from [Atkey et al., a], defining Time as the type of natural numbers related by $\leqslant$ and the universe U as small sets related by proof-irrelevant relations (their "Small, Discrete, Proof Irrelevant Universe").

4.1 Reflexive Graphs as a Category with Families

The model is formulated as a Category with Families [Dybjer], of which we do not repeat the full definition but the main components are

- a category Cxt of contexts
- a collection $\mathsf{Ty}(\Gamma)$ of semantic types for each $\Gamma \in \mathrm{Obj}(\mathsf{Cxt})$
- a collection $\mathsf{Tm}(\Gamma, A)$ of semantic terms, for each $A \in \mathsf{Ty}(\Gamma)$.

The category Cxt in our case is the functor category $\mathsf{Set}^{\mathsf{RG}}$, where RG is the small category with two objects, two parallel arrows, and a common section. Since Cxt is a functor category into Set it inherits a standard Category with Families structure [Hofmann] including definitions for the standard connectives, most of them lifted pointwise from Set, here we will mention only enough to explain our own connectives.

An object Γ of Cxt is best thought of as a triple $(\Gamma_O, \Gamma_R, \Gamma_{\mathrm{refl}})$ where Γ_O is a set of objects, Γ_R is a binary relation over Γ_O and Γ_{refl} is a function witnessing the reflexivity of Γ_R.

$$\begin{aligned}
&\Gamma_O &&: Set\\
&\Gamma_R &&: \Gamma_O \times \Gamma_O \to Set\\
&\Gamma_{\mathrm{refl}} &&: \forall \gamma \in \Gamma_O.\ \Gamma_R(\gamma, \gamma)
\end{aligned}$$

We will refer to an element of Γ_O as an environment. Morphisms $f : \Gamma \to \Delta$ are then a pair of functions f_o and f_r which commute with reflexivity:

$$\begin{aligned}
&f_o : \Gamma_O \to \Delta_O\\
&f_r : \forall \gamma_0, \gamma_1 \in \Gamma.\ \Gamma_R(\gamma_0, \gamma_1) \to \Delta_R(f_o(\gamma_0), f_o(\gamma_1))\\
&\text{such that}\\
&\forall \gamma \in \Gamma_O.\ f_r(\Gamma_{\mathrm{refl}}(\gamma)) = \Delta_{\mathrm{refl}}(f_o(\gamma))
\end{aligned}$$

These morphisms should be thought of as substitutions, since they map environments of the context Γ into environments of Δ. We use the notations $A\{f\}$ and $M\{f\}$ to apply the substitution f to the type A and term M.

The collection $\mathsf{Ty}(\Gamma)$ of types then consists of families of reflexive graphs: a semantic type $A \in \mathsf{Ty}(\Gamma)$ is also a triple $(A_O, A_R, A_{\mathrm{refl}})$ but each component is indexed by the corresponding one from Γ, allowing types to depend on values from the environment.

$$\begin{aligned}
&A_O : \Gamma_O \to Set\\
&A_R : \forall \gamma_0, \gamma_1 \in \Gamma,\ \Gamma_R(\gamma_0, \gamma_1) \to A_O(\gamma_0) \times A_O(\gamma_1) \to Set\\
&A_{\mathrm{refl}} : \forall \gamma \in \Gamma,\ \forall a \in A_O(\gamma).\ A_R(\Gamma_{\mathrm{refl}}(\gamma), a, a)
\end{aligned}$$

A semantic term $M \in \mathsf{Tm}(\Gamma, A)$ correponds in principle to a map $\Gamma \to \Gamma.A$ such that $\mathsf{fst} \circ M = id_\Gamma$. It is however defined explicitly as a pair of the following

components:

$$
\begin{array}{l}
M_o : \forall \gamma \in \Gamma_O,\ A_O(\gamma) \\
M_r : \forall \gamma_0, \gamma_1 \in \Gamma_O,\ \forall \gamma_r \in \Gamma_R(\gamma_0, \gamma_1),\ A_R(\gamma_r, M_o(\gamma_0), M_o(\gamma_1)) \\
\text{such that} \\
\forall \gamma \in \Gamma_O,\ M_r(\Gamma_{\text{refl}}(\gamma)) = A_{\text{refl}}(\Gamma_{\text{refl}}(\gamma), M_o(\gamma))
\end{array}
$$

The empty context $\epsilon \in \mathrm{Obj}(\mathsf{Cxt})$ is defined as the singleton reflexive graph $\epsilon = (\{*\}, (\lambda \_ \_ . \{*\}), (\lambda \_ \_ . *))$. As a consequence an element of $Ty(\epsilon)$ corresponds to an object of Cxt. We can also extend a context Γ by a type $A \in \mathsf{Ty}(\Gamma)$ to obtain another context $\Gamma.A$ by pairing up each component.

$$
\begin{array}{l}
(\Gamma.A)_O = \{(\gamma, a) \mid \gamma \in \Gamma, a \in A_O(\gamma)\} \\
(\Gamma.A)_R((\gamma_0, a_0), (\gamma_1, a_1)) \\
\qquad = \{(\gamma_r, a_r) \mid \gamma_r \in \Gamma_R(\gamma_0, \gamma_1), a_r \in A_R(\gamma_r, a_0, a_1)\}
\end{array}
$$

We then have a map $\mathsf{fst} : \Gamma.A \to \Gamma$ which projects out the first component and has the role of a weakening substitution.

4.2 A Small, Discrete, Proof Irrelevant Universe

In order to recover standard parametricity results like the free theorems from [Wadler, 1989], Atkey et. al define a universe $\mathrm{U} \in \mathsf{Ty}(\Gamma)$ to connect the relations of the reflexive graphs to the equality of their set of objects. In particular for each $A \in \mathsf{Tm}(\Gamma, \mathrm{U})$ we get a type $\mathrm{El}\ A \in \mathsf{Ty}(\Gamma)$ such that $(\mathrm{El}\ A)_R(\Gamma_{\text{refl}}(\gamma))$ is the equality relation of $(\mathrm{El}\ A)_O(\gamma)$, up to isomorphism.

Fixing a set-theoretic universe $\mathcal{U}$, we can define the following properties of reflexive graphs:

Definition 4.1. A reflexive graph A is:

- small if $A_O \in \mathcal{U}$ and for all $a_0, a_1 \in A_O$, $A_R(a_0, a_1) \in \mathcal{U}$
- discrete if A is isomorphic to a reflexive graph generated by a set, i.e. $A \cong (X, =_X, \mathrm{refl}_{=_X})$ for some set X.
- proof-irrelevant if, for all $a_0, a_1 \in A_O$, the map $A_R(a_0, a_1) \to \{*\}$ is injective.

We are now ready to define U and El:

$$
\begin{array}{ll}
\mathrm{U} \in \mathsf{Ty}(\Gamma) & \\
\mathrm{U}_O(\gamma) & = \text{the set of small discrete reflexive graphs} \\
\mathrm{U}_R(\gamma_r)(A,B) & = \{R : A_O \to B_O \to \mathit{Set} \mid \\
& \qquad \forall a \in A_O, b \in B_O, R(a,b) \in \mathcal{U} \\
& \qquad \text{, the map } R(a,b) \to \{*\} \text{ is injective } \} \\
\mathrm{U}_{\mathrm{refl}}(\gamma)(A) & = A_R \\
 & \\
\mathrm{El} \in \mathsf{Ty}(\Gamma.\mathrm{U}) & \\
\mathrm{El}_O(\gamma, A) & = A_O \\
\mathrm{El}_R(\gamma_r, R) & = R \\
\mathrm{El}_{\mathrm{refl}}(\gamma)(A) & = A_{\mathrm{refl}}
\end{array}
$$

Assuming that the set-theoretic universe $\mathcal{U}$ is closed under the corresponding operations, the universe U is shown to contain product and sum types, natural numbers, to be closed under Σ types and to contain $\Pi(x : A).\mathrm{El}\ (B\ x)$ for any small type A.

4.2.1 Invariance through Discreteness

One main feature of U is exemplified by considering a term M from $\mathsf{Tm}(\Gamma.A, (\mathrm{El}\ B)\{\mathsf{fst}\})$, i.e., where the type of the result is in the universe and does not depend on A.

Lemma 4.1. *Let* $A \in \mathsf{Ty}(\Gamma)$*,* $B \in \mathsf{Tm}(\Gamma, \mathrm{U})$*, and* $M \in \mathsf{Tm}(\Gamma.A, (\mathrm{El}\ B)\{\mathsf{fst}\})$ *then*

$$
\begin{array}{l}
\forall \gamma \in \Gamma_O,\ \forall a_0, a_1 \in A_O(\gamma),\ \forall a_r \in \mathcal{A}_R(\Gamma_{\mathrm{refl}}(\gamma), a_0, a_1), \\
M_o(\gamma, a_0) = M_o(\gamma, a_1)
\end{array}
$$

Proof. Unfolding the application of fst and unpacking the environment for $\Gamma.A$ we get the following for the components of M. The condition of commuting with reflexivity is between two elements of a proof-irrelevant relation, so can be omitted.

$$
\begin{array}{l}
M_o : \forall \gamma \in \Gamma_O,\ \forall a \in A_O(\gamma),\ (\mathrm{El}\ B)_O(\gamma) \\
M_r : \forall \gamma_0, \gamma_1 \in \Gamma_O,\ \forall \gamma_r \in \Gamma_R(\gamma_0, \gamma_1), \\
\qquad \forall a_0 \in A_O(\gamma_0), a_1 \in A_O(\gamma_1),\ \forall a_r \in \mathcal{A}_R(\gamma_r, a_0, a_1), \\
\qquad (\mathrm{El}\ B)_R(\gamma_r, M_o(\gamma_0, a_0), M_o(\gamma_1, a_1))
\end{array}
$$

We see that the result of M_r does not mention a_r because El B does not depend on A, moreover if we specialize γ_r to $\Gamma_{\mathrm{refl}}(\gamma)$ we get

$$(\mathrm{El}\ B)_R(\Gamma_{\mathrm{refl}}(\gamma), M_o(\gamma, a_0), M_o(\gamma, a_1))$$

which we know to be isomorphic to $M_o(\gamma, a_0) = M_o(\gamma, a_1)$ so we have our result. □

In other words, for a fixed γ, we have that M_o considers any two related a_i as being equal, since it returns the same result.

4.3 Interpretation of the Language

We are left with having to interpret our own primitives.

4.3.1 Time

The type Time is intepreted as the reflexive graph of natural numbers with $n \leq m$ as the relation. Assuming that $\mathbb{N} \in \mathcal{U}$ we have that Time is small, but not discrete.

$$\begin{array}{ll} \mathsf{Time} \in \mathsf{Ty}(\epsilon) & \\ \mathsf{Time}_O & = \mathbb{N} \\ \mathsf{Time}_R(n, m) & = \{* \mid n \leq m\} \\ \mathsf{Time}_{\mathrm{refl}}(n) & = * \end{array}$$

The terms for 0 and $\uparrow$ are implemented by 0 and $+1$ on the underlying naturals, and $\sqcup$ by taking the maximum.

The use of $\leq$ as relation instead of $=$ is how we encode the invariance with respect to time values that we want in the model. In fact from Lemma 4.1 it follows that that a term $M \in \mathsf{Tm}(\Gamma.\mathsf{Time}, (\mathrm{El}\ B)\{\mathsf{fst}\})$ is going to produce the same result no matter what natural number it gets from the environment, since they are all related, which justifies the isomorphism $\forall i.\mathrm{El}\ B \cong \mathrm{El}\ B$ of the language.

The relation between times $i \leqslant j$ is intepreted by
$\leq \in \mathsf{Ty}(\mathsf{Time}.\mathsf{Time}\{\mathsf{fst}\})$, which also fits in U since it has no interesting inhabitants.

$$\begin{array}{ll} \leq_O (n, m) & = \{* \mid n \leq m\} \\ \leq_R (\_, \_) & = \{*\} \\ \leq_{\mathrm{refl}} (\_) & = * \end{array}$$

The fixpoint operator fix and its uniqueness are implemented through well-founded induction on the natural numbers.

4.3.2 Representationally Independent Existential

We will ultimately define the existential quantification over Time in the same style as a parametric colimit in the sense of [Dunphy and Reddy]. However we will show a connection with the standard Σ type by first defining a general operation to convert any small reflexive graph into a discrete and proof-irrelevant one.

Given a small $A \in \mathsf{Ty}(\Gamma)$ we define $\mathsf{Tr}A \in \mathsf{Tm}(\Gamma, \mathrm{U})$ which we call the discrete truncation of A.We first give some preliminary definitions on reflexive graphs, and then lift those to the case of families to define Tr.

For a reflexive graph $A \in \mathrm{Obj}(\mathsf{Cxt})$ we define $A_O/^{=}A_R$ to be the set obtained by quotienting A_O with $A_R^{=}$, which is how we denote the symmetric transitive closure of A_R.

$$A_O/^{=}A_R = A_O/(\lambda a_0\ a_1.\ \exists \tilde{a}.\ \tilde{a} \in A_R^{=}(a_0, a_1))$$

Moreover we define $\mathsf{Lift}_{(A,B)}(R)$ to lift a relation $R : A_O \to B_O \to Set$ to a relation $A_O/^{=}A_R \to B_O/^{=}B_R \to Set$, so that we have a function $\mathsf{lift}_R : \forall a, b.\, R(a,b) \to \mathsf{Lift}_{(A,B)}(R)([a],[b])$.

$$
\begin{aligned}
&\mathsf{Lift} : \forall A, B \in \mathrm{Obj}(\mathsf{Cxt}), \forall (R : A_O \to B_O \to Set),\\
&\qquad A_O/^{=}A_R \to B_O/^{=}B_R \to Set\\
&\mathsf{Lift}_{(A,B)}(R)([a],[b])\\
&= \{(a', \tilde{a}, b', \tilde{b}, r) \mid a' \in A_O, \tilde{a} \in A_R^{=}(a,a'),\\
&\qquad\qquad\qquad\qquad b' \in B_O, \tilde{b} \in B_R^{=}(b,b'),\\
&\qquad\qquad\qquad\qquad r \in R(a',b')\}/\top
\end{aligned}
$$

The definition of $\mathsf{Lift}_{(A,B)}(R)$ is given on the representatives a and b of the equivalence classes $[a]$ and $[b]$, this is justified because we produce logically equivalent relations for related elements. We note that $\mathsf{Lift}_{A,B}(R)$ is proof irrelevant by construction, since we define it as a quotient with the total relation which we name $\top$, and that $\mathsf{Lift}_{(A,A)}(A_R)$ is logically equivalent to the equality relation on $A_O/^{=}A_R$.

Finally we define $\mathsf{Tr}A$ for a given $A \in Ty(\Gamma)$:

$$
\begin{aligned}
&(\mathsf{Tr}A) : \mathsf{Tm}(\Gamma, \mathrm{U})\\
&(\mathsf{Tr}A)_o(\gamma) = (A_O(\gamma)/^{=}A_R(\Gamma_{\mathrm{refl}}(\gamma)), \mathsf{Lift}(A_R(\Gamma_{\mathrm{refl}}(\gamma))))\\
&(\mathsf{Tr}A)_r(\gamma_r) = \mathsf{Lift}(A_R(\gamma_r))
\end{aligned}
$$

we have to show that $(\mathsf{Tr}A)_o$ and $(\mathsf{Tr}A)_r$ commute with reflexivity,

$$\forall \gamma, \mathrm{U}_{\mathrm{refl}}((\mathsf{Tr}A)_o(\gamma)) = (\mathsf{Tr}A)_r(\Gamma_{\mathrm{refl}}(\gamma))$$

but $\mathrm{U}_{\mathrm{refl}}$ projects out the relation given as the second component of the tuple, so both sides reduce to $\mathsf{Lift}(A_r(\Gamma_{\mathrm{refl}}(\gamma)))$ and we are done.

We remark that, for any $A \in \mathsf{Tm}(\Gamma, \mathrm{U})$, the types $\mathrm{El}\ A$ and $\mathrm{El}\ (\mathsf{Tr}A)$ are equivalent. In fact $A_r(\Gamma_{\mathrm{refl}}(\gamma))$ is already equivalent to equality for $A_O(\gamma)$ so the quotient $A_O(\gamma)/^{=}A_r(\Gamma_{\mathrm{refl}}(\gamma))$ is equivalent to $A_O(\gamma)$, and for the same reason $\mathsf{Lift}(A_r(\gamma_r))$ is equivalent to $A_r(\gamma_r)$.

The next step is to define an introduction and an eliminator for $\mathsf{Tr}A$; the introduction sends an element of A to its equivalence class:

$$
\begin{aligned}
&\mathsf{tr} : \mathsf{Tm}(\Gamma.A, \mathrm{El}\ (\mathsf{Tr}A)\{\mathsf{fst}\})\\
&\mathsf{tr}_o(\_, a) \ = [a]\\
&\mathsf{tr}_r(\_, a_r) = \mathsf{lift}(a_r)
\end{aligned}
$$

We then have dependent elimination of $\mathsf{Tr}A$ into other types that live in the universe: $B \in \mathsf{Tm}(\Gamma.\mathrm{El}\ (\mathsf{Tr}A), \mathrm{U})$. Given a $t \in \mathsf{Tm}(\Gamma.A, \mathrm{El}\ B\{\mathsf{fst}, \mathsf{tr}\})$ we define elim:

$$
\begin{aligned}
&\mathsf{elim} : \mathsf{Tm}(\Gamma.\mathrm{El}\ (\mathsf{Tr}A), \mathrm{El}\ B)\\
&\mathsf{elim}_o(\gamma, [a]) \qquad\qquad\quad = t_o(\gamma, a)\\
&\mathsf{elim}_r(\gamma_r, [(a', \tilde{a}, b', \tilde{b}, r)]) = t_r(\gamma_r, r)
\end{aligned}
$$

Since elim_o and elim_r are defined by the representatives we need to show they are invariant under the quotienting relation, also $t_r(\gamma_r, r)$ is not immediately a member of the expected relation, both of these problems are solved by the discreteness of B.

First we show that t_r guarantees that t_o respects $A_R(\Gamma_{\text{refl}}(\gamma))$: we unfold the type of t_r,

$$\begin{aligned} t_r : \forall \gamma_0, \gamma_1 \in \Gamma_O,\ \gamma_r \in \Gamma_R(\gamma_0, \gamma_1), \\ \forall a_0 \in A_O(\gamma_0), a_1 \in A_O(\gamma_1), a_r \in A_R(\gamma_r, a_0, a_1), \\ (\text{El } B)_R((\gamma_r, \mathsf{tr}_r(a_r)), t_o(\gamma_0, a_0), t_o(\gamma_1, a_1)) \end{aligned}$$

and note that for any $\gamma \in \Gamma_O$ we can take $\gamma_r = \Gamma_{\text{refl}}(\gamma)$ and obtain

$$\begin{aligned} &\forall a_0, a_1 \in A_O(\gamma),\ a_r \in A_R(\Gamma_{\text{refl}}(\gamma), a_0, a_1), \\ &(\text{El } B)_R((\Gamma_{\text{refl}}(\gamma), \mathsf{tr}_r(a_r)), t_o(\gamma, a_0), t_o(\gamma, a_1)). \end{aligned}$$

To use the discreteness of (El B) we need the environment to be obtained by reflexivity of $(\Gamma.\mathsf{Tr}A)$, so we need $\mathsf{tr}_r(a_r) = (\mathsf{Tr}A)_{\text{refl}}(a_0)$, and that follows from the proof-irrelevance of $(\mathsf{Tr}A)_R$ and the fact that $\mathsf{tr}_o(a_0)$ equals $\mathsf{tr}_r(a_1)$. Hence from discreteness $(\text{El}B)_R(\Gamma_{\text{refl}}(\gamma), \mathsf{tr}_r(a_r))$ is equivalent to equality on $(\text{El } B)_O(\gamma, \mathsf{tr}_o(a_0))$ and we can conclude:

$$\begin{aligned} &\forall a_0, a_1 \in A_O(\gamma), \\ &A_R(\Gamma_{\text{refl}}(\gamma))(a_0, a_1) \to t_o(\gamma, a0) = t_o(\gamma, a_1) \end{aligned} \tag{1}$$

To justify the definition of elim_o we have to show that for two $a_0, a_1 \in A_O(\gamma)$ related by $a_r \in A_R(\Gamma_{\text{refl}}(\gamma), a_0, a_1)$, we have $t_o(\gamma, a_0) = t_o(\gamma, a_1)$; but this follows directly from the observation about t_r.

In the case of elim_r the proof-irrelevance of (El $B)_R$ already implies that we will produce the same result for different representatives, however it is less obvious that $t_r(\gamma_r, r)$ belongs in

$$(\text{El } B)_R(\gamma_r, [(a', \tilde{a}, b', \tilde{b}, r)]), t_o(\gamma_0, a), t_o(\gamma_1, b)).$$

We know that

$$t_r(\gamma_r, r) \in (\text{El } B)_R(\gamma_r, \mathsf{lift}(r)), t_o(\gamma_0, a'), t_o(\gamma_1, b'))$$

and from 1 we know that

$$t_o(\gamma_0, a) = t_o(\gamma_0, a')$$

and

$$t_o(\gamma_1, b) = t_o(\gamma_1, b')$$

so we also have

$$t_r(\gamma_r, r) \in (\text{El } B)_R(\gamma_r, \mathsf{lift}(r)), t_o(\gamma_0, a), t_o(\gamma_1, b))$$

and since $\mathsf{lift}(r) = [(a', \tilde{a}, b', \tilde{b}, r)]$ by proof-irrelevance, we obtain the result we wanted.

To define the existential we can then truncate the corresponding Σ type,

$$\exists i.A = \mathsf{Tr}(\Sigma i : \mathsf{Time}.A)$$

so that (i, a) is interpreted as the introduction for Σ followed by tr, while the case expression is interpreted as elim combined with the projections of Σ. More generally we could consider $\exists(x : A).B = \mathsf{Tr}(\Sigma x : A.B)$. If both A and B belong in U then $\exists(x : A).B$ is equivalent to $\Sigma(x : A).B$, which reproduces the standard result about recovering strong sums from weak ones by parametricity.

It is easy then to justify the isomorphism $\exists i.A \cong A$ for an A that doesn't mention i: the equality on $\mathsf{Tr}(\Sigma i : Time.A)$ ignores the Time fields because any two time values are related in some direction.

5 Related Works

The application of Nakano's guard modality[**?**] to coinductive types started with [Atkey and McBride, 2013] by the introduction of clock variables to an otherwise simply typed language, [Møgelberg] extended this result to a dependently typed setting where guarded recursive types are constructed via fixed points on universes. Their models are based on the topos of trees: a context with a free clock variable is interpreted as a functor $\mathsf{Set}^{\omega^{op}}$ where ω is the preorder of natural numbers as a category. In such a model every value available at present time can be transported to a later time by forgetting some of the contained information, e.g. guarded streams of natural numbers correspond to the functor

$$\mathsf{Stream}(n) = \mathbb{N}^{n+1}$$

where the action on morphisms, so-called restriction map, sends $\mathsf{Stream}(n+1)$ to $\mathsf{Stream}(n)$ by discarding the last element. Our $\diamond$ modality however would not fit in such a model because there is no map $(\diamond A)(1) \to (\diamond A)(0)$ in general: $(\diamond A)(0)$ is an empty set, since there are no future times, while $(\diamond A)(1)$ is only empty when $A(0)$ is. We are then forced to give up these maps, but having only $A : \mathsf{Set}^{|\omega|}$ i.e., a collection of sets indexed by natural numbers, would not be enough, in the parametric model the associated relation is used to impose the invariance conditions needed. However we lose the full applicative functor structure of $\Box$, i.e. we lack

$$\mathsf{pure} : \forall\, i\,.\, (A\ i\ \to\ \forall\, j < i.\ A\ j)$$

for arbitrary types A and B. This is also the case for the system in [Krishnaswami, 2013] where the Nakano modality is used to control the resource usage of funtional reactive programs. The lack of pure does not seem to cause expressivity problems with regard to the examples in [Atkey and McBride, 2013], and

pure can be implemented explicitly for those types that would support restriction maps.

The notation we use for the $\Box$ and $\Diamond$ modalities agrees with their use in provability logic and its Kripke models. Unfortunately we conflict with other works on guarded recursive types where $\Box$ is used as a nameless $\forall\, i$ [Krishnaswami, 2013, Clouston et al.].

In HOL the system of tactics presented in [Blanchette et al.] allows corecursive calls to appear under "well-behaved" corecursive functions, which consume as much of their coinductive inputs as they produce, i.e. that in our system would preserve the time values. They do not consider more complex relations between inputs and outputs and the well-behavedness of a function is not part of its type, so the user interface is simpler, even if less expressive.

5.1 Comparison to Sized Types

When extended with copatterns [Abel and Pientka], Sized Types also justify the totality of (co)recursive definitions by well-founded induction on what there is called *Size*. The calculus presented there is defined as an extension of System F-omega so equality of terms does not affect typing. However, Abel and Pientka specify a strong normalizing reduction semantics while we have only specified equalities. The calculus allows direct recursion, with which we can define a general fixed point combinator

$$\begin{aligned}
&\mathsf{fix}_\Box : \forall\, (A : \mathit{Size} \to *).\, (\forall\, i.\, (\forall\, j < i.\, A\ j) \to A\ i) \\
&\qquad \to \forall\, i.\, \forall\, j < i.\, A\ j \\
&\mathsf{fix}_\Box\ A\ f\ i\ j = f\ j\ (\mathsf{fix}_\Box\ A\ f\ j) \\
&\mathsf{fix} : \forall\, (A : \mathit{Size} \to *).\, (\forall\, i.\, (\forall\, j < i.\, A\ j) \to A\ i) \\
&\qquad \to \forall\, i.\, A\ i \\
&\mathsf{fix}\ A\ f\ i = f\ i\ (\mathsf{fix}_\Box\ A\ f\ i)
\end{aligned}$$

We have that $\mathsf{fix}\ A\ f\ i$ reduces to $f\ j\ (\mathsf{fix}_\Box\ A\ f\ j)$ which does not reduce further unless f does. Because of this, reduction alone does not validate the equality $\mathsf{fix}\ f\ i = f\ (\mathsf{fix}\ f)$ of our language, which instead would lead to the loss of strong normalization. This fixed point operator cannot be used to define recursive types since the calculus does not allow programs to compute types. Instead there are explicit μ and ν type formers for inductive and coinductive types. The calculus can handle arbitrary branching inductive types thanks to a model where *Size* is interpreted as the set of ordinal numbers. Sized Types have also been experimentally added to Agda and have been useful to allow more general patterns of recursion [Abel and Chapman], however the current definitional equality of Agda does not validate the isomorphisms from our language, so the problem of values that should be equal but differ only in *Size* values is still present.

Another important distinction is that Abel and Pientka aim for a judgemental equality that includes the the equations the user writes, because of this they include extra restrictions on when it is allowed to have a lambda abstraction over sizes, e.g., defining a second fixed-point combinator like the following

$$
\begin{aligned}
&\mathit{fix}' : \forall\,(A : \mathit{Size} \;\to\; *).\,(\forall\, i.\,(\forall\, j < i.\, A\; j) \;\to\; A\; i)\\
&\quad \to \forall\, i.\, A\; i\\
&\mathit{fix}'\; A\; f\; i \;=\; f\; i\; (\lambda\, j.\, \mathsf{fix}\; A\; f\; j)
\end{aligned}
$$

would not be accepted as a definition, the typing rule for the abstraction over $j < i$ checks whether it is already known that i is non-zero, and here that's not the case. This restriction ensures that evaluation, even when going under lambdas, never assumes more about the sizes in the context than what was already known, so the well-foundedness of the order on sizes can be used to show termination.

Our work instead does not impose additional restrictions on lambda abstractions involving sizes, so that the user can trust her intuition of well-founded induction to know whether an expression will be accepted. The rewriting semantics will need to cope with this by using a different strategy to block evaluation.

6 Conclusions and Further Work

We have presented a model for a dependently typed calculus which ensures the totality of both recursive and corecursive definitions in a modular way through typing. Recursive types are also reduced to well-founded recursion on Time and we have specified as isomorphisms the invariance properties needed to obtain the expected equational theory of inductive and coinductive types. We have left open the issue of handling infinitely-branching inductive types, because as explained in Section 2 we would need a limit operator for Time. We plan to investigate the meta-theoretic consequences of extending the theory with such a limit operator and modeling Time with a set of ordinals. Such an extension would allow us to embed a universe for dependent type theory as an inductive datatype, since we can encode induction-recursion as a fixed point of families of types, so the resulting theory would have a large proof-theoretic strength. We also want to formulate a strongly normalizing reduction semantics for our language, extending the result for Sized Types to our theory, in addition to formulating a decidable subset of the equational theory presented.

References

A. Abel and J. Chapman. Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types. *ArXiv e-prints*.

A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13*, pages 185–196. ACM Press.

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

R. Atkey, N. Ghani, and P. Johann. A relationally parametric model of dependent type theory. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 503–516, a.

R. Atkey, P. Johann, and A. Kennedy. Abstraction and invariance for algebraically indexed types. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, pages 87–100, b.

J. C. Blanchette, A. Popescu, and D. Traytel. Foundational extensible corecursion. *CoRR*, abs/1501.05425.

R. Clouston, A. Bizjak, H. B. Grathwohl, and L. Birkedal. Programming and reasoning with guarded recursion for coinductive types. *CoRR*, abs/1501.02925.

B. P. Dunphy and U. S. Reddy. Parametric limits. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 242–251.

P. Dybjer. Internal type theory. In *Types for Proofs and Programs, International Workshop TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134.

M. Hofmann. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory*, CPHC/BCS Distinguished Dissertations, pages 13–54. Springer London.

A. Kennedy. Types for units-of-measure: Theory and practice. In *Central European Functional Programming School - Third Summer School, CEFP 2009*, volume 6299 of *Lecture Notes in Computer Science*, pages 268–305. Springer.

N. R. Krishnaswami. Higher-order reactive programming without spacetime leaks. In *International Conference on Functional Programming (ICFP)*, 2013.

R. E. Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14*, page 71.

J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523.

P. Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.