Guarded Recursion in Agda via Sized Types

Niccolò Veltri
Department of Computer Science, IT University of Copenhagen, Denmark
nive@itu.dk

Niels van der Weide
Institute for Computation and Information Sciences, Radboud University
Nijmegen, The Netherlands
nweide@cs.ru.nl

Abstract

In type theory, programming and reasoning with possibly non-terminating programs and potentially infinite objects is achieved using coinductive types. Recursively defined programs of these types need to be productive to guarantee the consistency of the type system. Proof assistants such as Agda and Coq traditionally employ strict syntactic productivity checks, which often make programming with coinductive types convoluted. One way to overcome this issue is by encoding productivity at the level of types so that the type system forbids the implementation of non-productive corecursive programs. In this paper we compare two different approaches to type-based productivity: guarded recursion and sized types. More specifically, we show how to simulate guarded recursion in Agda using sized types. We formalize the syntax of a simple type theory for guarded recursion, which is a variant of Atkey and McBride's calculus for productive coprogramming. Then we give a denotational semantics using presheaves over the preorder of sizes. Sized types are fundamentally used to interpret the characteristic features of guarded recursion, notably the fixpoint combinator.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Categorical semantics

Keywords and phrases guarded recursion, type theory, semantics, coinduction, sized types

Digital Object Identifier 10.4230/LIPIcs.FSCD.2019.32

Funding *Niccolò Veltri*: Veltri was supported by a research grant (13156) from VILLUM FONDEN.

Acknowledgements We are thankful to Andreas Abel, Guillaume Allais, Herman Geuvers, Rasmus Ejlers Møgelberg and Andrea Vezzosi for discussions and valuable hints. We thank the anonymous referees for their useful comments.

1 Introduction

Dependent type theory is an expressive functional programming language that underlies the deductive system of proof assistants such as Agda [22] and Coq [10]. It is a total language, meaning that every program definable inside type theory is necessarily terminating. This is an important requirement that ensures the consistency of the type system.

Possibly non-terminating computations and infinite structures, such as non-wellfounded trees, can be represented in type theory by extending the type system with coinductive types. Recursively defined elements of these types need to be productive in the sense that every finite part of the output can be computed in a finite number of steps [15]. In Agda's encoding of coinductive types using "musical notation" [16], productivity is enforced via a strict obligation: in the definition of a corecursive function, recursive calls must appear directly under the application of a constructor. A similar syntactic check is used in Coq. This restriction typically makes programming with coinductive types cumbersome, which spawned the search for alternative techniques to ensure the well-definedness of corecursive definitions.

We focus on two of these techniques in which productivity is encoded at the level of types: sized types and guarded recursion. A sized type is a type annotated with the number of unfoldings that elements of this type can undergo [17]. Sized types have been implemented in Agda and can be used in combination with coinductive records to specify coinductive types [4, 6], as exemplified by Abel and Chapman's work [3]. There also are other approaches to sized types. One of those is developed by Sacchini [23], which allows less terms to be typed compared to Agda's approach.

Guarded recursion [21] is a different approach where the type system is enhanced with a modality, called "later" and written $\triangleright$, encoding time delay in types. The later modality comes with a general fixpoint combinator for programming with productive recursive functions and allows the specification of guarded recursive types. These types can be used in combination with clock quantification to define coinductive types [8, 9, 12]. Currently there is no implementation of a calculus for guarded recursion.

Sized types and guarded recursion are different solutions to the same problem, but a thorough study of the relation between these two disciplines is still missing. In this paper we take a first step in this direction by showing how guarded recursion can be simulated in Agda using sized types.

Utilizing techniques for representing "type theory in type theory" [7, 13], we present an Agda formalization of the syntax of a simple type theory for guarded recursion. This object language, which we call **GTT**, is a variant of Atkey and McBride's type system for productive coprogramming [8] in which the clock context can contain at most one clock. The types of **GTT** include the aforementioned $\triangleright$ modality and a $\square$ modality, a nameless analogue of Atkey and McBride's universal clock quantification. Clouston *et al.* [14] studied a guarded variant of lambda calculus extended with a $\square$ operation, which they call "constant". Our object calculus differs from theirs in that our judgments are indexed by a clock context, which can be empty or containing exactly one clock. The design of **GTT** has the benefit of allowing a more appealing introduction rule for $\square$ than Clouston *et al.*'s. **GTT** also differs from Clouston *et al.*'s calculus in the class of definable guarded recursive types. We follow Atkey and McBride's approach and restrict the least fixpoint operator μ to act on strictly positive functors, while μ type former in Clouston *et al.*solely operates on functors in which all variables are guarded by an occurrence of the $\triangleright$ modality.

Afterwards we develop a categorical semantics for **GTT** using sized types. More precisely, we define a presheaf model where the type formers of simply typed lambda calculus are evaluated using the standard Kripke semantics. Typically, the semantics of a type theory for guarded recursion is given in the topos of trees or variations thereof [11, 19, 20]. Here, to clarify the connection between guarded recursion and sized types, we take the preorder of sizes as the indexing category of our presheaves. This means that types and contexts of **GTT** are interpreted as antitone sized types. The well-order relation on sizes is fundamental for constructing a terminating definition of the semantic fixpoint combinator.

The decision to restrict the clock context of **GTT** to contain at most one clock variable was dictated by our choice to use Agda with sized types as the metatheory. In Agda, it is possible to encode semantic clock contexts containing multiple clocks as lists of sizes or as functions from a finite interval of natural numbers into sizes. However, Agda's support for sized types is tailored to types depending on exactly one size, or on a finite but precise number of sizes, which makes it cumbersome to work with types depending on clock contexts containing an indefinite number of clocks. More technically, as it was privately communicated to us by Agda implementors, Agda's type inference only works properly for first-order size constraints. We believe that if future Agda releases add the capability of handling multiple

sizes, it would be possible to extend the semantics of **GTT** to include the full Atkey and McBride's type theory. We also believe that extending our formalization to the multiclock case, while technically challenging, would not be conceptually harder.

GTT is stricly less expressive than Atkey and McBride's calculus, since we are unable to implement nested coinductive types. But the one clock variant of the calculus still allows the construction of a large class of coinductive types and it is the subject of active research [14].

To summarize, the contributions of this paper are twofold:

1. We formalize syntax and semantics of a type theory for guarded recursion in Agda. This is the first such denotational model developed using an extension of Martin-Löf intensional type theory as the metalanguage, in contrast with the previous set-theoretic models of guarded recursion.
2. The interpretation of the characteristic features of guarded recursion crucially requires the employment of sized types. This shows that guarded recursion can be reduced to sized types and constitutes a stepping stone towards a clear understanding of the connections between different approaches to productive coprogramming.

This paper only include the essential parts of our Agda formalization. The full code is available at `https://github.com/niccoloveltri/agda-gtt`. The formalization uses Agda 2.5.4.2 and Agda standard library 0.16. The paper is extracted from a literate Agda file, which implies that all the displayed code passed Agda's type and termination checker.

The material is organized in the following way. In Section 2, we discuss the metatheory and we give an overview of sized types. In Section 3, we introduce the syntax of **GTT**, our object type theory for guarded recursion. Subsequently, we give a categorical semantics. In Section 4, we show how to implement presheaves over sizes and how to model the fragment of **GTT** corresponding to simply typed lambda calculus. In Section 5, we discuss the semantics of the guarded recursive and coinductive features. In Section 6, we prove the object language sound w.r.t. the categorical model, which in turn entails the consistency of the syntax. Finally, in Section 7, we draw conclusions and suggest future directions of work.

2 The Host Type Theory

We work in Martin-Löf type theory extended with functional extensionality, uniqueness of identity proofs (UIP), and sized types. Practically, we work in Agda, which supports sized types and where UIP holds by default. In this section, we give a brief overview of these principles and we introduce the basic Agda notation that we employ in our formalization.

We write $=$ for judgmental equality and $\equiv$ for propositional equality. Implicit arguments of functions are delimited by curly brackets. We write $\forall\ \{\Delta\}$ for an implicit argument Δ whose type can be inferred by Agda. We write Set, Set_1 and Set_2 for the first three universes of types. We write $\bot$ for the empty type.

We make extensive use of record types. These are like iterated Σ-types, in which each component, also called field, has been given a name. We open each record we introduce, which allows us to access a field by function application. For example, given a record type R containing a field f of type A, we have $\mathsf{f}\ \mathsf{R} : A$. We use Agda's copatterns for defining elements of a record type. If a record type R contains fields $\mathsf{f}_1 : A_1$ and $\mathsf{f}_2 : A_2$, we construct a term $\mathsf{r} : \mathsf{R}$ by specifying its components $\mathsf{f}_1\ \mathsf{r} : A_1$ and $\mathsf{f}_2\ \mathsf{r} : A_2$.

The principle of functional extensionality states that every two functions f and g in the same function space are equal whenever $f\ x$ and $g\ x$ are equal for all inputs x. This principle is not provable in Agda, so we need to postulate it. UIP states that all proofs of an identity are propositionally equal. Agda natively supports this principle, which is therefore derivable.

Agda also natively supports sized types [2, 6]. Intuitively, a sized type is a type annotated with an abstract ordinal indicating the number of possible unfoldings that can be performed of elements of the type. These abstract ordinals, called sizes, assist the termination checker in assessing the well-definedness of corecursive definitions.

In Agda, there is a type Size of sizes and a type Size< i of sizes strictly smaller than i. Every size j : Size< i is coerced to j : Size. The order relation on sizes is transitive, which means that whenever j : Size< i and k : Size< j, then k : Size< i. The order relation is also well-founded, which is used to define productive corecursive functions. There is a successor operation ↑ on sizes and a size ∞ such that i : Size< ∞ for all i. Lastly, we define a sized type to be a type indexed by Size.

3 The Object Type Theory

The object language we consider is simply typed lambda calculus extended with additional features for programming with guarded recursive and coinductive types. We call this language **GTT**. It is a variant of Atkey and McBride's type system for productive coprogramming [8]. In Atkey and McBride's calculus, all judgments are indexed by a clock context, which may contain several different clocks. They extend simply typed lambda calculus with two additional type formers: a modality ▷ for encoding time delay into types and universal quantification over clock variables ∀, which is used in combination with ▷ to specify coinductive types. In **GTT**, judgments are also indexed by a clock context, but in our case the latter can contain at most one clock variable κ. The type system of **GTT** also includes a ▷ modality, plus a box modality corresponding to Atkey and McBride's quantification over the clock variable κ.

In this section we introduce the syntax of **GTT** as in our Agda formalization. We give a more standard presentation of the calculus in Appendix A.

GTT is a type theory with explicit substitutions [1]. It comprises well-formed types and contexts, well-typed terms and substitutions, definitional equality of terms and of substitutions. All of them depend on a clock context. In **GTT**, the clock context can either be empty or contain a single clock κ.

```
data ClockCtx : Set where
  ∅ : ClockCtx
  κ : ClockCtx
```

We refer to types and contexts in the empty clock context as ∅-types and ∅-contexts respectively. Similarly, κ-types and κ-contexts are types and contexts depending on κ.

3.1 Types

The well-formed types of **GTT** include the unit type, products, coproducts, and function spaces. Notice that **1** is a ∅-type.

```
data Ty : ClockCtx → Set where
  1 : Ty ∅
  _⊠_ : ∀ {Δ} → Ty Δ → Ty Δ → Ty Δ
  _⊞_ : ∀ {Δ} → Ty Δ → Ty Δ → Ty Δ
  _→_ : ∀ {Δ} → Ty Δ → Ty Δ → Ty Δ
```

We include a modality ▷ as an operation on κ-types similar to the one in Atkey and McBride's system. There also is a nameless analogue of clock quantification, which we call "box" and denote by □ following [14]. The box modality takes a κ-type and returns a ∅-type.

The well-formed types of **GTT** include a weakening type former $\uparrow$, which maps $\varnothing$-types to κ-types.

```
▷ : Ty κ → Ty κ
□ : Ty κ → Ty ∅
↑ : Ty ∅ → Ty κ
```

Guarded recursive types are defined using a least fixpoint type former μ.

```
μ : ∀ {Δ} → Code Δ → Ty Δ
```

For μ to be well-defined, one typically limits its applicability to strictly positive functors. We instead consider a grammar Code Δ for functors, which has codes for constant functors, the identity, products, coproducts, and the later modality. Since there is a code for constant functors, the type family Code needs to be defined simultaneously with Ty.

```
data Code : ClockCtx → Set where
  ℂ : ∀ {Δ} → Ty Δ → Code Δ
  I : ∀ {Δ} → Code Δ
  _⊠_ : ∀ {Δ} → Code Δ → Code Δ → Code Δ
  _⊞_ : ∀ {Δ} → Code Δ → Code Δ → Code Δ
  ▷ : Code κ → Code κ
```

The constructors of Code Δ suffice for the specification of interesting examples of guarded recursive types such as streams. Nevertheless, it would not be complicated to add exponentials with constant domain and the box modality to the grammar. In addition, this grammar does not allow the possibility of defining nested inductive types.

3.2 Contexts

The well-formed contexts of **GTT** are built from the empty context, context extension, and context weakening. The last operation embeds $\varnothing$-contexts into κ-contexts. Notice that we are overloading the symbol $\uparrow$, which is used for both type and context weakening.

```
data Ctx : ClockCtx → Set where
  ▪ : ∀ {Δ} → Ctx Δ
  _,_ : ∀ {Δ} → Ctx Δ → Ty Δ → Ctx Δ
  ↑ : Ctx ∅ → Ctx κ
```

3.3 Terms

The well-typed terms and substitutions of **GTT** are defined simultaneously. Terms include constructors for variables and substitutions.

```
data Tm : ∀ {Δ} → Ctx Δ → Ty Δ → Set where
  var : ∀ {Δ} (Γ : Ctx Δ) (A : Ty Δ) → Tm (Γ , A) A
  sub : ∀ {Δ} {Γ₁ Γ₂ : Ctx Δ} {A : Ty Δ} → Tm Γ₂ A → Sub Γ₁ Γ₂ → Tm Γ₁ A
```

We have lambda abstraction and application, plus the usual introduction and elimination rules for the unit types, products, coproducts, and guarded recursive types. Here we only show the typing rules associated to function types and guarded recursive types. The function

eval evaluates codes in Code Δ into endofunctors on Ty Δ. We use a categorical combinator app for application. We derive the conventional application, taking additionally an element in Tm Γ A and returning an inhabitant of Tm Γ B, in Section 3.4.

```
lambda : ∀ {Δ} {Γ : Ctx Δ} {A B : Ty Δ} → Tm (Γ , A) B → Tm Γ (A → B)
app : ∀ {Δ} {Γ : Ctx Δ} {A B : Ty Δ} → Tm Γ (A → B) → Tm (Γ , A) B
cons : ∀ {Δ} {Γ : Ctx Δ} (P : Code Δ) → Tm Γ (eval P (μ P)) → Tm Γ (μ P)
primrec : ∀ {Δ} (P : Code Δ) {Γ : Ctx Δ} {A : Ty Δ}
  → Tm Γ (eval P (μ P ⊠ A) → A) → Tm Γ (μ P → A)
```

The later modality is required to be an applicative functor, which means that we have terms next and $\circledast$. The delayed fixpoint combinator dfix [9] allows defining productive recursive programs. The usual fixpoint returning a term in A instead of $\triangleright\ A$ is derivable.

```
next : {Γ : Ctx κ} {A : Ty κ} → Tm Γ A → Tm Γ (▷ A)
_⊛_ : {Γ : Ctx κ} {A B : Ty κ} → Tm Γ (▷ (A → B)) → Tm Γ (▷ A) → Tm Γ (▷ B)
dfix : {Γ : Ctx κ} {A : Ty κ} → Tm Γ (▷ A → A) → Tm Γ (▷ A)
```

We have introduction and elimination rules for the $\Box$ modality. The rule box is the analogue in **GTT** of Atkey and McBride's rule for clock abstraction [8]. Notice that box can only be applied to terms of type A over a weakened context $\uparrow\ \Gamma$. This is analogous to Atkey and McBride's side condition requiring the universally quantified clock variable to not appear freely in the context Γ. Similarly, the rule unbox corresponds to clock application. The operation force is used for removing occurrences of $\triangleright$ protected by the $\Box$ modality.

```
box : {Γ : Ctx ∅} {A : Ty κ} → Tm (↑ Γ) A → Tm Γ (□ A)
unbox : {Γ : Ctx ∅} {A : Ty κ} → Tm Γ (□ A) → Tm (↑ Γ) A
force : {Γ : Ctx ∅} {A : Ty κ} → Tm Γ (□ (▷ A)) → Tm Γ (□ A)
```

The introduction and elimination rules for type weakening say that elements of Tm Γ A can be embedded in Tm $(\uparrow\ \Gamma)$ $(\uparrow\ A)$ and vice versa.

```
up : {Γ : Ctx ∅} {A : Ty ∅} → Tm Γ A → Tm (↑ Γ) (↑ A)
down : {Γ : Ctx ∅} {A : Ty ∅} → Tm (↑ Γ) (↑ A) → Tm Γ A
```

Atkey and McBride assume the existence of certain type equalities [8]. Møgelberg, who works in a dependently typed setting, considers similar type isomorphisms [20]. In **GTT**, we follow the second approach. This means that we do not introduce an equivalence relation on types specifying which types should be considered equal, as in Chapman's object type theory [13]. Instead, we include additional term constructors corresponding to functions underlying the required type isomorphisms. For example, the clock irrelevance axiom formulated in our setting states that every $\varnothing$-type A is isomorphic to $\Box\ (\uparrow\ A)$. This is obtained by adding to Tm a constructor □const.

```
□const : {Γ : Ctx ∅} (A : Ty ∅) → Tm Γ (□ (↑ A) → A)
```

A function const□ A in the other direction is derivable. When defining definitional equality on terms, described in Section 3.5, we ask for □const and const□ to be each other inverses. The other type isomorphisms, listed in Appendix A are constructed in a similar way.

3.4 Substitutions

For substitutions, we need the canonical necessary operations [7, 13]: identity and composition of substitutions, the empty substitution, the extension with an additional term, and the projection which forgets the last term.

```
data Sub : ∀ {Δ} → Ctx Δ → Ctx Δ → Set where
  ε : ∀ {Δ} (Γ : Ctx Δ) → Sub Γ ▪
  id : ∀ {Δ} (Γ : Ctx Δ) → Sub Γ Γ
  _,_ : ∀ {Δ} {Γ₁ Γ₂ : Ctx Δ} {A : Ty Δ} → Sub Γ₁ Γ₂ → Tm Γ₁ A → Sub Γ₁ (Γ₂ , A)
  _∘_ : ∀ {Δ} {Γ₁ Γ₂ Γ₃ : Ctx Δ} → Sub Γ₂ Γ₃ → Sub Γ₁ Γ₂ → Sub Γ₁ Γ₃
  pr : ∀ {Δ} {Γ₁ Γ₂ : Ctx Δ} {A : Ty Δ} → Sub Γ₁ (Γ₂ , A) → Sub Γ₁ Γ₂
```

We also add rules for embedding substitutions between $\varnothing$-contexts into substitutions between κ-contexts and vice versa.

```
  up : {Γ₁ Γ₂ : Ctx ∅} → Sub Γ₁ Γ₂ → Sub (↑ Γ₁) (↑ Γ₂)
  down : {Γ₁ Γ₂ : Ctx ∅} → Sub (↑ Γ₁) (↑ Γ₂) → Sub Γ₁ Γ₂
```

In addition, we require the existence of two context isomorphisms. The context $\uparrow \blacksquare$ needs to be isomorphic to $\blacksquare$ and $\uparrow (\Gamma , A)$ needs to be isomorphic to $\uparrow \Gamma , \uparrow A$. For both of them, we add a constructor representing the underlying functions.

```
  ▪↑ : Sub ▪ (↑ ▪)
  ,↑ : (Γ : Ctx ∅) (A : Ty ∅) → Sub (↑ Γ , ↑ A) (↑ (Γ , A))
```

An element $\uparrow\blacksquare$ in Sub $(\uparrow \blacksquare)$ $\blacksquare$ is derivable. In the definitional equality on substitutions, we ask for $\blacksquare\uparrow$ and $\uparrow\blacksquare$ to be each other inverses. We proceed similarly with $,\uparrow$.

Using the term constructor sub, we can derive a weakening operation for terms and the conventional application combinator.

```
wk : ∀ {Δ} {Γ : Ctx Δ} {A B : Ty Δ} → Tm Γ B → Tm (Γ , A) B
wk {Δ} {Γ} {A} x = sub x (pr (id (Γ , A)))

_$_ : ∀ {Δ} {Γ : Ctx Δ} {A B : Ty Δ} → Tm Γ (A → B) → Tm Γ A → Tm Γ B
_$_ {Δ} {Γ} f x = sub (app f) (id Γ , x)
```

3.5 Definitional equalities

Definitional equalities on terms and substitutions are defined simultaneously. Here we only discuss equality on terms and we refer to the formalization for the equality on substitutions.

```
data _∼_ : ∀ {Δ} {Γ : Ctx Δ} {A : Ty Δ} (t₁ t₂ : Tm Γ A) → Set where
```

The term equality includes rules for equivalence, congruence, and substitution. There also are β and η rules for the type formers. We only show the ones associated to the $\Box$ modality here. The rules state that box and unbox are each other's inverses.

```
□-β : ∀ {Γ} {A} (t : Tm (↑ Γ) A) → unbox (box t) ∼ t
□-η : ∀ {Γ} {A} (t : Tm Γ (□ A)) → box (unbox t) ∼ t
```

We include definitional equalities stating that $\triangleright$ is an applicative functor w.r.t. the operations next and $\circledast$. Furthermore, the delayed fixpoint combinator dfix must satisfy its characteristic unfolding equation. We refer to Møgelberg's paper [20] for a complete list of the required definitional equalities for $\triangleright$ and $\Box$.

For the type isomorphisms, we require term equalities exhibiting that certain maps are mutual inverses. For example, we have the following two equalities about $\Box$const and const$\Box$:

```
const□const : ∀ {Γ} {A} (t : Tm Γ (□ (↑ A))) → const□ A $ (□const A $ t) ∼ t
□const□ : ∀ {Γ} {A} (t : Tm Γ A) → □const A $ (const□ A $ t) ∼ t
```

The last group of term equalities describes the relationship between the weakening operations up and down and other term constructors. Here we omit their description and we refer the interested reader to the Agda formalization.

3.6 Example: Streams

We give a taste of how to program with streams in **GTT**. Our implementation is based on Atkey and McBride's approach to coinductive types [8]. To define a type of streams, we first define guarded streams over a $\varnothing$-type A. It is the least fixpoint of the functor with code $\mathbb{C}$ ($\uparrow$ A) $\boxtimes$ $\triangleright$ I.

```
F : Ty ∅ → Code κ
F A = ℂ (↑ A) ⊠ ▷ I
```

```
g-Str : Ty ∅ → Ty κ
g-Str A = μ (F A)
```

The usual type of streams over A is then obtained by applying the $\Box$ modality to g-Str A.

```
Str : Ty ∅ → Ty ∅
Str A = □ (g-Str A)
```

We compute the head and tail of a stream using a function decons, which destructs an element of an inductive type. The term decons is the inverse of cons and it is derivable using primrec. Note that in both cases we need to use unbox, because of the application of the box modality in the definition of Str. For the tail, we also use box and force. The operations π_1 and π_2 are the projections associated to the product type former $\boxtimes$.

```
hd : {Γ : Ctx ∅} {A : Ty ∅} → Tm Γ (Str A) → Tm Γ A
hd xs = down (π₁ (decons (unbox xs)))
```

```
tl : {Γ : Ctx ∅} {A : Ty ∅} → Tm Γ (Str A) → Tm Γ (Str A)
tl xs = force (box (π₂ (decons (unbox xs))))
```

Given a **GTT** term a of type A, we can construct the constant guarded stream over a using the fixpoint combinator.

```
g-const-str : {Γ : Ctx ∅} {A : Ty ∅} → Tm Γ A → Tm (↑ Γ) (g-Str A)
g-const-str {Γ} {A} a = fix (lambda (cons (F A) [ wk (up a) & var (↑ Γ) (▷ (g-Str A)) ]))
```

The constant stream over a is obtained by boxing the guarded stream g-const a.

```
const-str : {Γ : Ctx ∅} {A : Ty ∅} → Tm Γ A → Tm Γ (Str A)
const-str a = box (g-const-str a)
```

In our Agda formalization, we also construct a function removing the elements at even indices out of a given stream, which is an example of a non-causal function.

4 Categorical Semantics

Next we give a categorical semantics for the calculus introduced in Section 3. We take inspiration from Møgelberg's model [20] in which a (simple) type in a clock context containing n clocks is interpreted as a presheaf in $\mathbf{Set}^{(\omega^n)^{\mathrm{op}}}$, where ω is the preorder of ordered natural numbers. In our model, we replace the category ω with the preorder of sizes. Moreover, we only consider the cases where either the clock context is empty or it contains exactly one clock. This means that a type in **GTT** is either interpreted as an element of Set or as a presheaf over Size. In this section, we show how to implement presheaves and how to model the fragment of **GTT** corresponding to simply typed lambda calculus. The interpretation of the guarded recursive and coinductive features of **GTT** is given in Section 5.

4.1 Presheaves

Presheaves are defined as a record PSh. The fields Obj and Mor represent the actions on objects and morphisms respectively, while MorId and MorComp are the functor laws. The type Size< (↑ i) contains sizes smaller or equal than i. In the type of MorId we use the reflexivity of the order on sizes, which means i : Size< (↑ i). In the type of MorComp we use transitivity.

```
record PSh : Set₁ where
  field
    Obj : Size → Set
    Mor : (i : Size) (j : Size< (↑ i)) → Obj i → Obj j
    MorId : {i : Size} {x : Obj i} → Mor i i x ≡ x
    MorComp : {i : Size} {j : Size< (↑ i)} {k : Size< (↑ j)} {x : Obj i}
      → Mor i k x ≡ Mor j k (Mor i j x)
```

Beside presheaves, we also need natural transformations. These are defined as a record NatTrans, consisting of a map nat-map and a proof of the usual commutativity requirement.

```
record NatTrans (P Q : PSh) : Set where
  field
    nat-map : (i : Size) → Obj P i → Obj Q i
    nat-com : (i : Size) (j : Size< (↑ i)) (x : Obj P i)
      → Mor Q i j (nat-map i x) ≡ nat-map j (Mor P i j x)
```

Products and sums of presheaves are defined pointwise. More precisely, we define the product as follows

```
ProdObj : Size → Set
ProdObj i = Obj P i × Obj Q i
```

The sum is defined similarly. The weakening type former ↑ of **GTT** is modeled using the constant presheaf, which we denote by Const. Function spaces are defined as the exponential of presheaves. The action on a size i of this presheaf consists of natural transformations restricted to sizes smaller or equal than i.

```
record ExpObj (P Q : PSh) (i : Size) : Set where
  field
    fun : (j : Size< (↑ i)) → Obj P j → Obj Q j
    funcom : (j : Size< (↑ i)) (k : Size< (↑ j)) (x : Obj P j)
      → Mor Q j k (fun j x) ≡ fun k (Mor P j k x)
```

All in all, we get an operation Exp : PSh → PSh → PSh.

4.2 Modelling Simple Types

To interpret the fragment of **GTT** corresponding to simply typed lambda calculus, we use Kripke semantics [18]. Semantic judgments, similar to their syntactic counterparts, are indexed by a clock context. We reuse the type ClockCtx for the semantic clock contexts. The semantic variable contexts are sets if the clock context is empty, and they are presheaves otherwise.

```
SemCtx : ClockCtx → Set₁
SemCtx ∅ = Set
SemCtx κ = PSh
```

Note that **GTT** is a simple type theory, thus types do not depend on contexts. For this reason, we define the type SemTy of semantic types in the same way as SemCtx.

The semantic terms of type A in context Γ are functions from Γ to A if the clock context is empty, and they are natural transformations between Γ and A otherwise.

```
SemTm : {Δ : ClockCtx} (Γ : SemCtx Δ) (A : SemTy Δ) → Set
SemTm {∅} Γ A = Γ → A
SemTm {κ} Γ A = NatTrans Γ A
```

Since **GTT** is a type theory with explicit substitutions, we must provide an interpretation for them as well. Semantic substitutions are maps between contexts and we define the type SemSub in the same way as SemTm. Definitional equality of semantic terms and substitutions is modeled as propositional equality.

We also need to provide a semantic version of the context operations, the simple type formers and the operations on substitutions. Since their definitions are standard, we do not discuss them in detail. For each of them, we need to make a case distinction based on the clock context. For example, the empty variable context ▪ is interpreted as the unit type in the clock context ∅, and it is interpreted as the terminal presheaf in the clock context κ. We use the operations on presheaves defined in Section 4.1 to interpret simple type formers. In the next section, we use the interpretation of function types, whose action of objects in the clock context κ is given by ExpObj. This is denoted by $A \Rightarrow B$ for semantic types A and B.

```
_⇒_ : ∀ {Δ} (A B : SemTy Δ) → SemTy Δ
_⇒_ {∅} A B = A → B
_⇒_ {κ} A B = Exp A B
```

5 Modelling Guarded Recursion

In this section, we add the required guarded recursive and coinductive features to the denotational semantics. We start by defining the semantic box modality together with its introduction and elimination rule. Then we construct the semantic later modality. We show how to define the fixpoint combinator and the force operation using sized types. In the end, we discuss how to model guarded recursive types.

5.1 Context Weakening and the Box Modality

Similarly to the weakening type former ↑, the weakening context former ↑ is modeled using the constant presheaf Const.

```
⇑ : SemCtx ∅ → SemCtx κ
⇑ Γ = Const Γ
```

Møgelberg models universal clock quantification by taking limits [20]. We define the semantic box modality ■ similarly: given a presheaf A, we take ■ A to be the limit of A. Formally, the limit of A is constructed as a record with two fields. The field ■cone is given by a family $f\ i$ in Obj $A\ i$ for each size i. The field ■com is a proof that the restriction of $f\ i$ to a size j smaller than i is equal to $f\ j$.

```
record ■ (A : SemTy κ) : SemTy ∅ where
  field
    ■cone : (i : Size) → Obj A i
    ■com : (i : Size) (j : Size< (↑ i)) → Mor A i j (■cone i) ≡ ■cone j
```

The semantic box modality is right adjoint to context weakening. In other words, the types Tm (⇑ Γ) A and Tm Γ (■ A) are isomorphic for all Γ and A. The function underlying the isomorphism is sem-box and its inverse is sem-unbox, modeling box and unbox respectively.

```
sem-box : (Γ : SemCtx ∅) (A : SemTy κ) (t : SemTm (⇑ Γ) A) → SemTm Γ (■ A)
■cone (sem-box Γ A t x) i      = nat-map t i x
■com (sem-box Γ A t x) i j     = nat-com t i j x

sem-unbox : (Γ : SemCtx ∅) (A : SemTy κ) (t : SemTm Γ (■ A)) → SemTm (⇑ Γ) A
nat-map (sem-unbox Γ A t) i x  = ■cone (t x) i
nat-com (sem-unbox Γ A t) i j x = ■com (t x) i j
```

5.2 The Later Modality

The semantic later modality is an operation ▶ on semantic κ-types. Recall that the later modality in the topos of trees [11] is defined as

$$(\blacktriangleright\ A)(0) = \{*\}$$
$$(\blacktriangleright\ A)(n+1) = A(n)$$

We cannot directly replicate the latter in our setting since the preorder of sizes does not possess a least element. However, we know from [11] that the definition above is equivalent to $(\blacktriangleright\ A)(n) = \lim_{k<n} A(k)$. Adapting this to our setting leads to the following action of ▶ on objects:

```
record ▶ObjTry (A : SemTy κ) (i : Size) : Set where
  field
    ▶cone : (j : Size< i) → Obj A j
    ▶com : (j : Size< i) (k : Size< (↑ j)) → Mor A j k (▶cone j) ≡ ▶cone k
```

However, with this definition, we are unable to implement a terminating semantic fixpoint combinator. Later in this section we discuss why this is the case.

There are several ways to modify the above definition and implement a terminating fixpoint operation. A possible solution, which was suggested to us by Andrea Vezzosi, is using an inductive analogue of the predicate Size<, which we call SizeLt.

```
data SizeLt (i : Size) : Set where
  [_] : Size< i → SizeLt i
```

The type SizeLt is a mechanism for suspending computations. If we define a function f of type $(j : \mathsf{SizeLt}\ i) \to \mathsf{Obj}\ A\ j$ by pattern matching, then $f\ j$ does not reduce unless j is of the form $[\ j'\ ]$ for some $j' : \mathsf{Size<}\ i$. This simple observation turns out to be essential for a terminating implementation of the fixpoint combinator.

From an inhabitant of SizeLt, we obtain an actual size by pattern matching.

```
size : ∀ {i} → SizeLt i → Size
size [ j ] = j
```

Furthermore, functions with domain $\mathsf{SizeLt}\ i$ can be specified using functions on $\mathsf{Size<}\ i$.

```
elimLt : {A : Size → Set₁} {i : Size} → ((j : Size< i) → A j)
  → (j : SizeLt i) → A (size j)
elimLt f [ j ] = f j
```

We define the action on objects of the semantic later modality similarly to ▶ObjTry but with SizeLt in place of Size<. Before we do so, we introduce two auxiliary functions, which turn out to be handy when modeling guarded recursive types. The first is a function Later, which instead of a semantic κ-type, takes a sized type as its input. Its definition is the same as the type of the field ▶cone in ▶ObjTry but with Size< replaced by SizeLt.

```
Later : (Size → Set) → Size → Set
Later A i = (j : SizeLt i) → A (size j)
```

The second auxiliary function is LaterLim. It takes as input a sized type A together with a proof that it is antitone. Again its definition is the same as the type of the field ▶com in ▶ObjTry but with two applications of elimLt and Size< replaced by SizeLt.

```
LaterLim : (A : Size → Set) (m : (i : Size) (j : Size< (↑ i)) → A i → A j)
  → (i : Size) (x : Later A i) → Set
LaterLim A m i x = (j : SizeLt i)
  → elimLt (λ { j' → (k : SizeLt (↑ j'))
    → elimLt (λ k' → m j' k' (x [ j' ]) ≡ x [ k' ]) k }) j
```

Putting everything together, we obtain the following definition of the object part of the semantic later modality ▶. We refer to the Agda formalization for the action on the morphisms and the functor laws.

```
record ▶Obj (A : SemTy κ) (i : Size) : Set where
  field
    ▶cone : Later (Obj A) i
    ▶com : LaterLim (Obj A) (Mor A) i ▶cone
```

We omit the semantic equivalents of next and ⊛. To interpret the delayed fixpoint combinator dfix, we introduce an auxiliary term $\mathsf{sem\text{-}dfix}_1$, for which we only show how to construct the field ▶cone. This is defined using self-application.

```
sem-dfix₁ : (A : SemTy κ) (i : Size) → ExpObj (▶ A) A i → ▶Obj A i
▶cone (sem-dfix₁ A i f) [ j ] = fun f j (sem-dfix₁ A j f)
```

This definition is accepted by Agda's termination checker for two reasons:

- every recursive call is applied to a strictly smaller size;
- the usage of SizeLt in place of Size< in the definition of Later prevents indefinite unfolding, which would have happened if we used ▶ObjTry instead of ▶Obj.

In fact, if we would replace ▶Obj by ▶ObjTry as the return type of sem-dfix₁ while keeping the same definition (with j in place of [j]), we would obtain the following non-terminating sequence of reductions:

```
▶cone (sem-dfix₁ A i f)
  = λ j → fun f j (sem-dfix₁ A j f)
  = λ j → fun f j (record { ▶cone = λ k → fun f k (sem-dfix₁ A k f) ; ▶com = ... })
  = ...
```

The termination of the ▶com component of sem-dfix₁ additionally relies on the presence of elimLt in the definition of LaterLim.

The field nat-map of sem-dfix is defined using sem-dfix₁. We omit the nat-com component.

```
sem-dfix : (Γ : SemCtx κ) (A : SemTy κ) (f : SemTm Γ (▶ A ⇒ A)) → SemTm Γ (▶ A)
nat-map (sem-dfix Γ A f) i γ = sem-dfix₁ A i (nat-map f i γ)
```

Finally, we show how to interpret force. To this aim, we introduce an auxiliary function sem-force', which, given a type A and an inhabitant t of ■(▶ A), returns a term in ■ A. For the field ■cone of sem-force' A t, we are required to construct an element of Obj A i for each size i. Notice that ■cone t, when applied to a size i', gives a term t' of type ▶Obj A i'. Furthermore, the component ▶cone of t', when applied to a size j' smaller than i', returns a term of type Obj A j'. Hence, in order to construct the required inhabitant of Obj A i, it suffices to find a size j greater than i. An option for such a size j is ∞. The field ■com is defined in a similar way.

```
sem-force' : (A : SemTy κ) → ■ (▶ A) → ■ A
■cone (sem-force' A t) i = ▶cone (■cone t ∞) [ i ]
■com (sem-force' A t) i j = ▶com (■cone t ∞) [ i ] [ j ]
```

The semantic force operation follows immediately from sem-force'.

```
sem-force : (Γ : SemCtx ∅) (A : SemTy κ) (t : SemTm Γ (■ (▶ A))) → SemTm Γ (■ A)
sem-force Γ A t x = sem-force' A (t x)
```

5.3 Guarded Recursive Types

For semantic guarded recursive types, we introduce a type of semantic codes for functors. We cannot reutilize the syntactic grammar Code since the code for the constant functor should depend on SemTy rather than Ty. Instead we use the following definition.

```
data SemCode : ClockCtx → Set₁ where
  ℂ : ∀ {Δ} → SemTy Δ → SemCode Δ
  I : ∀ {Δ} → SemCode Δ
```

```
_⊞_ : ∀ {Δ} → SemCode Δ → SemCode Δ → SemCode Δ
_⊠_ : ∀ {Δ} → SemCode Δ → SemCode Δ → SemCode Δ
▶ : SemCode κ → SemCode κ
```

In the remainder of this section, we only discuss guarded recursive κ-types. The interpretation of μ in the clock context $\varnothing$ is standard and therefore omitted. Given a semantic code P, our goal is to construct the action on objects and morphisms of a presheaf mu-κ P.

A first naïve attempt would be to define the action on objects muObj P as the initial algebra of sem-eval P, where sem-eval evaluates a code as an endofunctor on SemTy κ. This means defining muObj P as an inductive type with one constructor taking in input sem-eval P (muObj P). This idea does not work, since muObj P is a sized type while sem-eval P expects a semantic κ-type.

Another possibility would be to define muObj P by induction on P. However, there is a problem when we arrive at the I constructor. In this case, we would like to make a recursive call to muObj applied to the original code P, which is unavailable at this point. We solve the issue by introducing an auxiliary inductive type family muObj', which depends on two codes instead of one. The first code is the original one used to define the guarded recursive type and we do induction on the second one. Then we define muObj P to be muObj' P P.

The constructors of muObj' P Q follow the structure of Q. If Q is a product we have a pairing constructor, if it is a sum we have the two injections. When Q is the code for the identity functor, we make a recursive call. For the ▶ case, we have a constructor later taking two arguments with the same types as the two fields of ▶Obj. Since LaterLim depends both on a sized type and a proof that it is antitone, we need to define muObj' mutually with its own proof of antitonicity muMor'. This construction works since Later and LaterLim take in input part of the data of a presheaf and they crucially do not depend on the functor laws.

```
mutual
  data muObj' (P : SemCode κ) : SemCode κ → Size → Set where
    const : {X : PSh} {i : Size} → Obj X i → muObj' P (ℂ X) i
    rec : ∀{i} → muObj' P P i → muObj' P I i
    _,_ : ∀{Q R i} → muObj' P Q i → muObj' P R i → muObj' P (Q ⊠ R) i
    in₁ : ∀{Q R i} → muObj' P Q i → muObj' P (Q ⊞ R) i
    in₂ : ∀{Q R i} → muObj' P R i → muObj' P (Q ⊞ R) i
    later : ∀{Q i} (x : Later (muObj' P Q) i)
      → LaterLim (muObj' P Q) (muMor' P Q) i x → muObj' P (▶ Q) i
```

```
muMor' : (P Q : SemCode κ) (i : Size) (j : Size< (↑ i)) → muObj' P Q i → muObj' P Q j
muMor' P (ℂ X) i j (const x) = const (Mor X i j x)
muMor' P I i j (rec x) = rec (muMor' P P i j x)
muMor' P (Q ⊠ R) i j (x , y) = muMor' P Q i j x , muMor' P R i j y
muMor' P (Q ⊞ R) i j (in₁ x) = in₁ (muMor' P Q i j x)
muMor' P (Q ⊞ R) i j (in₂ x) = in₂ (muMor' P R i j x)
muMor' P (▶ Q) i j (later x p) = later x (λ { [ k ] → p [ k ] })
```

We define muMor P to be muMor' P P. Since muMor preserves the identity and composition, we get a presheaf mu-κ P for each P. This is used to interpret μ in the clock context κ. We write mu for the interpretation of μ in a general clock context.

6 Soundness and Consistency

We now define the notion of interpretation of **GTT**. To interpret types, one must give a type of semantical types and a function mapping each syntactic type to its semantical counterpart. Similarly for contexts, terms, substitutions and definitional equalities. This leads to the following record where we only show the fields related to types.

```
record interpret-syntax : Set₂ where
  field
    semTy : ClockCtx → Set₁
    _⟦_⟧Ty : ∀ {Δ} → Ty Δ → semTy Δ
```

Now we prove **GTT** sound w.r.t. the categorical semantics. We only show the interpretation of the types whose semantics were explicitly discussed in Sections 4 and 5. Since syntactic types are defined mutually with codes, the interpretation of types ⟦_⟧type has to be defined simultaneously with the interpretation of codes ⟦_⟧code, which we omit here.

```
⟦_⟧type : ∀ {Δ} → Ty Δ → SemTy Δ
⟦ A → B ⟧type = ⟦ A ⟧type ⇒ ⟦ B ⟧type
⟦ ▷ A ⟧type = ▶ ⟦ A ⟧type
⟦ □ A ⟧type = ■ ⟦ A ⟧type
⟦ μ P ⟧type = mu ⟦ P ⟧code
```

Similarly, Ctx, Tm and Sub are mapped into SemCtx, SemTm and SemSub respectively. Definitional equality of terms is interpreted as Agda's propositional equality, the same for definitional equality of substitutions.

```
sem : interpret-syntax
semTy sem = SemTy
_⟦_⟧Ty sem = ⟦_⟧type
```

Using the interpetation sem, we can show that **GTT** is consistent, by which we mean that not every definitional equality is deducible. We first define a type bool : Ty ∅ as **1** ⊞ **1** and two terms TRUE and FALSE as in_1 tt and in_2 tt respectively, where in_1 and in_2 are the two constructors of ⊞. We say that **GTT** is consistent if TRUE and FALSE are not definitionally equal.

```
consistent : Set
consistent = TRUE ∼ FALSE → ⊥
```

This is proved by noticing that if TRUE were definitionally equal to FALSE, then their interpretations in sem would be equal. However, they are interpreted as inj_1 tt and inj_2 tt respectively, and those are unequal. Hence, **GTT** is consistent.

7 Conclusions and Future Work

We presented a simple type theory for guarded recursion that we called **GTT**. We formalized its syntax and semantics in Agda. Sized types were employed to interpret the characteristic features of guarded recursion. From this, we conclude that guarded recursion can be simulated using sized types. We greatly benefited from the fact that sized types constitute a native feature of Agda, so we were able to fully develop our theory inside a proof assistant.

This work can be extended in several different directions. Various type theories for guarded recursion in the literature include dependent types. Currently, there exist two disciplines for mixing dependent types with guarded recursion: delayed substitutions [12] and ticks [9]. It would be interesting to extend the syntax and semantics of **GTT** with dependent types following one of these two methods.

Abel and Vezzosi [5] formalized a simple type theory extended with the later modality in Agda. They focus on operational properties of the calculus and give a certified proof of strong normalization. As future work, we plan to investigate the metatheory and the dynamic behavior of **GTT** by formalizing a type checker and a normalization algorithm. This would also set up the basis for a usable implementation of **GTT**.

In **GTT** we restrict the least fixpoint operator μ to act exclusively on strictly positive functors. This restriction is already present in Atkey and McBride's calculus, which is aimed at encoding coinductive types using the later modality and it does not allow solving general guarded recursive domain equations. **GTT** is a variant of Atkey and McBride's type theory, therefore the similar restriction to stricly positive functors. From the formalization perspective, this restriction allows us to use Agda's inductive types to model the guarded recursive types of GTT, as shown in Section 5.3. In future work, we plan to extend **GTT** with the possibility of solving general guarded recursive domain equations. This could be done in two ways: extending the language with a universe, which allows the encoding of guarded recursive types as fixpoints in the universe as shown e.g. by Møgelberg [20]; or considering a μ type former which, besides stricly positive functors, also operates on functors where all variables are guarded by an occurrence of the later modality. In the second option we have to consider strictly positive functors to encode usual inductive types, such as the natural numbers.

Finally, we would like to understand whether sized types can be simulated using guarded recursion. This could either be done by modeling sized types in a topos of trees-like category [11, 19] or via the encoding of a type theory with sized types into a dependent type theory for guarded recursion such as Clocked Type Theory [9].

References

1 Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit Substitutions. *J. Funct. Program.*, 1(4):375–416, 1991. doi:10.1017/S0956796800000186.

2 Andreas Abel. MiniAgda: Integrating Sized and Dependent Types. In *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010, Edinburgh, UK, July 15, 2010*, pages 18–32, 2010. URL: http://www.easychair.org/publications/paper/51657.

3 Andreas Abel and James Chapman. Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.*, pages 51–67, 2014. doi:10.4204/EPTCS.153.4.

4 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming Infinite Structures by Observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 27–38, 2013. doi:10.1145/2429069.2429075.

5 Andreas Abel and Andrea Vezzosi. A Formalized Proof of Strong Normalization for Guarded Recursive Types. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, pages 140–158, 2014. doi:10.1007/978-3-319-12736-1_8.

6 Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. Normalization by Evaluation for Sized Dependent Types. *PACMPL*, 1(ICFP):33:1–33:30, 2017. doi:10.1145/3110277.

7 Thorsten Altenkirch and Ambrus Kaposi. Type Theory in Type Theory using Quotient Inductive Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29, 2016. doi:10.1145/2837614.2837638.

8 Robert Atkey and Conor McBride. Productive Coprogramming with Guarded Recursion. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 197–208, 2013.

9 Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The Clocks are Ticking: No More Delays! In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005097.

10 Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq Proof Assistant Reference Manual: Version 6.1*. PhD thesis, Inria, 1997.

11 Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:1)2012.

12 Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E Møgelberg, and Lars Birkedal. Guarded Dependent Type Theory with Coinductive Types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 20–35. Springer, 2016.

13 James Chapman. Type Theory Should Eat Itself. *Electr. Notes Theor. Comput. Sci.*, 228:21–36, 2009. doi:10.1016/j.entcs.2008.12.114.

14 Ranald Clouston, Ales Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 407–421, 2015. doi:10.1007/978-3-662-46678-0_26.

15 Thierry Coquand. Infinite Objects in Type Theory. In *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, pages 62–78, 1993. doi:10.1007/3-540-58085-9_72.

16 Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, Declaratively. In *Mathematics of Program Construction, 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010. Proceedings*, pages 100–118, 2010. doi:10.1007/978-3-642-13321-3_8.

17 John Hughes, Lars Pareto, and Amr Sabry. 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*, pages 410–423, 1996. doi:10.1145/237721.240882.

18 Saunders MacLane and Ieke Moerdijk. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer Science & Business Media, 1992.

19 Bassel Mannaa and Rasmus Ejlers Møgelberg. The Clocks They Are Adjunctions Denotational Semantics for Clocked Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 23:1–23:17, 2018. doi:10.4230/LIPIcs.FSCD.2018.23.

20 Rasmus Ejlers 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, Vienna, Austria, July 14 - 18, 2014*, pages 71:1–71:10, 2014. doi:10.1145/2603088.2603132.

21 Hiroshi Nakano. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*, pages 255–266, 2000. doi:10.1109/LICS.2000.855774.

22 Ulf Norell. Dependently Typed Programming in Agda. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2, 2009.

23 Jorge Luis Sacchini. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 233–242, 2013. doi:10.1109/LICS.2013.29.

A Syntax of GTT

Contexts

$$\frac{}{- \vdash_\Delta} \qquad \frac{\Gamma \vdash_\Delta \quad \Gamma \vdash_\Delta A \text{ type}}{\Gamma, x : A \vdash_\Delta} \qquad \frac{\Gamma \vdash_\varnothing}{\uparrow\Gamma \vdash_\kappa}$$

Codes

$$\frac{}{\Gamma \vdash_\Delta \mathrm{I} \text{ code}} \qquad \frac{\Gamma \vdash_\Delta A \text{ type}}{\Gamma \vdash_\Delta A \text{ code}}$$

$$\frac{\Gamma \vdash_\Delta P \text{ code} \quad \Gamma \vdash_\Delta Q \text{ code}}{\Gamma \vdash_\Delta P \times Q \text{ code}} \qquad \frac{\Gamma \vdash_\Delta P \text{ code} \quad \Gamma \vdash_\Delta Q \text{ code}}{\Gamma \vdash_\Delta P + Q \text{ code}} \qquad \frac{\Gamma \vdash_\kappa P \text{ code}}{\Gamma \vdash_\kappa \triangleright P \text{ code}}$$

Types

$$\frac{}{\Gamma \vdash_\varnothing 1 \text{ type}} \qquad \frac{\Gamma \vdash_\Delta A \text{ type} \quad \Gamma \vdash_\Delta B \text{ type}}{\Gamma \vdash_\Delta A \times B \text{ type}}$$

$$\frac{\Gamma \vdash_\Delta A \text{ type} \quad \Gamma \vdash_\Delta B \text{ type}}{\Gamma \vdash_\Delta A + B \text{ type}} \qquad \frac{\Gamma \vdash_\Delta A \text{ type} \quad \Gamma \vdash_\Delta B \text{ type}}{\Gamma \vdash_\Delta A \to B \text{ type}}$$

$$\frac{\uparrow\Gamma \vdash_\kappa A \text{ type}}{\Gamma \vdash_\varnothing \Box A \text{ type}} \qquad \frac{\Gamma \vdash_\varnothing A \text{ type}}{\uparrow\Gamma \vdash_\kappa \uparrow A \text{ type}}$$

$$\frac{\Gamma \vdash_\kappa A \text{ type}}{\Gamma \vdash_\kappa \triangleright A \text{ type}} \qquad \frac{\Gamma \vdash_\Delta P \text{ code}}{\Gamma \vdash_\Delta \mu P \text{ type}}$$

Substitutions

$$\frac{\Gamma \vdash_\Delta}{\vdash_\Delta \varepsilon : \Gamma \to -} \qquad \frac{\Gamma \vdash_\Delta}{\vdash_\Delta \mathsf{id} : \Gamma \to \Gamma} \qquad \frac{\vdash_\Delta s : \Gamma_1 \to \Gamma_2, A}{\vdash_\Delta \mathsf{pr}\, s : \Gamma_1 \to \Gamma_2}$$

$$\frac{\vdash_\Delta s : \Gamma_1 \to \Gamma_2 \quad \Gamma_1 \vdash_\Delta t : A}{\vdash_\Delta s, t : \Gamma_1 \to \Gamma_2, A} \qquad \frac{\vdash_\Delta s_1 : \Gamma_1 \to \Gamma_2 \quad \vdash_\Delta s_2 : \Gamma_2 \to \Gamma_3}{\vdash_\Delta s_2 \circ s_1 : \Gamma_1 \to \Gamma_3}$$

$$\frac{\vdash_\varnothing s : \Gamma_1 \to \Gamma_2}{\vdash_\kappa \mathsf{up}\, s : \uparrow\Gamma_1 \to \uparrow\Gamma_2} \qquad \frac{\vdash_\kappa s : \uparrow\Gamma_1 \to \uparrow\Gamma_2}{\vdash_\varnothing \mathsf{down}\, s : \Gamma_1 \to \Gamma_2}$$

Terms

$$\frac{x : A \in \Gamma}{\Gamma \vdash_\Delta x : A} \qquad \frac{\vdash_\Delta s : \Gamma_1 \to \Gamma_2 \quad \Gamma_2 \vdash_\Delta t : A}{\Gamma_1 \vdash_\Delta t[s] : A}$$

$$\frac{\Gamma, x : A \vdash_\Delta t : B}{\Gamma \vdash_\Delta \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash_\Delta f : A \to B \quad \Gamma \vdash_\Delta t : A}{\Gamma \vdash_\Delta f\, t : B}$$

$$\frac{}{\Gamma \vdash_\varnothing \mathsf{tt} : 1} \qquad \frac{\Gamma \vdash_\varnothing t : A}{\Gamma, x : 1 \vdash_\varnothing \mathsf{unitrec}\, t : A}$$

$$\frac{\Gamma \vdash_\Delta t : A \times B}{\Gamma \vdash_\Delta \pi_1 t : A} \qquad \frac{\Gamma \vdash_\Delta t : A \times B}{\Gamma \vdash_\Delta \pi_2 t : B}$$

$$\frac{\Gamma \vdash_\Delta t : A}{\Gamma \vdash_\Delta \mathsf{in}_1 t : A + B} \qquad \frac{\Gamma \vdash_\Delta t : B}{\Gamma \vdash_\Delta \mathsf{in}_2 t : A + B}$$

$$\frac{\Gamma \vdash_\Delta t_1 : A \quad \Gamma \vdash_\Delta t_2 : B}{\Gamma \vdash_\Delta (t_1, t_2) : A \times B} \qquad \frac{\Gamma, x : A \vdash_\Delta t_1 : C \quad \Gamma, y : B \vdash_\Delta t_2 : C}{\Gamma, z : A + B \vdash_\Delta \mathsf{plusrec}\, t_1\, t_2 : C}$$

$$\frac{\uparrow\Gamma \vdash_\kappa : t : A}{\Gamma \vdash_\varnothing \mathsf{box}\, t : \Box A} \qquad \frac{\Gamma \vdash_\varnothing : t : \Box A}{\uparrow\Gamma \vdash_\kappa \mathsf{unbox}\, t : A}$$

$$\frac{\Gamma \vdash_\varnothing t : A}{\uparrow\Gamma \vdash_\kappa \mathsf{up}\, t : \uparrow A} \qquad \frac{\uparrow\Gamma \vdash_\kappa t : \uparrow A}{\Gamma \vdash_\varnothing \mathsf{down}\, t : A}$$

$$\frac{\Gamma \vdash_\kappa t : A}{\Gamma \vdash_\kappa \mathsf{next}\, t : \triangleright A} \qquad \frac{\Gamma \vdash_\kappa f : \triangleright(A \to B) \quad \Gamma \vdash_\kappa t : \triangleright A}{\Gamma \vdash_\kappa f \circledast t : \triangleright B}$$

$$\frac{\Gamma \vdash_\kappa f : \triangleright A \to A}{\Gamma \vdash_\kappa \mathsf{dfix}\, f : \triangleright A} \qquad \frac{\Gamma \vdash_\varnothing t : \Box \triangleright A}{\Gamma \vdash_\varnothing \mathsf{force}\, t : \Box A}$$

$$\frac{\Gamma \vdash_\Delta t : F_P(\mu P)}{\Gamma \vdash_\Delta \mathsf{cons}\, t : \mu P} \qquad \frac{\Gamma \vdash f : F_P(\mu P \times A) \to A}{\Gamma \vdash_\Delta \mathsf{primrec}\, f : \mu P \to A}$$

where F_P is the evaluation of the code P into endofunctors on types, called eval P in Section 3. We omit the presentation of the definitional equalities of terms and substitutions, which can be found in our Agda formalization.

Type Isomorphisms

$$\Box(\uparrow A) \cong A \qquad \Box(A + B) \cong \Box A + \Box B \qquad \uparrow(A \to B) \cong \uparrow A \to \uparrow B \qquad \uparrow(\mu P) \cong \mu(\uparrow P)$$

Context Isomorphisms

$$- \cong \uparrow - \qquad \uparrow\Gamma, \uparrow A \cong \uparrow(\Gamma, A)$$