CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

# A Reflexive Graph Model of Sized Types

Master's thesis in computer science and engineering

Jannis Limperg

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Master's thesis 2019

# A Reflexive Graph Model of Sized Types

Jannis Limperg

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

A Reflexive Graph Model of Sized Types
Jannis Limperg



Supervisor: Andreas Abel, Department of Computer Science
Examiner: John Hughes, Department of Computer Science

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

A Reflexive Graph Model of Sized Types
Jannis Limperg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

# Abstract

Sized types are a type-based termination checking mechanism for dependently typed languages. Compared to syntactic termination checkers, sized types make termination checking more modular and allow for an elegant treatment of coinductive and nested data types.

This thesis investigates λST, a simply-typed lambda calculus with sized types similar to those of Agda. Its primary contribution is a relationally parametric denotational semantics for λST in the form of a reflexive graph model. In this model, sizes are irrelevant: they do not affect the result of any computations. The calculus and model are fully formalised in Agda (without sized types).

# Acknowledgements

I am grateful, first and foremost, to my supervisor Andreas Abel. He suggested this project and supported me from start to finish, even when some of my questions led to lengthy evening coding sessions.

Equally important was Andrea Vezzosi, effectively my co-supervisor. Among other things, he discovered crucial flaws in two early attempts to model sized types and pointed me to the literature on reflexive graph models, which was essential for the project's success.

This document benefited greatly from thorough feedback by Andreas Abel, Fabian Glöckle, Helen Goppelt, John Hughes and Ayberk Tosun.

Many people made my time at Chalmers better. To name but a few: Andreas Abel, Jesper Cockz and Sandro Stucki organised the Initial Types Club, which was more interesting than any official course and whose participants were always up for discussions about more or less crazy ideas. Freðerik Hanghøj Iversen, Nachiappan Valliappan, Pierre Kraft and Siavash Hamedani set up a category theory self-study course, graciously supervised by Thierry Coquand, which turned out to be perhaps the second-most important university course I have taken.

Finally, by far the biggest thanks goes to my family: my parents, sister and grandparents. Their unwavering, unconditional support made this whole journey possible.

Jannis Limperg, Fellbach, December 2019

# Contents

# List of Figures

# 1

# Introduction

Dependent type theories are now well-established as a technology for certifying software and formalising mathematics. Yet they remain rough around some edges, one of which is termination checking. If a type theory is to be consistent as a logic, it must ensure that all recursive programs terminate (and that all corecursive programs are productive). In mainstream dependently typed languages such as Coq, Agda, Idris and Lean, this termination check is implemented by heuristics based on a simple syntactic criterion for termination, the principle of structural recursion.

Despite its simplicity, structural recursion is surprisingly powerful in practice. Still, it suffers from some flaws that are directly related to its syntactic nature. It makes type checking non-compositional: we cannot always replace a term by another term of the same type because some terms have special meaning to the termination checker. It does not play well with more complex data types such as the nested inductive type of rose trees. It does not easily accommodate corecursive definitions, which require a separate productivity check. And when structural checkers are extended to handle more complex scenarios, such as mutual and nested recursion, they tend to become complex and therefore hard to implement correctly.

To address the problems with structural recursion, several other termination checking regimes have been proposed. One of these is *sized types*, an umbrella term for a family of broadly similar type systems that use type annotations to ensure termination. In these systems, inductive types are annotated with a size: Nat $n$, for example, is the type of natural numbers with size $n$. For the moment, we can view $n$ as a natural number and say that Nat $n$ contains only natural numbers $m \leq n$. This interpretation suggests a simple termination criterion: if, in a recursive definition, we receive an input in Nat $n$ and only recurse on terms in Nat $m$ for some $m < n$, then the recursion must stop (at the latest) when it reaches the size zero.

Since this termination check is based entirely on type-level information, it avoids the non-compositionality of syntactic approaches. It also handles nested inductive data types well and, when combined with copatterns [4], can be used to check productivity of corecursive definitions in a similarly natural way. The dependently typed language Agda features an implementation of sized types that demonstrates these advantages. Ch. 2 gives a brief comparison between structural termination checking and Agda's sized types.

Motivated by the desirable properties of sized types, this thesis investigates a custom lambda calculus with sized types called λST, defined in Ch. 3. The calculus extends the simply-typed lambda calculus with sized types that closely resemble Agda's. As such, it provides a setting for "experiments" with Agda's sized types: investigations of the properties of sized types without the surrounding complexity of a full dependently-typed language.

Of these properties, two are of special interest. The first is normalisation: the type system should ensure that all programs do actually terminate. This is the raison d'être of sized types and thus the focus of existing work. The second property we are interested in is size irrelevance: sizes should only be used during type checking to ensure termination/productivity and should not affect the runtime behaviour of programs. This means that sizes can be erased at runtime, which is good for performance.

In Ch. 4, I give a denotational semantics of λST that incorporates a notion of size irrelevance. In the semantics, types are interpreted as reflexive graphs. This technique is usually used to establish parametricity properties, for example of type quantification in System F or Π-types in dependent type theories. It is not surprising that reflexive graphs also yield a model of λST since size irrelevance can be viewed as a parametricity property: it just means that any term which depends on a size is parametric in that size. The model is fully formalised in Agda (without sized types); Ch. 5 discusses the formalisation and some of its technical challenges.

Before I settled on the reflexive graph approach, I also investigated two category-theoretical modelling approaches which looked promising, but turned out to be inadequate. Ch. 6 briefly discusses these models and their problems.

# 2

# Background

## 2.1 Termination Checking

Proof assistants based on dependently typed languages must, if they are to be consistent, ensure that all recursive programs terminate. The usual solution to this problem, used by Coq, Agda, Idris, Lean and others, is to augment the type checker with a separate termination checker based on the principle of structural recursion. That principle says roughly: a program terminates if the input to any recursive call is a subterm of the original input. For example, consider the usual definition of addition for natural numbers:

```
plus : ℕ → ℕ → ℕ
plus zero    m = m
plus (suc n) m = suc (plus n m)
```

In the recursive call in the second equation, `n` is a subterm of the original input, `suc n`. Since this is the only recursive call, the definition as a whole is structurally recursive and is accepted by Agda's termination checker. This is justified because the natural numbers are inductively defined, so any closed term of type `ℕ` consists of finitely many applications of the `suc` constructor. If we "peel off" one constructor for every recursive call, we must eventually reach the `zero` case and the recursion ends.

Structural recursion as a basis for termination checking has many advantages. It is conceptually simple, relatively easy to implement and surprisingly versatile: many interesting definitions are naturally structurally recursive. However, structural termination checkers are also inadequate in some important ways.

First, structural recursion is based on the syntactic notion of a subterm. This makes termination checking non-compositional: we cannot, in general, replace a term with another term of the same type in a recursive definition and still expect the definition to typecheck. For a contrived example, take our previous definition of `plus` and replace the `n` by `id n` in the recursive call. The identity function `id` just returns its argument, so the two terms are obviously equivalent, but a naive termination checker would not accept the modified definition – after all, `id n` is not a syntactic subterm of `suc n`. (Agda's termination checker is smart enough to evaluate `id n` to `n` and therefore accepts the modified definition.)

Unfortunately, this non-compositionality bites not only in such contrived situations. An often-cited example is the mapping function on rose trees (for which we also need the standard list type and its mapping function):

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A
```

```
mapList : ∀ {A B} → (A → B) → List A → List B
mapList f []       = []
mapList f (x ∷ xs) = f x ∷ mapList f xs

data Tree (A : Set) : Set where
  leaf : A → Tree A
  node : List (Tree A) → Tree A

mapTree : ∀ {A B} → (A → B) → Tree A → Tree B
mapTree f (leaf x)  = leaf (f x)
mapTree f (node xs) = node (mapList (mapTree f) xs)
```

Agda's termination checker does not accept `mapTree`, and with good reason: the recursive call in the second equation does not involve a subterm of any input. We can argue that when we unfold `mapList`, it becomes apparent that `mapTree` is always applied to an element of the list `xs` and these elements are obviously subterms of `xs`. But Agda's syntactic check is not smart enough to realise this. (Coq's termination checker employs a heuristic that can deal with `mapTree` by essentially inlining a specialised version of `mapList`.)

A second problem of termination checkers based on structural recursion is that it is tempting to extend them with various heuristics to deal with more complicated variants of structural recursion such as mutually recursive definitions, lexicographic termination measures and nested inductive datatypes like our rose trees. As a result, the termination checkers of Coq and Agda have become quite complex, and with complexity come bugs: both Coq and Agda have shipped versions with errors in their termination checkers that made the systems inconsistent [11, 12].

The final problem with structural recursion is that it does not easily accommodate coinductive datatypes and corecursive definitions. Corecursive functions produce values of coinductive types, which can be infinite. For example, we can coinductively define the type of streams whose values are infinite lists. A function which produces a stream obviously cannot terminate. Instead, we must demand that the function is *productive*, meaning that it generates any finite prefix of its output in finite time. In other words, any finite observation of an infinite stream must terminate. There are syntactic methods, analogous to structural recursion, to ensure productivity, but they are very inflexible. Sized types, on the other hand, when combined with copatterns [4], yield a practical productivity checker mostly for free.

To address these problems of structural termination checkers, we look to sized types as a type-based termination checking mechanism.

## 2.2 Sized Types

Sized types are a termination checking methodology that does not rely on a syntactic analysis. Instead, terms of inductive and coinductive types are annotated at the type level with a size. For terms of inductive types, this size may be thought of as an upper bound on the height of the term, viewed as a constructor tree. For coinductive types, the size denotes the maximum depth to which the term may be inspected. With this setup, checking a recursive

definition becomes easy: a recursive call is justified if the size of its argument decreases. A corecursive call is justified if it increases the maximum observation depth of its result.

Various type systems based on these principles have been proposed (see Sec. 7.1 for references). This thesis investigates a calculus whose sized types closely resemble Agda's, so the remainder of this section gives a brief overview of Agda's system. More details and examples can be found in Agda's manual [5] and in [4].

A size is a term of type `Size< n`, where `n` is also a size. Primitive sizes are variables, the successor of a size `↑ n` and "infinity", `∞`. The size `∞` plays a special role: it designates "fully defined" types, i.e. those whose values could have any size.

For a size `m` to have type `Size< n` it must be less than `n` according to an order `<`. This order is mostly straightforward; for example, we have `n < ↑ n` for all `n`. However, we also have `n < ∞` for all `n` and in particular `∞ < ∞`. As we will see, this rule creates significant problems.

A sized inductive type is simply an inductive type with a parameter of type `Size` (which is the same as `Size< ∞`). Continuing our rose tree example, we define sized lists and a mapping function:

```
data Listₛ (A : Set) (n : Size) : Set where
  []   : Listₛ A n
  cons : (m : Size< n) → A → Listₛ A m → Listₛ A n

mapListₛ : ∀ {A B} n → (A → B) → Listₛ A n → Listₛ B n
mapListₛ n f []           = []
mapListₛ n f (cons m x xs) = cons m (f x) (mapListₛ m f xs)
```

The typing of the `cons` constructor reflects the intuition that the height of `cons m x xs`, `n`, is strictly greater than the height of `xs`, `m`.[1] We can then exploit this in `mapListₛ`: in the recursive call, `mapListₛ` is applied to `m`, which we know from the type of `cons` is less than `n`. Thus, each recursive call decreases in size and the recursion is justified. The definition also demonstrates that the order on sizes induces a subtyping relation: the right-hand side of the second equation has type `Listₛ A m`, which is a subtype of `Listₛ A n` for `m < n`.

So far, we could have just as well used a structural termination checker. The benefits of sized types become obvious when we turn to rose trees:

```
data Treeₛ (A : Set) (n : Size) : Set where
  leaf : A → Treeₛ A n
  node : (m : Size< n) → Listₛ (Treeₛ A m) ∞ → Treeₛ A n

mapTreeₛ : ∀ {A B} n → (A → B) → Treeₛ A n → Treeₛ B n
mapTreeₛ n f (leaf x)    = leaf (f x)
mapTreeₛ n f (node m xs) = node m (mapListₛ ∞ (mapTreeₛ m f) xs)
```

The definition of `Treeₛ` demonstrates the utility of `∞`: we can say that a `node` should have a list of children without caring about the size of that list. But more importantly, the termination checker now accepts `mapTreeₛ`: the recursive call is now at size `m < n` and

[1] Agda also supports a different style of using sized types where `cons` takes a `Listₛ A n` as input and returns a `Listₛ A (↑ n)` (making `n` an index rather than a parameter of `Listₛ`). This works just as well for the most part, but there are technical reasons to prefer our "quantifier style".

thus justified. In effect, we have encoded in the type of `node` our intuition that the height of `node xs` is strictly greater than the height of any of the elements of `xs`.

Agda's sized types thus deliver on the compositionality promise: all information the termination checker needs is encoded at the type level, so we can freely abstract over terms. Sized types are also mostly straightforward to implement, being just an extension of the type system.[2]

Unfortunately, such convenience currently comes at the ultimate price: Agda's implementation of sized types is, and has been for some time, inconsistent. The culprit seems to be the highly dubious rule `n < ∞`, in particular `∞ < ∞`. This rule makes the `<` relation obviously non-well-founded (meaning there is an infinite descending chain `∞ < ∞ < …`) but Agda assumes that `<` is well-founded. This assumption can be exploited in different ways [1, 2, 3, 28] to sneak non-terminating programs past the termination checker.

It is currently unclear how to satisfactorily resolve this issue with Agda's design. In this thesis, I adopt the obvious solution: changing the `<` relation so that `∞ ≮ ∞`. Doing the same in Agda would, however, lead to issues with the constructors and fields of sized data types. For example, we would like to use the `cons` constructor for sized lists at type

```
A → Listₛ A ∞ → Listₛ A ∞
```

but this is impossible with `∞ ≮ ∞`. As a workaround, we can define a consing operation for lists at `∞` that is extensionally equal to `cons`, but this requires a pattern match on the input list – an inefficiency that should not be necessary. How best to "rescue" Agda's sized types thus remains an open question for now.

Another possible criticism of Agda's sized types concerns expressivity. Agda's size arithmetic is restricted to the successor; we cannot add or multiply sizes. This means that we cannot give a precise type to, for example, the list appending function, whose output size should be the sum of its input sizes. This lack of expressivity, however, is a conscious design decision. In return, we get a size inference algorithm that can infer almost all sizes in a typical program. This significantly lowers the cost of adopting sized types.

[2] I say 'mostly' because Agda's current implementation includes some subtle checks to prevent inconsistent size assumptions, which is necessary to preserve decidability of type checking. Agda also has a sophisticated size inference engine (so we could have left all sizes in our example code implicit), but this engine is not part of the trusted computing base.

# 3

# Object Language

This chapter introduces λST, the calculus that will be the object of study for the remainder of this thesis.

## 3.1 Overview

Before we dive into the technicalities of λST, I give an overview of the calculus by example. Those who prefer their exposition top-down may skip to Sec. 3.2 and return later.

λST features two base types: sized natural numbers and sized streams of natural numbers. Our first example is the function `half` which divides a natural number by two (rounding down if the input is odd). In non-sized Agda, this function would be written as follows:

```
half : ℕ → ℕ
half zero = zero
half (suc zero) = zero
half (suc (suc n)) = suc (half n)
```

The λST version is broadly similar, only with added size information and more cumbersome syntax:

```
half : ∀ n < ↑ ∞. Nat n → Nat n
half ≔ fix[Nat • → Nat •]
         (Λ n < ↑ ∞. λ rec : (∀ m < n. Nat m → Nat m). λ i : Nat n.
           case i of
             (zero n)  → zero n
             (suc m j) →
               case j of
                 (zero m)  → zero n
                 (suc o k) → suc n o (rec o k)
```

The definition of `half` already showcases most features of λST. The type of natural numbers of size `n` is called `Nat n`. If we think of sizes as ordinals, `Nat n` is the type of naturals less than or equal to `n`. This intuition justifies the types of its constructors:

```
zero : ∀ n < ↑ ∞. Nat n
suc  : ∀ n < ↑ ∞. ∀ m < n. Nat m → Nat n
```

Zero is less than or equal to any `n`, so it is always in `Nat n`. The successor of a natural number `k` of size `m` is less than or equal to `n` if `m < n`. If we were to define sized natural

numbers in Agda, we would get constructors with very similar types. To eliminate sized natural numbers, we use Haskell-style pattern matching. (The formal definition of λST in Ch. 3 uses an eliminator instead.) The meaning of `↑ ∞` is explained below.

λST's size grammar is similar to that of Agda, but for now we only need size variables. These are introduced at the type level by a size quantification of the form `∀ x < m`, where `x` is a size variable and `m`, a size, is the upper bound of `x` (so `x` can only be instantiated with sizes below `m`). At the term level, size variables are introduced by binders of the form `Λ x < m` and terms with a size-quantified type can be applied to sizes below the declared bound. All this makes size quantification resemble the type quantification of System F: the bound of a size variable corresponds to the kind of a type variable, size binders correspond to type binders and size application corresponds to type application.

As mentioned in Sec. 2.2, λST departs from Agda in its handling of the "infinite" size ∞. In Agda, ∞ is less than itself, so the type `Size< ∞` contains ∞. This is convenient, but also a likely culprit of size-related inconsistencies. In λST, we therefore omit the rule `∞ < ∞`. Where we want to quantify over all sizes up to and including ∞, as in the type of `half`, we use the successor of ∞, `↑ ∞`, as a bound.

The last ingredient we need to define `half` is the centrepiece of λST: the size-based fixpoint operator `fix` of type

```
fix[T •] : (∀ n < ↑ ∞. (∀ m < n. T m) → T n) → ∀ n < ↑ ∞. T n
```

Here, `T` is an arbitrary type with one free size variable denoted by the 'hole' •. The type of `fix` corresponds to induction over sizes: if from an element of `T` with size `m` we can construct an element of `T` with a greater size `n`, then we can construct an element of `T` with any size.

Besides natural numbers, the prototypical inductive type, λST also features the prototypical coinductive type: sized streams (of natural numbers). They have one constructor, `cons`, which prepends an element to a stream, and two destructors, `head` and `tail`, which respectively extract the first element and the rest of a stream. These have the following types, again analogous to those one would obtain in Agda:

```
cons : ∀ n < ↑ ∞. Nat ∞ → (∀ m < n. Stream m) → Stream n
head : ∀ n < ↑ ∞. Stream n → Nat ∞
tail : ∀ n < ↑ ∞. Stream n → ∀ m < n. Stream m
```

Using these primitives, we can define, for example, the stream of zeros:

```
zeros : ∀ n < ↑ ∞. Stream n
zeros ≔ fix[Stream •]
          (Λ n < ↑ ∞. λ rec : (∀ m < n. Stream m).
            cons (zero ∞) n rec)
```

More interestingly, λST also admits the standard mapping function for streams:

```
map : (Nat ∞ → Nat ∞) → ∀ n < ↑ ∞. Stream n → Stream n
map ≔ λ f : Nat ∞ → Nat ∞. fix[Stream • → Stream •]
      (Λ n < ↑ ∞. λ rec : (Λ m < n. Stream m → Stream m). λ xs : Stream n.
          cons (f (head n xs)) n (Λ m < n. rec m (tail n xs m)))
```

| | | | | |
|---|---|---|---|---|
| size | $n, m, o$ | $::=$ | $v_x$ | (size variable) |
| | | $\mid$ | $0$ | (zero) |
| | | $\mid$ | $\uparrow n$ | (successor) |
| | | $\mid$ | $\infty$ | (infinity) |
| size context | $\Delta, \Omega$ | $::=$ | $()$ | (empty context) |
| | | $\mid$ | $\Delta, n$ | (context extension) |

**Figure 3.1:** Syntax of sizes, size variables and size contexts

Note that according to its type, `map` is size-preserving: if we apply it to a stream with at least `n` defined elements, its result also has at least `n` defined elements. The same applies to `half`, whose type indicates that its output is no greater than its input. As we saw with the rose tree example in Sec. 2.2, such preservation properties are very useful.

## 3.2 Sizes

In the following sections we define λST. This section corresponds to the Agda module Source.Size in the formalisation [16].[3]

*Sizes* and *size contexts* are defined by the grammar in Fig. 3.1. Size variables are de Bruijn indices; we write $v_x$ for the $x$th variable. (The examples in the previous sections instead used named variables for convenience.)

Our size grammar is essentially that of Agda (although Agda does not have a separate grammatical category for sizes). We only add a zero size, which could easily be added to Agda as well.[4] Contexts are telescopes of size bounds: in the context $\Delta, n$, the zeroth variable $v_0$ ranges over sizes less than $n$ and $n$ may involve variables in $\Delta$.

The typing judgments for sizes are mutually inductively generated by the rules in Fig. 3.2. The judgment $\Delta \vdash_x v_x < n$ holds if $v_x$ is a valid index into $\Delta$ and $n$ is the bound of $v_x$. The other two judgments, $\Delta \vdash n$ and $\vdash \Delta$, ensure that sizes and contexts are scope-safe. In the definition of $\Delta \vdash_x v_x < n$, we use a weakened size $\mathrm{wk}(n)$, which is $n$ with all variables $v_x$ replaced by $v_{x+1}$.

For convenience, the judgments are defined such that $\Delta \vdash_x v_x < n$ implies $\Delta \vdash n$ and $\Delta \vdash n$ implies $\vdash \Delta$, as is easily proved by induction on the respective derivation. All other judgments we define below are structured similarly, so for example $\Delta \vdash T$ (scope-safety of the type $T$) implies $\vdash \Delta$ and $\Delta; \Gamma \vdash t : T$ (well-typedness of a term $t$ of type $T$ in context $\Gamma$ and size context $\Delta$) implies $\Delta \vdash T$. I will henceforth use these lemmas implicitly.

Next, we define a syntactic comparison relation $\Delta \vdash n < m$ between sizes in the same context, which is inductively generated by the rules in Fig. 3.3. The rules of $<$ are all

[3] In electronic versions of this document, module names such as Source.Size link to a HTML rendering of the respective module.

[4] We need the zero size because we omit Agda's `∞ < ∞` rule. Without this rule, certain Agda constructions, such as a natural number successor that works on `Nat ∞`, do not transfer directly to λST. Adding the zero size allows us to approximate these constructions in λST.

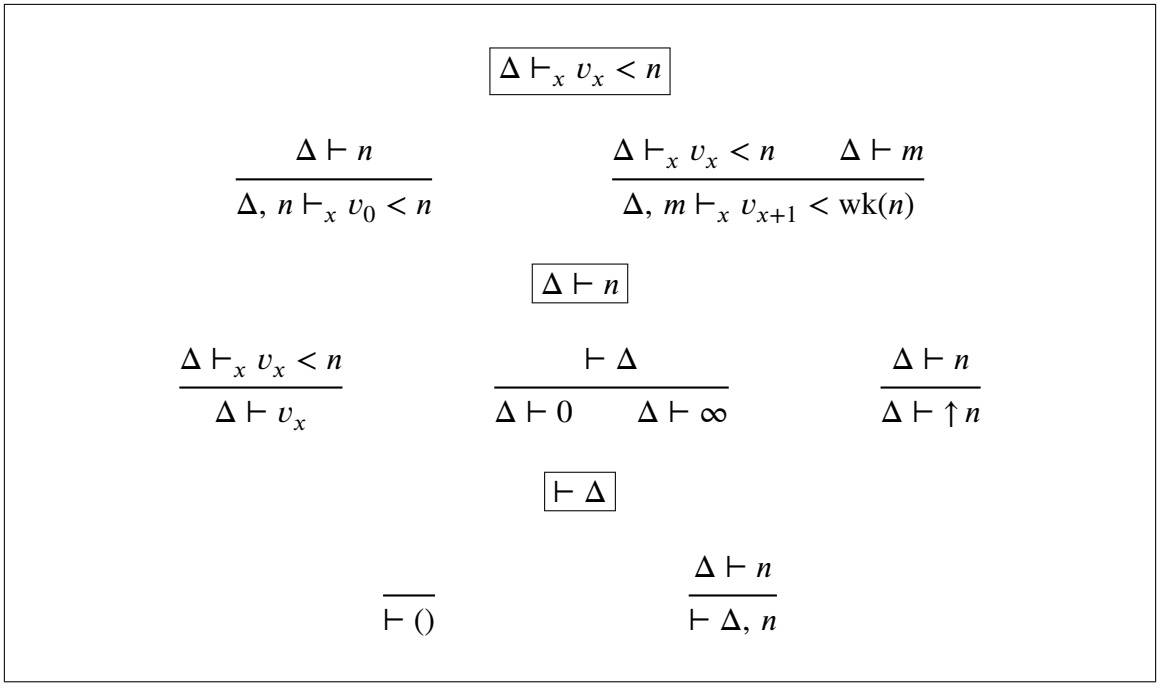


**Figure 3.2:** Typing rules for sizes and size contexts

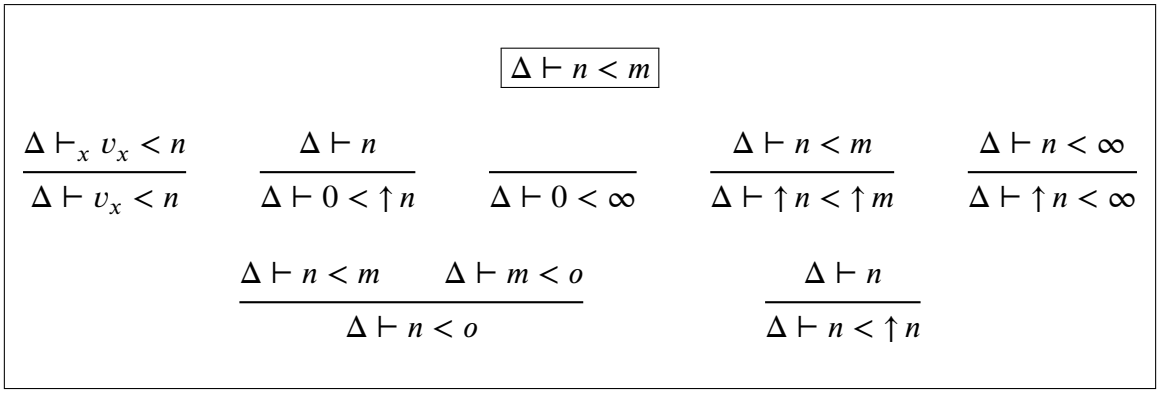


**Figure 3.3:** Size comparison

supported by the intuition that sizes of the form $\uparrow \ldots \uparrow 0$ are natural numbers and sizes of the form $\uparrow \ldots \uparrow \infty$ are ordinals $\omega + i$. Indeed, this will be the interpretation of sizes in the model. As mentioned, we do not admit Agda's dubious rule $\infty < \infty$ (see Sec. 2.2).

The following technical lemma will be needed later:

**Lemma 3.1** (Weakening preserves typing and $<$)**.** *If* $\Delta \vdash n$ *and* $\Delta \vdash m$*, then* $\Delta, m \vdash \mathrm{wk}(n)$*. If* $\Delta \vdash o$ *and* $\Delta \vdash n < m$*, then* $\Delta, o \vdash \mathrm{wk}(n) < \mathrm{wk}(m)$*.*

I do not give a proof of this lemma, and indeed most of the following lemmas, because both statements are proved by an almost trivial induction. Instead, the reader may refer to the formalisation, which closely follows the structure of this thesis and provides all the details one could ever wish for.

$$\boxed{\sigma : \Delta \Rightarrow \Omega}$$

$$\frac{\vdash \Delta}{() : \Delta \Rightarrow ()}$$

$$\frac{\sigma : \Delta \Rightarrow \Omega \qquad \Delta \vdash n \qquad \Omega \vdash m \qquad \Delta \vdash n < m[\sigma]}{\sigma, n : \Delta \Rightarrow \Omega,\ m}$$

**Figure 3.4:** Typing rules for size substitutions

## 3.3 Size Substitutions

Having size variables, we must say how to substitute for them. λST uses implicit substitutions, so substitutions are not part of the calculus' grammar. We do, however, define a universe of simultaneous substitutions between given size contexts – this will be helpful when we get to the model, where substitutions will correspond to morphisms between (interpretations of) size contexts. Size substitutions and their properties are formalised in Source.Size.Substitution.Canonical.

A *size substitution* is a snoc-list of sizes. We write () for the empty substitution and $\sigma, n$ for the extension of $\sigma$ with a size $n$. The application of a substitution to a size, $n[\sigma]$, is defined by recursion on $n$:

$$\begin{aligned}
v_0[\sigma, n] &:= n \\
v_{x+1}[\sigma, n] &:= v_x[\sigma] \\
0[\sigma] &:= 0 \\
(\uparrow n)[\sigma] &:= \uparrow n[\sigma] \\
\infty[\sigma] &:= \infty.
\end{aligned}$$

Fig. 3.4 gives the typing rules for size substitutions. In the previous definition, there is no equation for the case $v_x[()]$ because this is impossible if $n$ and $\sigma$ are well-typed.

The typing rules ensure that if we have $\sigma : \Delta \Rightarrow \Omega$, then $\sigma$ maps every variable $v_x$ in $\Omega$ to a size $n$ such that $n$ contains only variables in $\Delta$ and $n$ respects the bound declared for $v_x$ in $\Omega$. Due to these properties, well-typed substitutions are well-behaved with regards to typing and size comparison:

**Lemma 3.2** (Size substitutions preserve typing and $<$)**.** *If* $\sigma : \Delta \Rightarrow \Omega$ *and* $\Omega \vdash n$*, then* $\Delta \vdash n[\sigma]$*. If* $\sigma : \Delta \Rightarrow \Omega$ *and* $\Omega \vdash n < m$*, then* $\Delta \vdash n[\sigma] < m[\sigma]$*.*

*Proof.* By induction on the derivation of $\Omega \vdash n$. In the proof of $\Delta \vdash n[\sigma] < m[\sigma]$ we use the lemma $\mathrm{wk}(m)[\sigma, n] = m[\sigma]$, which is proved by a simple induction. □

Note that we write $\sigma : \Delta \Rightarrow \Omega$ even though substitution with $\sigma$ transforms an object in $\Omega$ into an object in $\Delta$. This notation is inspired by the model, where $\Delta$ and $\Omega$ will be particular types and $\sigma$ a function between them.

Having constructed a universe of substitutions, we now define some particular substitutions in this universe which we will use later. We could also have axiomatised these substitutions directly but that would have made the substitution operation and any proofs about it more complex, and we would also have had to axiomatise some equations between substitutions instead of proving them as a lemma. Our design leads to some subtle problems in the formalisation; see Sec. 5.2 for details and a workaround.

We define the following well-typed substitutions (assuming well-typed sizes $\Omega \vdash n$ and $\Delta \vdash m$ as well as substitutions $\sigma : \Delta \Rightarrow \Omega$, $\tau : \Omega \Rightarrow \Omega'$):

- The *weakening* of $\sigma$ is

  $$\begin{aligned}&\mathrm{Weaken}(\sigma) : \Delta, m \Rightarrow \Omega\\ &\mathrm{Weaken}(()) \;\coloneqq ()\\ &\mathrm{Weaken}(\sigma, n) \coloneqq \mathrm{Weaken}(\sigma), \mathrm{wk}(n).\end{aligned}$$

- The *lifting* of $\sigma$ is

  $$\begin{aligned}&\mathrm{Lift}(\sigma) : \Delta, n[\sigma] \Rightarrow \Omega, n\\ &\mathrm{Lift}(\sigma) \coloneqq \mathrm{Weaken}(\sigma), v_0.\end{aligned}$$

- The *identity substitution* $\mathrm{Id}_\Delta$ is

  $$\begin{aligned}&\mathrm{Id}_\Delta : \Delta \Rightarrow \Delta\\ &\mathrm{Id}_{()} \;\coloneqq ()\\ &\mathrm{Id}_{\Delta, n} \coloneqq \mathrm{Lift}(\mathrm{Id}_\Delta).\end{aligned}$$

- The *forward composition* of $\sigma$ and $\tau$ is

  $$\begin{aligned}&\sigma \gg \tau : \Delta \Rightarrow \Omega'\\ &\sigma \gg () \;\;\coloneqq ()\\ &\sigma \gg (\tau, n) \coloneqq (\sigma \gg \tau), n[\sigma].\end{aligned}$$

- The *weakening substitution* is

  $$\begin{aligned}&\mathrm{Wk} : \Delta, m \Rightarrow \Delta\\ &\mathrm{Wk} \coloneqq \mathrm{Weaken}(\mathrm{Id}).\end{aligned}$$

- Assuming that $\Delta \vdash n < m$, the *singleton substitution* of $n$ is

  $$\begin{aligned}&\mathrm{Sing}(n) : \Delta \Rightarrow \Delta, m\\ &\mathrm{Sing}(n) \coloneqq \mathrm{Id}_\Delta, n.\end{aligned}$$

- The *skipping substitution* is

  $$\begin{aligned}&\mathrm{Skip} : \Delta, m, v_0 \Rightarrow \Delta, m\\ &\mathrm{Skip} \coloneqq \mathrm{Weaken}(\mathrm{Wk}), v_0.\end{aligned}$$

The first six of these substitutions are standard. The skipping substitution is perhaps easier to understand if we switch to explicit variables for a moment: given a size $n$ in the context $\Delta$, $x < m$, $n\big[\mathrm{Skip}\big]$ lives in the context $\Delta$, $x < m$, $y < x$ and all occurrences of $x$ in $n$ are replaced by $y$. In this sense, $n\big[\mathrm{Skip}\big]$ "skips" $x$.

$$
\begin{array}{llll}
\text{type} & T, U, V & ::= & \text{Nat } n \quad \text{(sized naturals)} \\
& & \mid & \text{Stream } n \quad \text{(sized streams)} \\
& & \mid & T \to U \quad \text{(function space)} \\
& & \mid & \forall n.\ T \quad \text{(size quantification)} \\
\text{context} & \Gamma, \Psi & ::= & () \quad \text{(empty context)} \\
& & \mid & \Gamma,\ T \quad \text{(context extension)}
\end{array}
$$

**Figure 3.5:** Syntax of types and contexts

$$\boxed{\Delta \vdash T}$$

$$\frac{\Delta \vdash n}{\Delta \vdash \text{Nat } n \qquad \Delta \vdash \text{Stream } n} \qquad \frac{\Delta \vdash T \qquad \Delta \vdash U}{\Delta \vdash T \to U} \qquad \frac{\Delta,\ n \vdash T}{\Delta \vdash \forall n.\ T}$$

$$\boxed{\Delta \vdash \Gamma}$$

$$\frac{\vdash \Delta}{\Delta \vdash ()} \qquad \frac{\Delta \vdash \Gamma \qquad \Delta \vdash T}{\Delta \vdash \Gamma,\ T}$$

**Figure 3.6:** Typing rules for types and contexts

## 3.4 Types

We move on to the type language of λST. This section is formalised in Source.Type.

The types and contexts of λST are generated by the grammar in Fig. 3.5. Their typing rules appear in Fig. 3.6.

λST features two base types: Nat $n$, the type of sized natural numbers, and Stream $n$, the type of sized streams. We will discuss these in more detail when we get to the terms of λST. As for composite types, λST has a non-dependent function space $T \to U$ and size quantification $\forall n.\ T$. Note that $n$ is not a variable, as the notation may suggest, but the bound of the zeroth de Bruijn index in $T$. Size quantification works essentially like quantification in System F if we think of the bound $n$ as the "type" of a size variable. Contexts are lists of types. The typing rules for types and contexts ensure scope-safety relative to some size context.

Since types can contain size variables, we define *size substitution in types*, written

| | | | |
|---|---|---|---|
| term | $t, u, v$ | $::=$ | $v_x$ (variable) |
| | | $\mid$ | $\lambda T.\ t$ (abstraction) |
| | | $\mid$ | $t\ u$ (application) |
| | | $\mid$ | $\lambda n.\ t$ (size abstraction) |
| | | $\mid$ | $t\ n$ (size application) |
| | | $\mid$ | zero $n$ (zero) |
| | | $\mid$ | suc $n\ m\ t$ (successor) |
| | | $\mid$ | caseNat$[T]\ n\ t\ u\ v$ (case analysis for Nat $n$) |
| | | $\mid$ | cons $t\ n\ u$ (stream construction) |
| | | $\mid$ | head $n\ t$ (stream head) |
| | | $\mid$ | tail $n\ t\ m$ (stream tail) |
| | | $\mid$ | fix$[T]\ t\ n$ (sized fixpoint) |

**Figure 3.7:** Syntax of terms

$T[\sigma]$, by recursion over the type:

$$
\begin{aligned}
(\text{Nat } n)[\sigma] &:= \text{Nat } n[\sigma] \\
(\text{Stream } n)[\sigma] &:= \text{Stream } n[\sigma] \\
(T \to U)[\sigma] &:= T[\sigma] \to U[\sigma] \\
(\forall n.\ T)[\sigma] &:= \forall n[\sigma].\ T[\text{Lift}(\sigma)].
\end{aligned}
$$

Substitution in contexts $\Gamma[\sigma]$ is the pointwise extension of substitution in types.

The only interesting case of the above definition is that for size quantification. Here we must recurse with $\text{Lift}(\sigma)$ rather than $\sigma$ because $T$ is a type in $\Delta$, $n$ while $\forall n.\ T$ is a type in $\Delta$. As usual, size substitution preserves scope safety: if $\Omega \vdash T$ and $\sigma : \Delta \Rightarrow \Omega$, then $\Delta \vdash T[\sigma]$ (and similar for contexts).

## 3.5 Terms

In this section, we define the terms of λST and their typing rules. The corresponding module in the formalisation is Source.Term.

The terms of λST are generated by the grammar in Fig. 3.7. Term variables, like size variables, are de Bruijn indices.

The first three term constructions are those of the simply-typed lambda calculus: variables, abstraction and application. The next two are size abstraction and size application, which respectively introduce and eliminate a size quantification. They are analogous to System F's type abstraction and type application.

The other terms are conceptually constants, though we introduce them as term formers to simplify the model (and particularly the formalisation). If they were constants, they

would have the following types (with explicit variables for convenience):

$$
\begin{array}{ll}
\text{zero} & : \forall x < \uparrow \infty.\ \text{Nat}\ x \\
\text{suc} & : \forall x < \uparrow \infty.\ \forall y < x.\ \text{Nat}\ y \to \text{Nat}\ x \\
\text{caseNat}[T] & : \forall x < \uparrow \infty.\ \text{Nat}\ x \to T \to (\forall y < x.\ T) \to T \\
\text{cons} & : \text{Nat}\ \infty \to \forall x < \uparrow \infty.\ (\forall y < x.\ \text{Stream}\ y) \to \text{Stream}\ x \\
\text{head} & : \forall x < \uparrow \infty.\ \text{Stream}\ x \to \text{Nat}\ \infty \\
\text{tail} & : \forall x < \uparrow \infty.\ \text{Stream}\ x \to \forall y < x.\ \text{Stream}\ y \\
\text{fix}[T(\bullet)] & : (\forall x < \uparrow \infty.\ (\forall y < x.\ T(y)) \to T(x)) \to \forall x < \uparrow \infty.\ T(x).
\end{array}
$$

The terms zero and suc serve as constructors of Nat $n$. The type Nat $n$ may be thought of as the set of natural numbers less than or equal to $n$. 0 is therefore a member of Nat $n$ for any $n$, as the type of zero indicates. Similarly, if we have a natural number $\leq n$, its successor is $\leq m$ for any $m > n$, which justifies the type of suc. The eliminator for Nat $n$ into some arbitrary type $T$ is caseNat[$T$]. It performs case analysis on its first non-size argument: in the zero case, it returns its second argument; in the successor case, it returns the result of applying the function given as the third argument to the predecessor of the first argument. That predecessor, of course, has some smaller size $m < n$.

For Stream $n$, the size $n$ has the opposite meaning: where Nat $n$ contains naturals up to *at most* $n$, Stream $n$ is roughly speaking the type of lists of naturals with *at least* $n$ elements. This is because Nat $n$ is an inductive type while Stream $n$ is coinductive. More precisely, a value of Stream $n$ is a function which maps any natural number $m \leq n$ to a natural number. Stream $\infty$ is then the type of functions $\mathbb{N} \to \mathbb{N}$, which is isomorphic to the coinductively defined type of infinite streams of naturals.

The constructor, cons, and the destructors, head and tail, reflect this understanding. The first destructor head extracts the element at index 0 from a Stream $n$ (which always exists since $0 \leq n$). The second destructor tail, when given a Stream $n$, returns its tail, which has one less element and is therefore an element of Stream $m$ for any $m < n$. The constructor cons takes a function which generates a Stream $m$ for any $m < n$ and prepends an element, returning a bigger Stream $n$.[5]

The final term, fix[$T$], is a size-based fixpoint combinator that allows us both to recurse over sized natural numbers and to build sized streams corecursively. It is parameterised by a type $T$ with one free size variable or "hole", as the notation $T(\bullet)$ suggests. The type of fix[$T$] is essentially the principle of induction over sizes: if from an object of size $m$ we can construct an object of size $n > m$, then we can construct an object of any size.

The typing rules of λST terms, which appear in Fig. 3.8, follow directly from the previous discussion, only with constants replaced by equivalent term formers and explicit variables replaced by de Bruijn indices and size substitutions.

[5] Another plausible type for cons would be $\text{Nat}\ \infty \to \forall x < \uparrow \infty.\ \forall y < x.\ \text{Stream}\ y \to \text{Stream}\ x$, where the stream argument is a single stream instead of a family. I choose not to use this simpler type because if we define sized streams in Agda, cons has the type given above.

$$\boxed{\Delta;\Gamma \vdash t : T}$$

$$\frac{\Delta \vdash \Gamma \qquad \Delta \vdash T}{\Delta;\Gamma, T \vdash v_0 : T} \qquad \frac{\Delta;\Gamma \vdash v_x : T \qquad \Delta \vdash U}{\Delta;\Gamma, U \vdash v_{x+1} : T} \qquad \frac{\Delta;\Gamma, T \vdash t : U}{\Delta;\Gamma \vdash \lambda T.\, t : T \to U}$$

$$\frac{\Delta;\Gamma \vdash t : T \to U \qquad \Delta;\Gamma \vdash u : T}{\Delta;\Gamma \vdash t\ u : U} \qquad \frac{\Delta, n;\Gamma[\mathrm{Wk}] \vdash t : T \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash \lambda n.\, t : \forall n.\, T}$$

$$\frac{\Delta;\Gamma \vdash t : \forall n.\, T \qquad \Delta \vdash m < n}{\Delta;\Gamma \vdash t\ m : T\left[\mathrm{Sing}(m)\right]} \qquad \frac{\Delta \vdash n < \uparrow \infty \qquad \Delta \vdash \Gamma}{\Delta;\Gamma \vdash \mathrm{zero}\ n : \mathrm{Nat}\ n}$$

$$\frac{\Delta \vdash n < \uparrow \infty \qquad \Delta \vdash m < n \qquad \Delta;\Gamma \vdash i : \mathrm{Nat}\ m}{\Delta;\Gamma \vdash \mathrm{suc}\ n\ m\ i : \mathrm{Nat}\ n}$$

$$\frac{\begin{array}{c}\Delta \vdash T \qquad \Delta \vdash n < \uparrow \infty \\ \Delta;\Gamma \vdash i : \mathrm{Nat}\ n \qquad \Delta;\Gamma \vdash z : T \qquad \Delta;\Gamma \vdash s : \forall n.\, \mathrm{Nat}\ v_0 \to T[\mathrm{Wk}]\end{array}}{\Delta;\Gamma \vdash \mathrm{caseNat}[T]\ n\ i\ z\ s : T}$$

$$\frac{\Delta \vdash n < \uparrow \infty \qquad \Delta;\Gamma \vdash i : \mathrm{Nat}\ \infty \qquad \Delta;\Gamma \vdash is : \forall n.\, \mathrm{Stream}\ v_0}{\Delta;\Gamma \vdash \mathrm{cons}\ n\ i\ is : \mathrm{Stream}\ n}$$

$$\frac{\Delta \vdash n < \uparrow \infty \qquad \Delta;\Gamma \vdash is : \mathrm{Stream}\ n}{\Delta;\Gamma \vdash \mathrm{head}\ n\ is : \mathrm{Nat}\ \infty}$$

$$\frac{\Delta \vdash n < \uparrow \infty \qquad \Delta;\Gamma \vdash is : \mathrm{Stream}\ n \qquad \Delta \vdash m < n}{\Delta;\Gamma \vdash \mathrm{tail}\ n\ is\ m : \mathrm{Stream}\ m}$$

$$\frac{\Delta, \uparrow \infty \vdash T \qquad \Delta;\Gamma \vdash t : \forall \uparrow \infty.\, (\forall v_0.\, T\left[\mathrm{Skip}\right]) \to T \qquad \Delta \vdash n < \uparrow \infty}{\Delta;\Gamma \vdash \mathrm{fix}[T]\ t\ n : T\left[\mathrm{Sing}(n)\right]}$$

**Figure 3.8:** Typing rules for terms

# 4

# Reflexive Graph Model

I now present a reflexive graph model of λST. The model is very similar to the standard set-theoretic models of (dependent) type theories, but it additionally captures a parametricity property: when we interpret a term $f : \forall n.\, T$, we get a dependent function

$$[\![f]\!] : (m : \mathrm{Size}_{<[\![n]\!]}) \to [\![T]\!](m)$$

such that $[\![f]\!](m)$ is independent, in an appropriate sense, of $m$. In other words, $[\![f]\!]$ is size-parametric.

To develop this model, I first introduce its central structure, (propositional) reflexive graphs and their families. I then show how to interpret each construct of λST – sizes, types and terms – in turn.

## 4.1 Propositional Reflexive Graphs

This section introduces the category of reflexive graphs, which will be used to interpret sizes and size contexts. The following results are formalised in Model.RGraph.

A *reflexive graph* is a tuple $(\Delta, \approx_\Delta, \mathrm{refl}_\Delta)$ where

$$\begin{array}{lll} \Delta & : \mathrm{Type} & \\ \approx_\Delta & : \Delta \to \Delta \to \mathrm{Type} & \\ \mathrm{refl}_\Delta(\delta) & : \delta \approx_\Delta \delta & \forall \delta : \Delta. \end{array}$$

I will generally identify a reflexive graph with its underlying type.

The relation $\approx_\Delta$ is what distinguishes this model from the standard set-theoretical model of type theory. As we interpret the constructions of λST in the next sections, we will define $\approx_\Delta$ to be "equality up to sizes". For example, the interpretation $[\![\Delta]\!]$ of a size context $\Delta$ will have $\delta \approx_{[\![\Delta]\!]} \delta'$ for all $\delta$ and $\delta'$ because any two interpretations of a size context should be considered equal up to sizes. This means that we can use interpretations of sizes internally – particularly to construct a terminating interpretation of fixpoints – but only in such a way that they do not affect the overall result of a computation.

*Remark.* Each reflexive graph is isomorphic to a presheaf over a particular category. Thus, reflexive graphs form a category with families [10], i.e. a model of Martin-Löf type theory. It is thus not surprising that λST also admits a reflexive graph model. Moreover, many of the following results are direct consequences of the theory of presheaves, but to keep the presentation elementary, I make the constructions explicit.

A reflexive graph $\Delta$ is *propositional* if $\approx_\Delta$ is a proposition in the sense of Homotopy Type Theory [27] (i.e. for any $\delta, \delta' : \Delta$ and $p, q : \delta \approx_\Delta \delta'$, $p = q$) and the underlying type $\Delta$ is a set (i.e. equality on $\Delta$ is a proposition). We call such a graph a *PRGraph*. These conditions are trivially fulfilled for all the PRGraphs we consider, so I will silently omit their proofs.

PRGraphs are simpler than non-propositional reflexive graphs since the proofs $\mathrm{refl}_\Delta(\delta)$ are irrelevant. This saves us a lot of effort, particularly in the formalisation. However, it means that the model probably would not accommodate theories where the identity type is non-propositional, such as cubical type theories.

Homomorphisms of PRGraphs are relation-preserving functions. More explicitly, a *PRGraph morphism* between PRGraphs $\Delta$ and $\Omega$ is a function $\sigma$ between their underlying types such that

$$\delta \approx_\Delta \delta' \Rightarrow \sigma(\delta) \approx_\Omega \sigma(\delta') \quad \forall \delta, \delta' : \Delta.$$

Given a proof $p : \delta \approx_\Delta \delta'$, I will occasionally write $\sigma(p) : \sigma(\delta) \approx_\Omega \sigma(\delta')$. Not very often, though, since by propositionality of $\Omega$ it does not matter which specific proof of $\sigma(\delta) \approx_\Omega \sigma(\delta')$ we consider. This also means that in particular $\sigma(\mathrm{refl}_\Delta(\delta)) = \mathrm{refl}_\Omega(\sigma(\delta))$; for non-propositional reflexive graphs, we would have to add this condition to the definition of morphisms.

PRGraphs and their morphisms obviously form a category, which we call PRGraphs.

## 4.2 Families of Propositional Reflexive Graphs

To interpret types and type contexts, which are indexed by size contexts, we need to move from PRGraphs to families of PRGraphs. These may be thought of as PRGraphs indexed by PRGraphs. The results in this section are formalised in Model.Type.Core.

A *family of PRGraphs* over a given PRGraph $\Delta$ is a tuple $(T, \approx_{T,p}, \mathrm{refl}_T)$ where

$$\begin{array}{lll} T & : \Delta \to \mathrm{Type} & \\ \approx_{T,p} & : T(\delta) \to T(\delta') \to \mathrm{Type} & \forall p : \delta \approx_\Delta \delta' \\ \mathrm{refl}_T(x) & : x \approx_{T,\mathrm{refl}_\Delta(\delta)} x & \forall x : T(\delta). \end{array}$$

such that $T(\delta)$ is a set for any $\delta$ and $\approx_{T,p}$ is a proposition for any $T$ and $p$.

Note that our "equality up to sizes" is now heterogeneous: we can ask whether an inhabitant of $T(\delta)$ is equal to an inhabitant of $T(\delta')$, as long as $\delta$ and $\delta'$ are equal up to sizes. In the model, $\delta$ and $\delta'$ will be (interpretations of) size valuations, which are always equal up to sizes. We will thus be able to compare, for example, (interpretations of) terms in Nat $n$ and Nat $m$ for arbitrary $n$ and $m$. This is necessary as well: if λST is indeed size-irrelevant and we take a term $t : \forall n.\ \mathrm{Nat}\ \upsilon_0$, we should be able to conclude that $t\ m$ and $t\ m'$ are equal up to sizes even though these terms have types Nat $m$ and Nat $m'$ respectively.

Morphisms of PRGraph families are again relation-preserving functions; we just need to carry around some more indices. To be precise, a *morphism between PRGraph families $T$ and $U$* over the PRGraph $\Delta$ is a family of functions

$$f_\delta : T(\delta) \to U(\delta) \quad \forall \delta : \Delta$$

such that

$$x \approx_{T,p} y \Rightarrow f(x) \approx_{U,p} f(y) \quad \forall p : \delta \approx_\Delta \delta';\ x : T(\delta);\ y : T(\delta').$$

PRGraph families over a given PRGraph $\Delta$ and their morphisms form an obvious category which we call PRGraphFams($\Delta$).

The last basic construction on PRGraph families is the interpretation of size substitution. Size substitutions $\sigma : \Delta \Rightarrow \Omega$ will be modelled as PRGraph morphisms $[\![\sigma]\!] : [\![\Delta]\!] \to [\![\Omega]\!]$, so we must say what it means to apply such a morphism to a family of PRGraphs over $[\![\Omega]\!]$. To that end, we define *semantic substitution*: given a PRGraph morphism $\sigma : \Delta \to \Omega$ and a family of PRGraphs $T$ over $\Omega$, the application of $\sigma$ to $T$, $T[\sigma]$, is the following PRGraph family over $\Delta$:

$$\begin{aligned}
(T[\sigma])(\delta) &:= T(\sigma(\delta)) \\
\approx_{T[\sigma],p} &:= \approx_{T,\sigma(p)} \\
\mathrm{refl}_{T[\sigma]}(x) &:= \mathrm{refl}_T(x).
\end{aligned}$$

The last equation is not immediately well-typed: $\mathrm{refl}_T(x)$ proves $x \approx_{T,\mathrm{refl}_\Delta(\sigma(\delta))} x$ (assuming $x : T(\sigma(\delta))$) whereas we need to prove $x \approx_{T,\sigma(\mathrm{refl}_\Delta(\delta))} x$. However, since $\approx_\Delta$ is propositional, $\mathrm{refl}_\Delta(\sigma(\delta))$ and $\sigma(\mathrm{refl}_\Delta(\delta))$ are, in fact, the same.

## 4.3 Properties of PRGraphFams

As usual, we will interpret the contexts of λST as finite products of types and the function space as an exponential. We therefore prove that PRGraphFams($\Delta$) admits these constructions. Throughout this section, $\Delta$ is an arbitrary PRGraph.

### 4.3.1 Finite Products

The following results are formalised in Model.Terminal and Model.Product.

Let $T, U$ be two RGraph families over $\Delta$. The *product of $T$ and $U$*, $T \times U$, is defined pointwise. Explicitly, it is the following RGraph family:

$$\begin{aligned}
(T \times U)(\delta) &:= T(\delta) \times U(\delta) \\
(x, x') \approx_{T\times U,p} (y, y') &:= x \approx_{T,p} y \wedge x' \approx_{U,p} y' \\
\mathrm{refl}_{T\times U}(x, y) &:= (\mathrm{refl}_T(x), \mathrm{refl}_U(y)).
\end{aligned}$$

In the first equation, the $\times$ on the right-hand side is the usual cartesian product of types. The projections out of $T \times U$, $\pi_1$ and $\pi_2$, are the usual projections out of the product type. It is easy to check that this defines a product in PRGraphFams($\Delta$).

To interpret all finite products, we additionally need a *terminal object*. This is the PRGraph family $\top$ with

$$\begin{aligned}
\top(\delta) &:= \top \\
x \approx_{\top,p} y &:= \top.
\end{aligned}$$

On the right-hand sides of the equations, $\top$ is the unit type. The relation $\approx_\top$ is trivial and so obviously reflexive.

### 4.3.2 Exponentials

The following results are formalised in Model.Exponential.

Before we can construct exponentials, we need an auxiliary definition: given a family of PRGraphs $T$ over $\Delta$ and an object $\delta$ of $\Delta$, the *application of $T$ to $\delta$*, $\mathrm{Ap}(T, \delta)$, is the following PRGraph:

$$\begin{aligned}
\mathrm{Ap}(T, \delta) &:= T(\delta) \\
\approx_{\mathrm{Ap}(T,\delta)} &:= \approx_{T,\mathrm{refl}_\Delta(\delta)} \\
\mathrm{refl}_{\mathrm{Ap}(T,\delta)}(x) &:= \mathrm{refl}_T(x).
\end{aligned}$$

Using Ap, we define exponentials. Let $T$ and $U$ be PRGraph families over $\Delta$. The *exponential of $T$ and $U$*, $T \rightsquigarrow U$, is the following PRGraph family:

$$\begin{aligned}
(T \rightsquigarrow U)(\delta) &:= \mathrm{Ap}(T, \delta) \to \mathrm{Ap}(U, \delta) \\
f \approx_{T \rightsquigarrow U, p} g &:= \forall x, y.\ x \approx_{T,p} y \Rightarrow f(x) \approx_{U,p} g(y) \qquad (p : \delta \approx_\Delta \delta').
\end{aligned}$$

$\mathrm{Ap}(T, \delta) \to \mathrm{Ap}(U, \delta)$ is the type of PRGraph morphisms from $\mathrm{Ap}(T, \delta)$ to $\mathrm{Ap}(U, \delta)$. The relation of $T \rightsquigarrow U$ is reflexive because for any such RGraph morphism $f$, $x \approx_{T,\mathrm{refl}_\Delta(\delta)} y$ implies $f(x) \approx_{U,\mathrm{refl}_\Delta(\delta)} f(y)$.

To show that $T \rightsquigarrow U$ is indeed an exponential, we need currying and evaluation morphisms. Given a morphism of PRGraph families $f : T \times U \to V$, we define

$$\begin{aligned}
&\mathrm{curry}(f) : T \to U \rightsquigarrow V \\
&\mathrm{curry}(f)_\delta(t) := \lambda u.\ f_\delta(t, u).
\end{aligned}$$

It is easy to check that $\mathrm{curry}(f)_\delta(t)$ is a PRGraph morphism from $\mathrm{Ap}(U, \delta)$ to $\mathrm{Ap}(V, \delta)$ and that $\mathrm{curry}(f)$ is a morphism of PRGraph families. We also define an evaluation morphism:

$$\begin{aligned}
&\mathrm{eval} : (T \rightsquigarrow U) \times T \to U \\
&\mathrm{eval}_\delta(f, t) := f(t).
\end{aligned}$$

Again, it is easy to check that eval is a morphism of PRGraph families and that $T \rightsquigarrow U$ together with curry and eval is an exponential in PRGraphFams($\Delta$).

## 4.4 Sizes

With the central notions defined, we can proceed to model λST. We start with sizes, whose interpretation is formalised in Model.Size.

A *semantic size* is an ordinal below $\omega \cdot 2$. These ordinals are either natural numbers or they have the form $\omega + i$ for some $i \in \mathbb{N}$. They are ordered by the usual strict order on ordinals, meaning

$$0 < 1 < \cdots < \omega < \omega + 1 < \ldots$$

We write $\leq$ for the reflexive closure of $<$, Size for the type of semantic sizes and $\mathrm{Size}_{<n}$ for the type of semantic sizes $m$ such that $m < n$.

Syntactic sizes are modelled by semantic sizes as follows. A size context $\Delta$ is interpreted as a set of size tuples. A well-typed size in $\Delta$ is interpreted as a function from $[\![\Delta]\!]$ to Size. These interpretations are mutually recursively defined:

$$
\begin{aligned}
[\![()]\!] &:= \top && (\top \text{ is the unit type}) \\
[\![\Delta, n]\!] &:= \Sigma_{\delta : [\![\Delta]\!]} \operatorname{Size}_{< [\![n]\!](\delta)}
\end{aligned}
$$

$$
\begin{aligned}
[\![v_0]\!](\delta, n) &:= n \\
[\![v_{x+1}]\!](\delta, n) &:= [\![v_x]\!](\delta) \\
[\![0]\!](\delta) &:= 0 \\
[\![\uparrow n]\!](\delta) &:= [\![n]\!](\delta) + 1 \\
[\![\infty]\!](\delta) &:= \omega.
\end{aligned}
$$

We will also consider $[\![\Delta]\!]$ as a PRGraph whose underlying type is $[\![\Delta]\!]$ and whose relation is trivial, i.e. any two elements are related. A size interpretation $[\![n]\!]$ is then a PRGraph morphism (if we similarly treat the type Size as a trivial PRGraph).

*Remark.* We will shortly interpret a type $T$ in a context $\Delta$ as a family of PRGraphs indexed by the PRGraph $[\![\Delta]\!]$. However, since $[\![\Delta]\!]$ is always trivial as a PRGraph, we could also interpret $T$ as a family of PRGraphs indexed by the *type* $[\![\Delta]\!]$, which would be a little simpler. I choose not to do so because that model would likely not support dependent types (and perhaps other interesting features).

Our interpretation of sizes is well-behaved with respect to size comparison:

**Lemma 4.1** (Interpretation of $<$)**.** *If* $\Delta \vdash n < m$ *and* $\delta : [\![\Delta]\!]$*, then* $[\![n]\!](\delta) < [\![m]\!](\delta)$.

*Proof.* By induction on the derivation of $\Delta \vdash n < m$. In the case where $n$ is a variable, we need to use the following fact: given $\Delta \vdash n$, $\Delta \vdash m$ and $(\delta, o) : [\![\Delta, n]\!]$, we have $[\![\mathrm{wk}(m)]\!](\delta, o) = [\![m]\!](\delta)$. □

This concludes the interpretation of sizes. Next, we interpret size substitutions as functions between interpretations of size contexts. As explained in Sec. 5.2, the results presented here are not exactly those formalised.

Let $\sigma : \Delta \Rightarrow \Omega$ be a well-typed substitution. The interpretation of $\sigma$ is a function from $[\![\Delta]\!]$ to $[\![\Omega]\!]$ defined by recursion over the derivation of $\sigma : \Delta \Rightarrow \Omega$:

$$
\begin{aligned}
&[\![()]\!] : [\![\Delta]\!] \to [\![()]\!] \\
&[\![()]\!](\delta) := tt \qquad (tt \text{ is the unique value of } \top)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\sigma, n]\!] : [\![\Delta]\!] \to \Sigma_{\omega : [\![\Omega]\!]} \operatorname{Size}_{< [\![m]\!](\omega)} \\
&[\![\sigma, n]\!](\delta) := [\![\sigma]\!](\delta), n.
\end{aligned}
$$

The last equation is well-typed because $\sigma, n : \Delta \Rightarrow \Omega, m$ implies $\Delta \vdash n < m[\sigma]$ and thus by Lemma 4.1 $[\![n]\!] < [\![m[\sigma]]\!]$. The following Lemma 4.2 then lets us conclude $[\![n]\!] < [\![m]\!]([\![\sigma]\!](\delta))$. (This means that the present definition and Lemma 4.2 are in fact mutually recursive.)

**Lemma 4.2** (Interpretation of size substitutions is correct)**.** *If* $\sigma : \Delta \Rightarrow \Omega$ *and* $\Omega \vdash n$*, then* $[\![\sigma[n]]\!] = [\![n]\!] \circ [\![\sigma]\!]$.

We will also need to know the interpretations of some of the specific substitutions from Sec. 3.3. The following lemma tells us how they behave when applied to a size valuation.

**Lemma 4.3** (Interpretation of specific size substitutions)**.** *Let* $(\omega, m, k) : [\![\Omega, n, v_0]\!]$, $\sigma : \Delta \to \Omega$ *and* $\tau : \Omega \to \Omega'$*. Then we have*

$$\begin{aligned}
[\![\mathrm{Id}]\!](\omega) &= \omega \\
[\![\sigma \gg \tau]\!](\omega) &= [\![\tau]\!]([\![\sigma]\!](\omega)) \\
[\![\mathrm{Wk}]\!](\omega, m) &= \omega \\
[\![\mathrm{Sing}(o)]\!](\omega) &= (\omega, [\![o]\!](\omega)) \\
[\![\mathrm{Lift}(\sigma)]\!](\omega, m) &= ([\![\sigma]\!](\omega), m) \\
[\![\mathrm{Skip}]\!](\omega, m, k) &= (\omega, k).
\end{aligned}$$

## 4.5 Types

In this section, we model the types of λST. Our results are formalised in Model.Nat, Model.Stream and Model.Quantification.

As foreshadowed, we interpret well-scoped types $T$ in a size context $\Delta$ as families of PRGraphs $[\![T]\!]$ over the PRGraph $[\![\Delta]\!]$. The interpretations of sized naturals, sized streams and functions are straightforward; only size quantification will be slightly more complex.

Let $\mathbb{N}_{\leq n}$ be the type of natural numbers $m$ such that $m \leq n$ (with $n$ an arbitrary size). The interpretation of $\Delta \vdash \mathrm{Nat}\ n$ is the following family of PRGraphs:

$$\begin{aligned}
[\![\mathrm{Nat}\ n]\!](\delta) &:= \mathbb{N}_{\leq [\![n]\!](\delta)} \\
i \approx_{[\![\mathrm{Nat}\ n]\!], p} j &:= i = j.
\end{aligned}$$

Streams are interpreted in a similar fashion:

$$\begin{aligned}
[\![\mathrm{Stream}\ n]\!](\delta) &:= \mathbb{N}_{\leq [\![n]\!](\delta)} \to \mathbb{N} \\
f \approx_{[\![\mathrm{Stream}\ n]\!], p} g &:= \forall m \leq \min([\![n]\!](\delta), [\![n]\!](\delta')).\ f(m) = g(m) \qquad (p : \delta \approx_\Delta \delta').
\end{aligned}$$

The function space is interpreted by the exponential of PRGraph families from Sec. 4.3.2:

$$[\![T \to U]\!] := [\![T]\!] \rightsquigarrow [\![U]\!].$$

Size quantification requires a little more effort. We want to think of the terms of $\forall n.\ T$ as functions which take a size argument, but they should be parametric in that argument: when we apply such a function to different sizes, it should return results that are equal up to sizes. To be more precise, let $T$ be a family of PRGraphs over $[\![\Delta, n]\!]$ and let $\delta : [\![\Delta]\!]$. A function

$$f : (m : \mathrm{Size}_{< [\![n]\!](\delta)}) \to T(\delta, m)$$

is *size-parametric* if

$$f(m) \approx_T f(m') \quad \forall m, m' : \mathrm{Size}_{<[\![n]\!](\delta)}.$$

This equation is well-typed because $\approx_{[\![\Delta, n]\!]}$ is trivial, so in particular $(\delta, m) \approx_{[\![\Delta, n]\!]} (\delta, m')$. We write $\mathrm{Param}(T, \delta)$ for the type of size-parametric functions into $T(\delta, \bullet)$.

Now we can interpret size quantification: the interpretation of $\Delta \vdash \forall n.\ T$ is the PRGraph family $\Pi([\![n]\!], [\![T]\!])$ with

$$\begin{aligned} \Pi(n, T)(\delta) &:= \mathrm{Param}(T, \delta) \\ f \approx_{\Pi(n,T)} g &:= \forall m, m'.\ f(m) \approx_T g(m'). \end{aligned}$$

If we think of $\approx_T$ as equality up to sizes, $[\![\forall n.\ T]\!]$ reflects the intuition that when we apply a term in $\forall n.\ T$ to a size $m$, it does not matter – up to sizes – which $m$ we choose.

Having interpreted all types, we can now move to type contexts. These are mere lists of types, so they are, as usual, modelled by finite products:

$$\begin{aligned} [\![()]\!] &:= \top \\ [\![\Gamma, T]\!] &:= [\![\Gamma]\!] \times [\![T]\!]. \end{aligned}$$

$\top$ is the terminal PRGraph family and $\times$ is the product of PRGraph families, both defined in Sec. 4.3.1.

With this, we have interpreted all type-level constructs of λST. What remains is to prove a lemma about the correspondence between syntactic and semantic substitution. To do that, we need some technical lemmas about semantic substitution first:

**Lemma 4.4** (Substitution in exponentials)**.** *Let $T$, $U$ be PRGraph families over $\Omega$ and $\sigma : \Delta \to \Omega$ a PRGraph morphism. Then*

$$(T \rightsquigarrow U)[\sigma] = T[\sigma] \rightsquigarrow U[\sigma].$$

**Lemma 4.5** (Substitution in quantifications)**.** *Let $T$ be a PRGraph family over $[\![\Omega, n]\!]$ and $\sigma : \Delta \Rightarrow \Omega$ a well-typed substitution. Then*

$$\Pi(n, T)\big[[\![\sigma]\!]\big] = \Pi(n\big[[\![\sigma]\!]\big], T\big[[\![\mathrm{Lift}(\sigma)]\!]\big]).$$

*Remark.* The correct equality for PRGraph families is isomorphism in $\mathrm{PRGraphFams}(\Delta)$, so the $=$ above really means $\cong$. This abuse of notation is justified because in the formalisation, isomorphic PRGraph families are in fact propositionally equal (by univalence).

From the previous two lemmas follows the main substitution lemma for types and contexts:

**Lemma 4.6** (Interpretation of substitution in types and contexts is correct)**.** *If $\Omega \vdash T$ and $\sigma : \Delta \Rightarrow \Omega$, then*

$$[\![T[\sigma]]\!] = [\![T]\!]\big[[\![\sigma]\!]\big]$$

*and similar for contexts.*

$$\frac{\Delta,\, n; \Gamma[\mathrm{Wk}] \vdash t : T \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \lambda\, n.\, t : \forall n.\, T} \qquad \frac{\Delta; \Gamma \vdash t : \forall n.\, T \qquad \Delta \vdash m < n}{\Delta; \Gamma \vdash t\ m : T\big[\mathrm{Sing}(m)\big]}$$

**Figure 4.1:** Typing rules for terms related to size quantification

## 4.6 Terms

Well-typed terms (more precisely, judgments $\Delta; \Gamma \vdash t : T$) are interpreted as morphisms between the PRGraph families $[\![\Gamma]\!]$ and $[\![T]\!]$. In this section, we show a natural interpretation of each of the terms of λST. Formalisations of these definitions can be found in Model.Term.

### 4.6.1 Functions

Recall that the function space is modelled by the exponential $[\![T]\!] \rightsquigarrow [\![U]\!]$ as in the standard categorical model of the simply-typed lambda calculus. Accordingly, the interpretations of abstraction and application are also standard.

To interpret $\lambda\, T.\, t$, we may assume a morphism $[\![t]\!] : [\![\Gamma]\!] \times [\![T]\!] \to [\![U]\!]$. We then define

$$\begin{aligned} [\![\lambda\, T.\, t]\!] &: [\![\Gamma]\!] \to [\![T]\!] \rightsquigarrow [\![U]\!] \\ [\![\lambda\, T.\, t]\!] &\coloneqq \mathrm{curry}([\![t]\!]). \end{aligned}$$

The interpretation of $t\ u$, assuming $[\![t]\!] : [\![\Gamma]\!] \to [\![T \rightsquigarrow U]\!]$ and $[\![u]\!] : [\![\Gamma]\!] \to [\![T]\!]$, is

$$\begin{aligned} [\![t\ u]\!] &: [\![\Gamma]\!] \to [\![U]\!] \\ [\![t\ u]\!] &\coloneqq \mathrm{eval} \circ \langle [\![t]\!] \times [\![u]\!] \rangle. \end{aligned}$$

The operator

$$\langle \bullet \times \bullet \rangle : (A \to A') \to (B \to B') \to A \times B \to A' \times B'$$

can be defined in any category with products.

### 4.6.2 Size Quantification

To interpret size quantification, we must consider size abstractions and size applications. Their typing rules are reproduced in Fig. 4.1 for reference.

When interpreting size quantification, we may assume a morphism $[\![t]\!] : [\![\Gamma[\mathrm{Wk}]]\!] \to [\![T]\!]$ with components

$$\begin{aligned} [\![t]\!]_{(\delta,n)} &: [\![\Gamma[\mathrm{Wk}]]\!](\delta, n) \to [\![T]\!](\delta, n) \\ &= [\![\Gamma]\!]\big[[\![\mathrm{Wk}]\!]\big](\delta, n) \to [\![T]\!](\delta, n) && \text{(by Lemma 4.6)} \\ &= [\![\Gamma]\!]([\![\mathrm{Wk}]\!](\delta, n)) \to [\![T]\!](\delta, n) && \text{(by definition)} \\ &= [\![\Gamma]\!](\delta) \to [\![T]\!](\delta, n) && \text{(by Lemma 4.3).} \end{aligned}$$

$$\frac{\Delta \vdash n < \uparrow \infty \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \text{zero } n : \text{Nat } n} \qquad \frac{\Delta \vdash n < \uparrow \infty \qquad \Delta \vdash m < n \qquad \Delta; \Gamma \vdash i : \text{Nat } m}{\Delta; \Gamma \vdash \text{suc } n\ m\ i : \text{Nat } n}$$

$$\frac{\Delta; \Gamma \vdash i : \text{Nat } n \qquad \begin{array}{c}\Delta \vdash T \qquad \Delta \vdash n < \uparrow \infty \\ \Delta; \Gamma \vdash z : T\end{array} \qquad \Delta; \Gamma \vdash s : \forall n.\ \text{Nat } v_0 \to T[\text{Wk}]}{\Delta; \Gamma \vdash \text{caseNat}[T]\ n\ i\ z\ s : T}$$

**Figure 4.2:** Typing rules for terms related to Nat $n$

We then define

$$[\![\lambda\, n.\ t]\!] \ :\ [\![\Gamma]\!] \to [\![\forall n.\ T]\!]$$
$$[\![\lambda\, n.\ t]\!]_\delta(\gamma) \coloneqq \lambda\, m < [\![n]\!](\delta).\ [\![t]\!]_{(\delta,m)}(\gamma).$$

This definition is valid due to the following observations:

- The right-hand side of the equation is a size-parametric function. Recall that for all $m$ and $m'$, $(\delta, m) \approx_{[\![\Delta, n]\!]} (\delta, m')$. We also have $\gamma \approx_{[\![\Gamma]\!]} \gamma$ by reflexivity and this implies $[\![t]\!]_{(\delta,m)}(\gamma) \approx_{[\![T]\!]} [\![t]\!]_{(\delta,m')}(\gamma)$ because $[\![t]\!]$ is a morphism of PRGraph families.
- $[\![\lambda\, n.\ t]\!]$ is a morphism of PRGraph families. This follows from essentially the same argument, only replacing the fact $\gamma \approx_{[\![\Gamma]\!]} \gamma$ with an assumption $\gamma \approx_{[\![\Gamma]\!]} \gamma'$ for some arbitrary $\gamma'$.

Moving to size application, we may assume a morphism

$$[\![t]\!] \ :\ [\![\Gamma]\!] \to [\![\forall n.\ T]\!]$$

and we construct the morphism

$$\begin{aligned} [\![t\ m]\!]_\delta \quad &: \ [\![\Gamma]\!](\delta) \to [\![T\left[\text{Sing}(m)\right]]\!] \\ &= \ [\![\Gamma]\!](\delta) \to [\![T]\!](\delta, [\![m]\!](\delta)) \qquad \text{(by Lemmas 4.6 and 4.3)} \\ [\![t\ m]\!]_\delta(\gamma) &\coloneqq [\![t]\!]_\delta(\gamma, [\![m]\!](\delta)). \end{aligned}$$

The size $[\![m]\!](\delta)$ is a valid argument to $[\![t]\!]_\delta(\gamma)$ because we have $\Delta \vdash m < n$ and thus (by Lemma 4.1) $[\![m]\!](\delta) < [\![n]\!](\delta)$ for all $\delta$. $[\![t\ m]\!]$ is a morphism of PRGraph families because $[\![t]\!]$ is.

### 4.6.3 Natural Numbers

The sized natural number type Nat $n$ has two constructors, zero and successor, and an eliminator, caseNat. Their typing rules are reproduced in Fig. 4.2. Recall that Nat $n$ is modelled by the type of natural numbers less than or equal to a size $n$.

The term zero $n$ is interpreted by

$$[\![\text{zero } n]\!]_\delta(\gamma) \coloneqq 0.$$

$$\frac{\Delta \vdash n < \uparrow \infty \qquad \Delta;\Gamma \vdash i : \text{Nat}\ \infty \qquad \Delta;\Gamma \vdash is : \forall n.\ \text{Stream}\ \upsilon_0}{\Delta;\Gamma \vdash \text{cons}\ n\ i\ is : \text{Stream}\ n}$$

$$\frac{\Delta \vdash n < \uparrow \infty \qquad \Delta;\Gamma \vdash is : \text{Stream}\ n}{\Delta;\Gamma \vdash \text{head}\ n\ is : \text{Nat}\ \infty}$$

$$\frac{\Delta \vdash n < \uparrow \infty \qquad \Delta;\Gamma \vdash is : \text{Stream}\ n \qquad \Delta \vdash m < n}{\Delta;\Gamma \vdash \text{tail}\ n\ is\ m : \text{Stream}\ m}$$

**Figure 4.3:** Typing rules for terms related to Stream $n$

This is valid since $0 \leq n$ for all $n$.

To interpret the successor, given a morphism $[\![i]\!] : [\![\Gamma]\!] \to [\![\text{Nat}\ n]\!]$, we define

$$[\![\text{suc}\ n\ m\ i]\!]_\delta(\gamma) \coloneqq [\![i]\!]_\delta(\gamma) + 1.$$

This is valid because we have $[\![i]\!]_\delta(\gamma) \leq [\![m]\!](\delta)$ (by definition of $[\![\text{Nat}\ m]\!]$) and $[\![m]\!](\delta) < [\![n]\!](\delta)$ (by Lemma 4.1), which implies $[\![i]\!]_\delta(\gamma) + 1 \leq [\![n]\!](\delta)$.

To interpret the eliminator caseNat, we assume morphisms $[\![i]\!] : [\![\Gamma]\!] \to [\![\text{Nat}\ n]\!]$, $[\![z]\!] : [\![\Gamma]\!] \to [\![T]\!]$ and $[\![s]\!] : [\![\Gamma]\!] \to [\![\forall n.\ \text{Nat}\ \upsilon_0 \to T[\text{Wk}]]\!]$. The interpretation is then

$$[\![\text{caseNat}[T]\ n\ i\ z\ s]\!]_\delta(\gamma) \coloneqq \begin{cases} [\![z]\!]_\delta(\gamma) & \text{if } [\![i]\!]_\delta(\gamma) = 0 \\ [\![s]\!]_\delta(\gamma, [\![n]\!](\delta) - 1, [\![i]\!]_\delta(\gamma) - 1) & \text{otherwise.} \end{cases}$$

Note that in the second branch, $[\![n]\!](\delta)$ cannot be zero because $[\![i]\!]_\delta(\gamma) \leq [\![n]\!](\delta)$ and $[\![i]\!]_\delta(\gamma) \neq 0$.

### 4.6.4 Streams

Sized streams are constructed by the term cons and eliminated by head and tail. The typing rules of these terms are reproduced in Fig. 4.3. Recall that the model of Stream $n$ is the type of functions $\mathbb{N}_{\leq n} \to \mathbb{N}$.

To interpret cons, we assume morphisms $[\![i]\!]$ and $[\![is]\!]$ with components

$$\begin{aligned}
[\![i]\!]_\delta &: [\![\Gamma]\!](\delta) \to [\![\text{Nat}\ \infty]\!](\delta) \\
&= [\![\Gamma]\!](\delta) \to \mathbb{N}_{<\omega} \\
&= [\![\Gamma]\!](\delta) \to \mathbb{N} \\
[\![is]\!]_\delta &: [\![\Gamma]\!](\delta) \to [\![\forall n.\ \text{Stream}\ \upsilon_0]\!](\delta) \\
&= [\![\Gamma]\!](\delta) \to \text{Param}([\![\text{Stream}\ \upsilon_0]\!], \delta) \\
&\approx [\![\Gamma]\!](\delta) \to (m : \text{Size}_{<[\![n]\!](\delta)}) \to [\![\text{Stream}\ \upsilon_0]\!](\delta, m) \\
&= [\![\Gamma]\!](\delta) \to (m : \text{Size}_{<[\![n]\!](\delta)}) \to \mathbb{N}_{\leq m} \to \mathbb{N}.
\end{aligned}$$

The $\approx$ on the penultimate line indicates that we are ignoring the parametricity requirement of Param to unfold it. We then define

$$[\![\text{cons } n\ i\ is]\!]_\delta(\gamma, k) \coloneqq \begin{cases} [\![i]\!]_\delta(\gamma) & \text{if } k = 0 \\ [\![is]\!]_\delta(\gamma, k-1, k-1) & \text{otherwise.} \end{cases}$$

Note that $k \leq [\![n]\!](\delta)$, so in the second equation we have $k - 1 < [\![n]\!](\delta)$.

The destructors of Stream $n$ have less complex interpretations. For head, we assume a morphism $[\![is]\!] : [\![\Gamma]\!] \to [\![\text{Stream } n]\!]$ and define

$$[\![\text{head } n\ is]\!] : [\![\Gamma]\!] \to [\![\text{Nat } \infty]\!]$$
$$[\![\text{head } n\ is]\!]_\delta(\gamma) \coloneqq [\![is]\!]_\delta(\gamma, 0).$$

Similarly, for tail, we assume $[\![is]\!] : [\![\Gamma]\!] \to [\![\text{Stream } n]\!]$ and $\Delta \vdash m < n$ and define

$$[\![\text{tail } n\ is\ m]\!] : [\![\text{Stream } m]\!]$$
$$[\![\text{tail } n\ is\ m]\!]_\delta(\gamma, k) \coloneqq [\![is]\!]_\delta(\gamma, k+1).$$

The application of $[\![is]\!]$ to $k + 1$ is well-typed because $k \leq [\![m]\!](\delta)$ and $[\![m]\!](\delta) < [\![n]\!](\delta)$ (by Lemma 4.1).

### 4.6.5 Fixpoint

The sized fixpoint combinator is the sole means of recursion in λST. As such, its interpretation is also recursive but otherwise mostly straightforward.

Recall the typing rule of fix:

$$\frac{\Delta, \uparrow\infty \vdash T \qquad \Delta; \Gamma \vdash t : \forall \uparrow \infty.\ (\forall v_0.\ T\left[\text{Skip}\right]) \to T \qquad \Delta \vdash n < \uparrow\infty}{\Delta; \Gamma \vdash \text{fix}[T]\ t\ n : T\left[\text{Sing}(n)\right]}$$

According to this rule, we may assume a morphism $[\![t]\!]$ with components

$$\begin{aligned} [\![t]\!]_\delta\ &:\ [\![\Gamma]\!](\delta) \to [\![\forall \uparrow \infty.\ (\forall v_0.\ T\left[\text{Skip}\right]) \to T]\!](\delta) \\ &= [\![\Gamma]\!](\delta) \to (n : \text{Size}_{<\omega+1}) \to ((m : \text{Size}_{<n}) \to [\![T]\!]([\![\text{Skip}]\!](\delta, n, m))) \to [\![T]\!](\delta, n) \\ &= [\![\Gamma]\!](\delta) \to (n : \text{Size}_{<\omega+1}) \to ((m : \text{Size}_{<n}) \to [\![T]\!](\delta, m)) \to [\![T]\!](\delta, n). \end{aligned}$$

The first step in this chain uses the fact that $[\![T\left[\text{Skip}\right]]\!] = [\![T]\!]\left[[\![\text{Skip}]\!]\right]$ (by Lemma 4.6). In the second step, we simplify $[\![\text{Skip}]\!](\delta, n, m)$ to $(\delta, m)$ (by Lemma 4.3).

Given such a $[\![t]\!]$, we first define the following auxiliary function:

$$f_\delta : [\![\Gamma]\!](\delta) \to (n : \text{Size}_{<\omega+1}) \to [\![T]\!](\delta, n)$$
$$f_\delta(\gamma, n) \coloneqq [\![t]\!]_\delta(\gamma, n, \lambda\, m.\ f_\delta(\gamma, m)).$$

This function is a standard fixpoint of $[\![t]\!]_\delta$. It terminates because the recursive call is on a size $m < n$ and $<$ is well-founded: there is no infinite descending chain $n_0 > n_1 > \ldots$ and

so a recursion which always decreases the size argument must eventually stop. Interpreting fix is then just a matter of applying $f$:

$$\begin{aligned}
[\![\mathrm{fix}[T]\ t\ n]\!]_\delta &: [\![\Gamma]\!](\delta) \to [\![T\left[\mathrm{Sing}(n)\right]]\!](\delta) \\
&= [\![\Gamma]\!](\delta) \to [\![T]\!](\delta, [\![n]\!](\delta)) \\
[\![\mathrm{fix}[T]\ t\ n]\!]_\delta(\gamma) &\coloneqq f_\delta(\gamma, [\![n]\!](\delta)).
\end{aligned}$$

In the above argument, we have, as usual, glossed over various proof obligations concerning size parametricity, preservation of relations and so on. However, one of these obligations is actually nontrivial: in the definition of the auxiliary function $f$, we use $f$ recursively as a size-parametric function. This means that we must prove $f$ size-parametric *mutually recursively* with its own definition. The full, somewhat gory details of this construction appear in Sec. 5.3.

## 4.7 Size Irrelevance

With all elements of λST modelled, we should briefly ask what the model tells us. It does not immediately allow us to conclude any syntactic size irrelevance results, which would mean, for example, that for any term $t : \forall n.\ T$ and sizes $m$, $m'$ the applications $t\ m$ and $t\ m'$ normalise to the same value up to sizes. (Indeed, I have not even given an operational semantics for λST.) Ideally, we would like to prove the even stronger claim that the entire reduction sequences of $t\ m$ and $t\ m'$ are equal up to sizes. This would allow us to fully erase sizes at runtime. Deriving answers to these questions from the model is significant additional work; I briefly return to them in Sec. 7.2.

What we do get from the model, however, is a denotational semantics of λST with "baked-in" size irrelevance. Our model construction can be read as an interpreter from λST into Agda, and this interpreter must, due to the parametricity requirement in our interpretation of size quantification, produce size-irrelevant functions.

# 5
# Formalisation

All results from the previous sections are formalised in the dependently typed proof assistant Agda. The formalisation is freely available online [16] and closely follows the informal discussion in this thesis, so in this chapter I only discuss the major design decisions that shape it.

## 5.1 Metatheory

λST is formalised in Agda [19], a proof assistant based loosely on Luo's UTT [17]. I use version 2.6.0.1 with axiom K disabled via the `--without-K` flag. This configuration disables Agda's sized types, so we do not reason circularly.

I extend the metatheory in one major way, by postulating the axiom of univalence [27]. Type theory with univalence has a model in simplicial sets [15], so this should be safe (as long as axiom K, which contradicts univalence, is disabled). Univalence allows us to avoid some of the boilerplate that is often needed when interpreting set-theoretical mathematics in type theory. In particular, with univalence, propositional equality is the correct notion of equality for all the constructions we work with, such as streams and reflexive graphs. Without univalence, we would have to define custom equivalence relations for these structures, which would in turn force us to parameterise the definitions of reflexive graphs and families of reflexive graphs by an equivalence relation. This is doable – and I conjecture that the formalisation could be adapted more or less mechanically to a type theory without univalence – but the resulting complications would obscure the constructions.

As an alternative to postulated univalence, we could also work in Cubical Agda [31], an extension of Agda that admits univalence as a theorem. Compared with postulated univalence, Cubical Agda has the advantage that more equations hold definitionally rather than just propositionally. However, in our particular case, we would not use these additional definitional equalities much. On the other hand, switching to Cubical Agda would complicate the interoperation with Agda's standard library, which does not use the cubical primitives (specifically the cubical identity type).

To benefit from univalence, I reimplement some basic definitions and results from Homotopy Type Theory such as H-Levels, the theory of equivalences and the derivation of functional extensionality from univalence. For reference, see the Homotopy Type Theory book [27]. Much of my code is inspired by Martín Escardó's implementation of Homotopy Type Theory in Agda [13].

## 5.2 Size Substitutions

The one instance where the formalisation departs somewhat from the account in this thesis concerns size substitutions. If we were to stick exactly to the text, we would face the following problem: when we apply the interpretation of the weakening substitution Wk to a size valuation $(\delta, n)$, we would get $[\![\mathrm{Wk}]\!](\delta, n) = \delta$ as a *propositional* equality, but not as a *definitional* equality (and similar for the other substitutions). This matters because $[\![\mathrm{Wk}]\!](\delta, n)$ sometimes appears in types, in which case we would have to use the transport operator `subst` to define terms of these types. This, in turn, makes it hard to prove facts about these terms.

To avoid this issue, we use a universe of size substitutions, formalised in Source.Size.Substitution.Universe. We define a data type `Sub` that has a constructor for each of the particular substitutions we need (e.g. Wk, Sing). We then define an interpretation function which maps each value of this datatype to the corresponding "canonical" substitution from Sec. 3.3. The application of a substitution in `Sub` to a size/type/context/term is then defined simply as the application of its canonical interpretation.

This setup gives us the best of both worlds: all lemmas about canonical substitutions are easily transferred to the universe construction, but when we model substitutions in `Sub` as PRGraph morphisms, we can *define* $[\![\mathrm{Wk}]\!](\delta, n) \coloneqq n$. This is exactly the definitional equality we wanted.

## 5.3 Interpretation of the Fixpoint

In Sec. 4.6.5, I mentioned that the interpretation of the term fix is not quite as simple as I made it appear. Recall that the typing rule of fix lets us assume a morphism $[\![t]\!]$ with components

$$[\![t]\!]_\delta : [\![\Gamma]\!](\delta) \to (n : \mathrm{Size}_{<\omega+1}) \to ((m : \mathrm{Size}_{<n}) \to [\![T]\!](\delta, m)) \to [\![T]\!](\delta, n).$$

We then defined the following fixpoint of $[\![t]\!]$:

$$f : [\![\Gamma]\!](\delta) \to (n : \mathrm{Size}_{<\omega+1}) \to [\![T]\!](\delta, n)$$
$$f(\gamma)(n) \coloneqq [\![t]\!](\gamma)(n)(\lambda\, m.\ f(\gamma)(m)).$$

This definition is easily replicated in Agda using well-founded recursion [21, 18], a technique that is commonly used to define non-structurally recursive functions. We will use a variation of this technique, so let us recall the basics first.

Well-founded recursion encodes the idea that given a well-founded relation $<$, a recursive definition which decreases this relation with every recursive call must terminate – for if it did not, we would have an infinite descending chain $n_0 > n_1 > \ldots$ and 'well-founded' means precisely that no such chain exists. Our definition of $f$ above follows this scheme: the recursive call is applied to $m < n$ and since $<$ is well-founded, the recursion must eventually stop.

To represent this idea in Agda, we first define what it means for a relation to be well-founded. To that end, we introduce the auxiliary concept of *accessibility* in the form of the `Acc` data type. This type can be found in the module Induction.WellFounded from Agda's

standard library; the remainder of this section is located either in the same module or in my Util.Induction.WellFounded.

```
data Acc (_<_ : A → A → Set) (x : A) : Set where
  acc : (∀ y → y < x → Acc _<_ y) → Acc _<_ x
```

This definition says that a term $x : A$ is accessible with respect to the relation $<$ if every $y < x$ is also accessible. `Acc` is inductively defined, which means that a proof of accessibility can "stack" only finitely many `acc` constructors. Therefore, when a term is accessible, any descending chain starting from that term must eventually end. This justifies the following definition of well-foundedness: a relation $<$ on some type $A$ is well-founded if every element of $A$ is accessible with respect to $<$.

```
WellFounded : (A → A → Set) → Set
WellFounded _<_ = ∀ x → Acc _<_ x
```

We can then encode well-founded recursion. First, we need the following helper function:

```
wfInd-acc : (P : A → Set)
  → (∀ x → (∀ y → y < x → P y) → P x)
  → ∀ x → Acc _<_ x → P x
wfInd-acc P f x (acc rs) = f x (λ y y<x → wfInd-acc P f y (rs y y<x))
```

The definition of `wfInd-acc` passes Agda's termination checker because it recurses on the argument `Acc _<_ x`: in the recursive call, that argument is `rs y y<x`, which counts as a syntactic subterm of `acc rs`. Bona fide well-founded recursion is a direct consequence of `wfInd-acc`:

```
wfInd : WellFounded _<_
  → (P : A → Set γ)
  → (∀ x → (∀ y → y < x → P y) → P x)
  → ∀ x → P x
wfInd <-wf P f x = wfInd-acc P f x (<-wf x)
```

The reader is invited to check that our earlier definition of the fixpoint $f$ is an instance of this recursion scheme.

However, that earlier definition is built on a white lie: The type of $[\![t]\!]$ is really more complex than advertised. Specifically, its argument of type $(m : \mathrm{Size}_{<n}) \to [\![T]\!](\delta, m)$ must be size-parametric. In the definition of $f$, that argument is the function $\lambda\, m.\ f(\gamma)(m)$ – so to define $f$, we must prove that $f$ is size-parametric. Our well-founded recursion combinator `wfInd` is too weak for this situation.

We therefore need to construct a variation of `wfInd` that allows us to prove properties of a defined function while defining it. We call this variation `wfIndΣ`. Like `wfInd`, it is an instance of a helper function, `wfIndΣ-acc`, which does most of the heavy lifting. That helper, and a mutually defined proof about it, is shown in Fig. 5.1.

Compared to `wfInd-acc`, the type of `wfIndΣ-acc` has changed in the following ways:

- `Q` is a heterogeneous relation on `P`, the (dependent) type which we are constructing. In our use case, `Q` will be instantiated with equality up to sizes.

```
mutual
  wfIndΣ-acc : (P : A → Set) (Q : ∀ x y → P x → P y → Set)
    → (f : ∀ x
        → (g : ∀ y → y < x → P y)
        → (∀ y y<x z z<x → Q y z (g y y<x) (g z z<x))
        → P x)
    → (∀ x g g-resp y h h-resp
        → (∀ z z<x z′ z′<y → Q z z′ (g z z<x) (h z′ z′<y))
        → Q x y (f x g g-resp) (f y h h-resp))
    → ∀ x → Acc _<_ x → P x
  wfIndΣ-acc P Q f f-resp x (acc rs)
    = f x
      (λ y y<x → wfIndΣ-acc P Q f f-resp y (rs y y<x))
      (λ y y<x z z<x
          → wfIndΣ-acc-resp P Q f f-resp y (rs y y<x) z (rs z z<x))

  wfIndΣ-acc-resp : (P : A → Set) (Q : ∀ x y → P x → P y → Set)
    → (f : ∀ x
        → (g : ∀ y → y < x → P y)
        → (∀ y y<x z z<x → Q y z (g y y<x) (g z z<x))
        → P x)
    → (f-resp : ∀ x g g-resp y h h-resp
        → (∀ z z<x z′ z′<y → Q z z′ (g z z<x) (h z′ z′<y))
        → Q x y (f x g g-resp) (f y h h-resp))
    → ∀ x x-acc y y-acc
    → Q x y
        (wfIndΣ-acc P Q f f-resp x x-acc)
        (wfIndΣ-acc P Q f f-resp y y-acc)
  wfIndΣ-acc-resp P Q f f-resp x (acc rsx) y (acc rsy)
    = f-resp _ _ _ _ _ _ λ z z<x z′ z′<y
        → wfIndΣ-acc-resp P Q f f-resp z (rsx z z<x) z′ (rsy z′ z′<y)
```

**Figure 5.1:** Implementation of wfIndΣ-acc and wfIndΣ-acc-resp

- The function `f`, which defines the computational behaviour of the fixpoint, still receives as an argument a function `g` which represents a recursive call on a smaller argument. Additionally, `f` now receives a proof that for any two smaller sizes `y` and `z`, the recursive calls `g y` and `g z` are related by `Q`. In our case, this means that `g` is size-parametric. This is precisely the ingredient we were previously missing.
- The next argument to `wfIndΣ-acc` is new. It demands a proof that `f` preserves the relation `Q`.

Mutually with `wfIndΣ-acc`, we define a lemma about it, `wfIndΣ-acc-resp`. This just says that any two terms constructed by `wfIndΣ-acc` are related by `Q` – so in our case, `wfIndΣ-acc` constructs a size-parametric function. As with `wfInd-acc`, the implementations of `wfIndΣ-acc` and `wfIndΣ-acc-resp` are straightforward.

It is now easy to define the fixpoint combinator `wfIndΣ` based on `wfIndΣ-acc`. Defining $f$ in terms of it is also conceptually simple but the details are a bit gnarly. For the full picture, see the module Model.Term.

# 6

# Negative Results

Before developing the reflexive graph model of λST, I explored two other modelling approaches which turned out, for different reasons, to be unsuited to the task at hand. They are both based on the idea of interpreting size contexts as preorder categories, types as functors from a size context category into the category of sets and terms as natural transformation between such functors. A model like this would yield a sort of size irrelevance in the form of the naturality of term interpretations: if the interpretation of a term in size context $\Delta$ is natural in $[\![\Delta]\!]$, then $[\![\Delta]\!]$ must be essentially irrelevant. In the next two sections, I briefly describe these potential models and why they do not, in fact, model λST. This is in somewhat surprising contrast to the work of Veltri and van der Weide [29], who model guarded recursion – another form of type-based termination checking that is quite similar to sized types – using essentially the same approach that fails for λST.

## 6.1 Covariant Presheaf Model

Our first potential model starts with the observation that size contexts have a natural interpretation as categories. Recall our previous interpretation of size contexts as types:

$$\begin{aligned}[\![()]\!] &= \top \\ [\![\Delta, n]\!] &= \Sigma_{\delta \in [\![\Delta]\!]} \, \mathrm{Size}_{<[\![n]\!](\delta)} \, .\end{aligned}$$

We can extend this interpretation from types to categories: $[\![()]\!]$ is the terminal category (whose object type is $\top$) and $[\![\Delta, n]\!]$ is the Grothendieck category of the functor $F : [\![\Delta]\!] \to \mathrm{Cats}$ with $F(\delta) \coloneqq \mathrm{Sizes}_{<[\![n]\!](\delta)}$. (Cats is the category of small categories.) Sizes are modelled as functors $[\![n]\!] : [\![\Delta]\!] \to \mathrm{Sizes}$ where Sizes is the category of sizes ordered by the preorder $\leq$; $\mathrm{Sizes}_{<n}$ is the subcategory of Sizes which contains only sizes less than $n$. The objects of $[\![\Delta]\!]$ are then telescopes of sizes, as before, and an arrow $f : \delta \to \delta'$ in $[\![\Delta]\!]$ is a proof that the sizes in $\delta$ are pointwise less than or equal to the sizes in $\delta'$. The judgment $\Delta \vdash n < m$ is modelled by a proof that $[\![n]\!](\delta) < [\![m]\!](\delta)$ for arbitrary $\delta : [\![\Delta]\!]$.

Continuing the categorical interpretation, we model types in a size context $\Delta$ as functors from $[\![\Delta]\!]$ to Sets, the category of sets (or rather, since we work in type theory, types). These are "covariant presheaves", meaning presheaves over the opposite category of $[\![\Delta]\!]$, $[\![\Delta]\!]^{\mathrm{op}}$.

One problem with this approach becomes apparent already at this stage: the type Stream $n$ of sized streams has no natural interpretation as a covariant functor from $[\![\Delta]\!]$ to Sets. We would like to define, as before,

$$[\![\mathrm{Stream}\ n]\!](\delta) \coloneqq \mathbb{N}_{<[\![n]\!](\delta)} \to \mathbb{N}$$

but this is no functor: for $\delta \leq \delta'$ we have $[\![n]\!](\delta) \leq [\![n]\!](\delta')$ by functoriality of $[\![n]\!]$, but there is no appropriate function from $\mathbb{N}_{<[\![n]\!](\delta)} \to \mathbb{N}$ to $\mathbb{N}_{<[\![n]\!](\delta')} \to \mathbb{N}$. This is not surprising – after all, $[\![\text{Stream } n]\!](\delta)$ is naturally contravariant in $\delta$.

Still, let us ignore this problem and move on to discover more fundamental issues. The other types of λST have more or less natural interpretations:

$$
\begin{aligned}
[\![\text{Nat } n]\!](\delta) &:= \mathbb{N}_{\leq [\![n]\!](\delta)} \\
[\![T \to U]\!](\delta) &:= \forall \delta' \geq \delta.\ [\![T]\!](\delta') \to [\![U]\!](\delta') \\
[\![\forall n.\ T]\!](\delta) &:= \forall \delta' \geq \delta.\ \forall m < [\![n]\!](\delta').\ [\![T]\!](\delta', m).
\end{aligned}
$$

Sized natural numbers Nat $n$ are interpreted as before. The function space $T \to U$ is modelled by the exponential of presheaves, which ensures that we can also model abstractions and applications. The exponential is defined using a monotonisation "trick": the function space $[\![T]\!](\delta) \to [\![U]\!](\delta)$ would not be functorial due to the negative occurrence of $\delta$, but we can force functoriality by quantifying over $\delta' \geq \delta$. The same approach also allows us to interpret size quantification, where again the natural interpretation without monotonisation would not be functorial.

Unfortunately, while this monotonisation trick works for the exponential, it does not yield an appropriate model of size quantification. This is not very surprising: in the above interpretation of $\forall n.\ T$, the size that we introduce in the model, $m$, is not, in general, smaller than $[\![n]\!](\delta)$. The bound $m < [\![n]\!](\delta')$ does not actually restrict the domain of $m$ since $\delta' \geq \delta$ and thus $[\![n]\!](\delta') \geq [\![n]\!](\delta)$. The reader is invited to check that this prevents us from interpreting fix.

## 6.2 Contravariant Presheaf Model

Seeing that the major problem with the previous model was the need to force-monotonise the interpretation of size quantification, we now consider a model based on *contravariant* presheaves (i.e. just presheaves). Size quantification is naturally contravariant, so no trickery is necessary to interpret it. Indeed, this approach leads to a more satisfactory interpretation of types (sizes, size contexts etc. are modelled as before):

$$
\begin{aligned}
[\![\text{Stream } n]\!](\delta) &:= \mathbb{N}_{\leq [\![n]\!](\delta)} \to \mathbb{N} \\
[\![T \to U]\!](\delta) &:= \forall \delta' \leq \delta.\ [\![T]\!](\delta') \to [\![U]\!](\delta') \\
[\![\forall n.\ T]\!](\delta) &:= \forall m < [\![n]\!](\delta).\ [\![T]\!](\delta, m).
\end{aligned}
$$

With the switch to contravariant functors, we lose the ability to model sized natural numbers, but gain the ability to model sized streams. As before, the function space is interpreted as an exponential of presheaves. Size quantification does not require monotonisation any more, and indeed this approach appears, on the surface, to support natural interpretations of all terms including fix.[6]

[6]The above interpretation of size quantification is not quite accurate: to successfully model λST, we would need to restrict it to allow only size-parametric functions, as in the reflexive graph model. However, this is unimportant for the rest of the discussion.

Alas, the contravariant presheaf approach still fails, this time due to a more subtle problem: the substitution lemma for size substitution in types does not hold. This means that in general we have

$$[\![T[\sigma]]\!] \neq [\![T]\!]\left[[\![\sigma]\!]\right].$$

To see why, we first consider the interpretation of substitutions. Recall that in the reflexive graph model, a well-typed substitution $\sigma : \Delta \Rightarrow \Omega$ was interpreted as a function between the types $[\![\Delta]\!]$ and $[\![\Omega]\!]$. Here, we upgrade this interpretation to a functor between the categories $[\![\Delta]\!]$ and $[\![\Omega]\!]$, but the underlying function remains the same. Semantic substitution is composition: given a type $T$ in size context $\Omega$, whose interpretation is a functor from $[\![\Omega]\!]$ to Sizes, we define

$$[\![T]\!]\left[[\![\sigma]\!]\right] \coloneqq [\![T]\!] \circ [\![\sigma]\!].$$

Now consider the substitution which assigns to the zeroth variable (in an otherwise empty context) the size 1:

$$\sigma \coloneqq \mathrm{Sing}(\uparrow 0) : () \to (), \infty.$$

Its interpretation is $[\![\sigma]\!](()) = ((), 1)$, where $()$ is the sole inhabitant of the unit type. Further, let $T$ be the following type (in the context $(), \infty$) of functions from streams to an arbitrary closed type $U$:

$$T \coloneqq \mathrm{Stream}\ \upsilon_0 \to U.$$

Then we have

$$\begin{aligned}
[\![T[\sigma]]\!](()) &= [\![\mathrm{Stream}\ 1 \to U]\!](()) \\
&= \forall() \leq (). \ [\![\mathrm{Stream}\ 1]\!](()) \to [\![U]\!](()) \\
&= \forall() \leq (). \ (\forall k \leq 1. \ \mathbb{N}) \to [\![U]\!](()) \\
&\cong (\forall k \leq 1. \ \mathbb{N}) \to [\![U]\!](()) \\
[\![T]\!]\left[[\![\sigma]\!]\right](()) &= [\![T]\!]([\![\sigma]\!](())) \\
&= [\![\mathrm{Stream}\ \upsilon_0 \to U]\!]((), 1) \\
&= \forall m \leq 1. \ [\![\mathrm{Stream}\ \upsilon_0]\!]((), m) \to [\![U]\!]((), m) \\
&= \forall m \leq 1. \ (\forall k \leq m. \ \mathbb{N}) \to [\![U]\!](()).
\end{aligned}$$

These two types are not isomorphic, so substitutions do not have the desired semantics.

# 7
# Conclusion

## 7.1 Related Work

**Sized Types** Sized types go back to Hughes, Pareto and Sabry [14], who coined the term and introduced the first calculus featuring sized types. Since then, a variety of calculi have been investigated, for example by Amadio and Coupet-Grimal [6]; Barthe, Frade, Giménez, Pinto and Uustalu [8]; Blanqui [9]; and Sacchini [23, 24]. These works all use slightly different notions of sizes on top of different base calculi (simply typed or dependently typed, with inductive or coinductive types, etc.), but none of them addresses the question of size irrelevance. Agda's particular notion of sized types, which λST emulates, is due to Abel and Pientka [4].

**Reflexive Graph Models** Reynolds [22] first introduced the technique of interpreting a type as a set together with a relation on that set. This is usually called a relationally parametric model. Atkey, Ghani and Johann [7] apply this technique to a dependent type theory, showing in particular how families of reflexive graphs can be used to interpret type dependencies. Vezzosi [30] gives a reflexive graph model of a type theory with guarded recursion, another type-based termination checking mechanism. Nuyts, Vezzosi and Devriese [20] present a dependent type theory with parametric quantifiers for which they also give a reflexive graph model. Their calculus allows users to reason internally about parametricity, which enables an encoding of sized types as types parametrically indexed by natural numbers (though they do not discuss the special size $\infty$).

## 7.2 Future Work

This thesis is but a first step towards a realistic account of Agda's sized types. In this section, I discuss some possible directions for future work (besides the obvious extensions of λST with dependent types, data types and so forth).

**Normalisation** This thesis is only concerned with one of the two interesting properties of λST, size irrelevance. The other is normalisation, which is arguably more important but also much better covered in the literature. In the case of λST, one would have to be careful with the definition of normalisation since its streams are infinite by design.

**Syntactic Size Irrelevance** As mentioned in Sec. 4.7, this thesis does not, in fact, prove any interesting properties of λST – only of its model. To connect the model with a syntactic

notion of size irrelevance, one might proceed as follows:

1. Give an operational semantics for λST.
2. Prove that if two terms are β-equal, their models are equal. (This is almost trivial informally but difficult to formalise.)
3. Give a notion of syntactic equality up to sizes.
4. Prove a lemma that allows us to conclude syntactic equality up to sizes from semantic equality.

Steps 1–3 are not hard, at least conceptually. For step 4, it is not clear to me exactly how to connect the model back to the syntax. One candidate is the following conjecture: for $u : \mathrm{Nat}\ m$ and $u' : \mathrm{Nat}\ m'$ (in an empty context), if $u$ and $u'$ are normal and $[\![u]\!] = [\![u']\!]$, then $u$ and $u'$ are syntactically equal up to sizes. This, together with the conjecture that β-equality implies equality in the model, would give us a weak form of size irrelevance restricted to base types and normal forms: if we have $t : \forall n.\ \mathrm{Nat}\ \upsilon_0$ and $t\ m$ normalises to $u$ and $t\ m'$ normalises to $u'$, then $u$ and $u'$ are equal up to sizes.

**Infinitely Branching Data Types** Natural numbers and streams, viewed as constructor trees, are finitely branching: each constructor has a finite number of subterms (one or zero in the case of natural numbers; one in the case of streams). However, Agda and other dependently typed languages also feature infinitely branching types such as this one:

```
data ℕTree (A : Set) : Set where
  leaf : A → ℕTree A
  node : (ℕ → ℕTree A) → ℕTree A
```

`ℕTree A` is a type of trees similar to rose trees except that each `node` has countably infinite children.

Such infinitely branching types complicate the interpretation of sizes. Recall that we interpreted sizes as the height of a constructor tree. For finitely branching types, this height is always a (computable) natural number. For infinitely branching types such as `ℕTree`, however, it is an ordinal: the height of a `node` is one plus the maximum height of its infinite collection of child trees. This means that in the model, $\infty$ could no longer be interpreted as $\omega$; instead it would have to be an ordinal that is greater than the height of any possible value of an infinitely branching data type.

Ordinals present some technical challenges since they are not trivially encoded in type theory. As part of the work for this thesis, I investigated a particular subclass of ordinals, the plump ordinals [26] as presented by Shulman [25], as candidates for a model of sizes. A formalisation of these ordinals can be found in Ordinal.Shulman. However, the investigation was inconclusive: while the formalisation shows that the plump ordinals have most of the properties we expect of a model of sizes, they do not admit the seemingly straightforward lemma that for ordinals $n < m$ and $n' < m$, the supremum $n \sqcup n'$ is also less than $m$. More precisely, this lemma is equivalent to classical logic – but then again, it is not clear that we need it to interpret a hypothetical extension of λST with infinitely branching types.

## 7.3 Conclusion

I have presented a calculus, λST, which approximates Agda's sized types in a simply-typed setting. This calculus has a reflexive graph model which gives a size-irrelevant denotational semantics. Both λST and the model are fully formalised.

# Bibliography


[1] Andreas Abel. Check needed when $\infty < \infty$ is ok for sizes. `https://github.com/agda/agda/issues/1201`, August 2015. Accessed: 2019-01-23.

[2] Andreas Abel. Sized types allow a type which is both inductive and coinductive in an inconsistent way. `https://github.com/agda/agda/issues/1946`, April 2016. Accessed: 2019-11-21.

[3] Andreas Abel. Equality is incompatible with sized types. `https://github.com/agda/agda/issues/2820`, October 2017. Accessed: 2019-11-21.

[4] Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26:e2, 2016.

[5] Agda contributors. Agda manual. `https://agda.readthedocs.io/en/v2.6.0.1/index.html`. Accessed: 2019-11-21.

[6] Roberto M. Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory. In *Proceedings of the International Conference on Foundations of Software Science and Computation Structure*, 1998.

[7] Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2014.

[8] Gilles Barthe, Maria João Frade, Eduardo Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[9] Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications*, 2004.

[10] Peter Dybjer. Internal type theory. In *Proceedings of the International Workshop on Types for Proofs and Programs*, 1995.

[11] Maxime Dénès. Propositional extensionality is inconsistent in Coq. `https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html`. Accessed: 2018-12-05.

[12] Maxime Dénès. Re: [HoTT] newbie questions about homotopy theory & advantage of UF/Coq. https://sympa.inria.fr/sympa/arc/coq-club/2013-12/msg00119.html. Accessed: 2018-12-05.

[13] Martín Escardó. Introduction to univalent foundations of mathematics with Agda. https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/HoTT-UF-Agda.html. Accessed: 2019-11-21.

[14] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 1996.

[15] Chris Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of univalent foundations (after Voevodsky). *arXiv preprint arXiv:1211.2851*, 2012.

[16] Jannis Limperg. A reflexive graph model of sized types (source code). https://doi.org/10.5281/zenodo.3572925, December 2019. Accessed: 2019-12-13.

[17] Zhaohui Luo. A unifying theory of dependent types: the schematic approach. In *Proceedings of the International Symposium on Logical Foundations of Computer Science*, 1992.

[18] Bengt Nordström. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619, 1988.

[19] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

[20] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2017.

[21] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2(4):325 – 355, 1986.

[22] John C Reynolds. Types, abstraction and parametric polymorphism. In *Proceedings of the IFIP congress*, 1983.

[23] Jorge Luis Sacchini. *On type-based termination and dependent pattern matching in the Calculus of Inductive Constructions*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2011.

[24] Jorge Luis Sacchini. Type-based productivity of stream definitions in the Calculus of Constructions. In *Proceedings of the Symposium on Logic in Computer Science (LICS)*, 2013.

[25] Mike Shulman. The surreals contain the plump ordinals. https://homotopytypetheory.org/2014/02/22/surreals-plump-ordinals/, February 2014. Accessed: 2019-03-31.

[26] Paul Taylor. Intuitionistic sets and ordinals. *Journal of Symbolic Logic*, 61:705–744, 1996.

[27] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[28] Twan van Laarhoven. Well-founded induction on Size is inconsistent. `https://github.com/agda/agda/issues/3026`, April 2018. Accessed: 2019-11-21.

[29] Niccolò Veltri and Niels van der Weide. Guarded recursion in Agda via sized types. In *Proceedings of the International Conference on Formal Structures for Computation and Deduction (FSCD)*, 2019.

[30] Andrea Vezzosi. Guarded recursive types in type theory. Licenciate thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2015. Technical Report 144L.

[31] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: a dependently typed programming language with univalence and higher inductive types. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2019.