Second-order generalised algebraic theories, by examples (under construction)

Ambrus Kaposi

December 23, 2025

In these notes, we study non-substructural programming languages at a high level of abstraction: a language is a second-order generalised algebraic theory (SOGAT). The syntax of such a language is its initial model, in which there are only well-typed (intrinsic) terms quotiented by conversion, every operation is automatically a congruence with respect to conversion and every operation is stable under substitution. These notes are pedagogical excerpts of the papers [KX24, BKS23]. A more categorical treatment can be found in the upcoming PhD thesis of Rafaël Bocquet.

Related ideas are higher-order abstract syntax [Hof99], logical frameworks [HHP93], two-level type theories [ACKS23], synthethic Tait computability [Ste22]. We will rely on being able to work informally in type theory (informal Agda, Coq, Idris or Lean).

These notes can be formalised in the following metatheories: observational type theory, extensional type theory with quotient inductive-inductive types and propositional extensionality, homotopy type theory, constructive set theory.

These notes were originally written for the International School on Logical Frameworks and Proof Systems Interoperability (LFPSI) which was in Orsay between 8–11 September 2025.

Thanks to Botond Mészáros and Vojtěch Štěpančík for fixing typos.

Sections compulsory for the type systems course at ELTE: 1, 2.3, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7.2, 3.8.

Contents

1 Levels of abstraction

A language can be described in different ways ranging from concrete to abstract, see Figure 1. In this section, we explain the different levels briefly via a simple expression language (Razor, see Subsection 2.3). The following example program can be written in Razor.

```
if isZero (num 0 + num 1) then false else isZero (num 0)
```

A Razor program is either a numeric or a boolean expression. Numbers can be formed using `num i` where `i` is a natural number. Booleans are `true` or `false`. We have the usual if-then-else operator, addition and an isZero operator which says whether a number is 0. The above program evaluates (runs) in the following steps (each new line is a step).

```
if isZero (num 0 + num 1) then false else isZero (num 0)
if isZero (num 1) then false else isZero (num 0)
if false then false else isZero (num 0)
isZero (num 0)
true
```

Figure 2 gives complete descriptions of Razor at levels of abstraction (1)–(5):

(1) As a first approximation, a program is a string, that is, a sequence of (ASCII) characters. This is how we write programs on a computer. Any string is a program. Many strings do not correspond to meaningful programs in our language such as `num 3 - num 2` as we don't have subtraction. Also, there are different strings which represent the same program. For example, `isZero  (num 1)` and `isZero (num 1)` are different as strings but should be the same programs as the extra spaces after `isZero` shouldn't matter. Instead of describing which strings are meaningful programs and defining an equivalence relation for identifying strings that represent the same program, we will describe programs using a more abstract structure.

(2) The more abstract structure is list of lexical elements. Now we have much fewer programs and `num 3 - num 2` is not a program anymore because there is no lexical element for `-`. Any two programs given as strings which differ only in the number of spaces will end up as the same program at this level: `isZero  (num 1)` and `isZero (num 1)` are both given by the sequence [`isZero`, `(`, `num`, `1`, `)`]. However, we still have meaningless programs, e.g. [`(`, `true`] (there is no closing parenthesis) or [`num`, `1`, `+`] (`+` needs two arguments), and so on. Also, there are programs which could be identified, e.g. [`(`, `true`, `)`] and [`true`] (the parentheses are redundant in the former). Again, to solve these issues, we move to a higher-level representation of programs. Note that there are standard ways to navigate between levels (1) and (2): (2) to (1) is printing. (1) to (2) is performed by a lexical analyser (lexer) which turns a string into a sequence of lexical elements or returns an error.

(3) Abstract syntax tree descriptions are usually given by BNF grammars. At this level, we only have well-parenthesised expressions and each operator receives the correct number of arguments. Programs are now trees which have `true`, `false` or `num i` at their leaves and they can have ternary branching with `ite` at the branching node, unary branching with `isZero` at the node or binary branching with `+` at the node.

(4) In well-typed (intrinsic) syntax trees, the types of the arguments are restricted to the correct ones.

(5) In well-typed quotiented syntax, two programs which have the same result are identified.

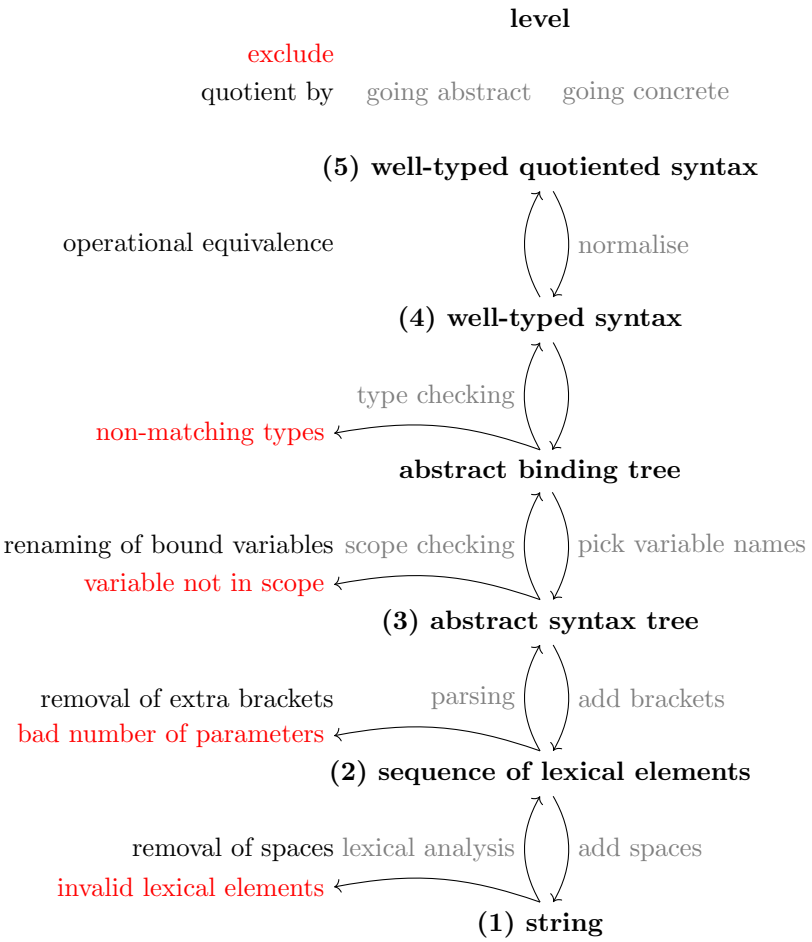


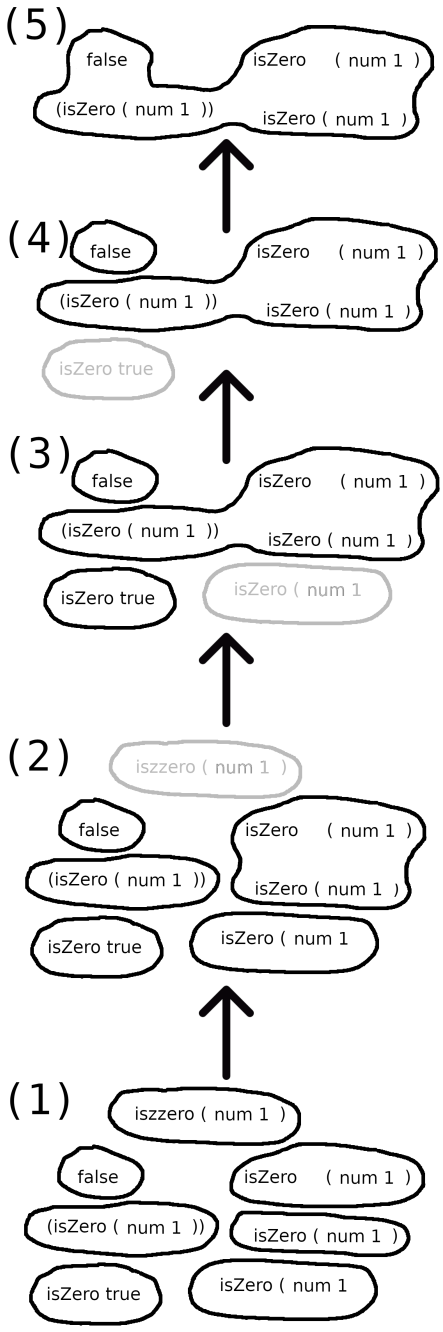
Figure 1: Different levels of abstraction when defining a programming language and transformations between levels. At more abstract levels, certain programs are excluded and others identified. Abstract binding trees are sometimes called well-scoped syntax trees.

Abstract binding trees are not relevant for Razor as there are no variables.

The *extrinsic* approach moves to more abstract levels by defining relations which select the well-formed (well-scoped, well-typed, etc) expressions in the concrete representation. In contrast, we use the *intrinsic* approach where in the more abstract representation the non-well-formed expressions are not even expressible. At level (5), there is no need to prove that conversion preserves typing ("preservation" from "progress and preservation") because the equations are already expressed in a typed way. Certain properties of the language cannot be expressed at the abstract levels. For example it is not possible to count the number of brackets in a program at level (3). At level (5), it is not possible to reason about program efficiency because all convertible programs are in the same equivalence class: we cannot write a function which distinguishes convertible programs. We view the lower levels as important parts of a programming language, but we see them as intermediate technical steps to reach the most abstract level which describes the essence of the language.

In these notes, we will use the most abstract level to describe languages.

There are several features of programming languages that we do not know how to define intrinsically, and these notes only cover a very small fragment of the theory of programming languages. We direct the curious reader to standard textbooks [Pie02, PdAC+23, Har16, WKS22, Csö12, DL11].



(5) well-typed quotiented syntax

```
Ty        :  Set
Tm        :  Ty → Set
Bool      :  Ty
Nat       :  Ty
true      :  Tm Bool
false     :  Tm Bool
ite       :  Tm Bool → Tm A → Tm A → Tm A
num       :  ℕ → Tm Nat
isZero    :  Tm Nat → Tm Bool
_+_       :  Tm Nat → Tm Nat → Tm Nat
iteβ1     :  ite true u v = u
iteβ2     :  ite false u v = v
isZeroβ1  :  isZero (num 0) = true
isZeroβ2  :  isZero (num (1+n)) = false
+β        :  num m + num n = num (m + n)
```

(4) well-typed syntax

```
Ty        :  Set
Tm        :  Ty → Set
Bool      :  Ty
Nat       :  Ty
true      :  Tm Bool
false     :  Tm Bool
ite       :  Tm Bool → Tm A → Tm A → Tm A
num       :  ℕ → Tm Nat
isZero    :  Tm Nat → Tm Bool
_+_       :  Tm Nat → Tm Nat → Tm Nat
```

(3) abstract syntax tree

```
Tm        :  Set
true      :  Tm
false     :  Tm
ite       :  Tm → Tm → Tm → Tm
num       :  ℕ → Tm
isZero    :  Tm → Tm
_+_       :  Tm → Tm → Tm
```

(2) list of the following lexical elements:

`(`, `)`, `true`, `false`, `if`, `then`, `else`, `num`, `isZero`, `+`, `0`, `1`, `2`, `3`, ...

(1) any string

Figure 2: Left: example Razor programs at different levels of abstraction. Each bubble represents a separate program. Right: description of the Razor expression language at levels (1)–(5).

2 Derivability and admissibility in GATs

As a warmup, we review how languages without binders can be seen as generalised algebraic theories (GATs). The goal of this section is to familiarise the reader with the following concepts: model, derivability, morphism, dependent model, dependent morphism, syntax, induction, iteration, admissibility, logical consistency, equational consistency, normal forms, normalisation. These concepts are specific to the particular GAT, and we show what they are for different example GATs. By the end of this section, the reader should be able to formulate them for any GAT (except normal forms which only exist for certain languages).

2.1 Monoid

We start with a well-known algebraic theory (AT): monoids. A model of the theory of monoid is also called a monoid algebra or simply monoid.

Definition 1 (Monoid)**.** *A monoid model comprises the following components:*

$$\begin{aligned}
&\mathsf{C} && : \mathsf{Set}\\
&- \cdot - && : \mathsf{C} \to \mathsf{C} \to \mathsf{C}\\
&\mathsf{ass} && : x \cdot (y \cdot z) = (x \cdot y) \cdot z\\
&\mathsf{u} && : \mathsf{C}\\
&\mathsf{idl} && : \mathsf{u} \cdot x = x\\
&\mathsf{idr} && : x \cdot \mathsf{u} = x
\end{aligned}$$

A model contains one carrier set (sort), two operations (one binary and one nullary) which satisfy three equations.

Example models (the equations also hold, but we don't write their proofs):

$\mathsf{C} := \mathbb{N}$	$\mathsf{C} := \mathbb{N}$	$\mathsf{C} := \{*\}$	$\mathsf{C} := \{\mathsf{tt}, \mathsf{ff}\}$
$x \cdot y := x + y$	$x \cdot y := x * y$	$x \cdot y := *$	$x \cdot y := x \wedge y$
$\mathsf{u} := 0$	$\mathsf{u} := 1$	$\mathsf{u} := *$	$\mathsf{u} := \mathsf{tt}$

Exercise 2. *Prove the equations for the above four monoids. Define all the monoids with sort* $\{\mathsf{tt}, \mathsf{ff}\}$*. Define the following monoids: strings with concatenation, square matrices with multiplication,* $A \to A$ *functions with composition, subsets of* A *with intersection/union.*

Non-examples: C is the empty set, $\mathsf{C} = \mathbb{N}$ with exponentiation, $\mathsf{C} = \mathbb{Z}$ with subtraction. We give names to the models and refer to their components via subscript. E.g. if we call the above first model M, then $\mathsf{C}_M = \mathbb{N}$, $x \cdot_M y = x + y$ and $\mathsf{u}_M = 0$.

A *derivable operation* is one that is defined for any model, for example $\mathsf{dup}\,(x : \mathsf{C}) : \mathsf{C} := x \cdot x$. A *derivable equation* is one that holds in any model, for example $(\mathsf{u} \cdot \mathsf{u}) \cdot \mathsf{u} = \mathsf{u}$ which is derived by

$$(\mathsf{u} \cdot \mathsf{u}) \cdot \mathsf{u} \overset{\mathsf{idr}}{=} \mathsf{u} \cdot \mathsf{u} \overset{\mathsf{idl}}{=} \mathsf{u}.$$

In a proof assistant, we can define derivable operations and equations by assuming a model as a module parameter / postulate / axiom / variable, and only using this when defining the operation or proving the equation. Then we can specialise the derivable things to particular models, for the above M, we have $\mathsf{dup}_M\, 3 = 3 \cdot_M 3 = 3 + 3 = 6$ and $(\mathsf{u}_M \cdot_M \mathsf{u}_M) \cdot_M \mathsf{u}_M = (0 + 0) + 0 = 0 + 0 = 0 = \mathsf{u}_M$.

A *morphism* (or homomorphism) between models is a function between the carriers that preserves the operations, precisely a morphism from M to N comprises the following components:

$$\begin{aligned}
&\mathsf{C} && : \mathsf{C}_M \to \mathsf{C}_N\\
&- \cdot - && : (x\, y : \mathsf{C}_M) \to \mathsf{C}\,(x \cdot_M y) = \mathsf{C}\,x \cdot_N \mathsf{C}\,y\\
&\mathsf{u} && : \mathsf{C}\,\mathsf{u}_M = \mathsf{u}_N
\end{aligned}$$

Note that there are no components corresponding to the equations (this only changes when the GAT has sort equations).

Exercise 3. *Define all the morphisms between any pair of models from the above four examples.*

A *dependent model* (displayed model, motive and methods of the induction principle) over a model M has the same number of components as a model, and is dependent over them, that is:

$$\begin{aligned}
&\mathsf{C} && : \mathsf{C}_M \to \mathsf{Set}\\
&- \cdot - && : \mathsf{C}\,x_M \to \mathsf{C}\,y_M \to \mathsf{C}\,(x_M \cdot_M y_M)\\
&\mathsf{ass} && : x \cdot (y \cdot z) = (x \cdot y) \cdot z\\
&\mathsf{u} && : \mathsf{C}\,\mathsf{u}_M\\
&\mathsf{idl} && : \mathsf{u} \cdot x = x\\
&\mathsf{idr} && : x \cdot \mathsf{u} = x
\end{aligned}$$

Here we used (the somewhat extreme) notation where metavariables have subscripts. $- \cdot -$ has two implicit arguments x_M and y_M, both in C_M (the equations ass, idl, idr in the notion of model also had implicit arguments). Note that ass also depends on ass_M: we have $x : \mathsf{C}\,x_M$, $y : \mathsf{C}\,y_M$, $z : \mathsf{C}\,z_M$ and the left hand side $x \cdot (y \cdot z)$ is in $\mathsf{C}\,(x_M \cdot_M (y_M \cdot_M z_M))$, the right hand side is in $\mathsf{C}\,((x_M \cdot_M y_M) \cdot_M z_M)$. These sets are equal by ass_M. The situation is similar for idl, idr. Examples where $M = (\mathbb{N}, +, 0)$:

$$\begin{aligned}
\mathsf{C}\,n &:= \mathsf{Vec}\,\mathbb{N}\,n & \mathsf{C}\,_ &:= \{*\}\\
x \cdot y &:= x +\!\!+ y & x \cdot y &:= *\\
\mathsf{u} &:= [] & \mathsf{u} &:= *
\end{aligned}$$

Exercise 4. *Any model can be turned into a dependent model where we ignore the dependency.*

Exercise 5. *Any dependent model D over M can be turned into a model together with a morphism into M. The carrier will be $(x_M : \mathsf{C}_M) \times \mathsf{C}_D\,x_M$ (a dependent Descartes-product, or Σ-type in the metatheory).*

A *dependent morphism* (section) from a model M to a dependent model D over M is like a homomorphism, but the function is dependent:

$$\begin{aligned}
&\mathsf{C} && : (x : \mathsf{C}_M) \to \mathsf{C}_D\,x\\
&- \cdot - && : (x\,y : \mathsf{C}_M) \to \mathsf{C}\,(x \cdot_M y) = \mathsf{C}\,x \cdot_D \mathsf{C}\,y\\
&\mathsf{u} && : \mathsf{C}\,\mathsf{u}_M = \mathsf{u}_D
\end{aligned}$$

The *syntax* is a model from which there is a dependent morphism into any dependent model (the dependent model has to be over the syntax for this to make sense). We denote the syntax by I (for initial model). The function which takes a dependent model over the syntax and returns the dependent morphism is called *induction* (also called (dependent) eliminator, universal property).

Exercise 6. *Show that there is a syntax for monoids (hint: the carrier is a particularly simple set).*

Exercise 7. *Show that that for a given model M, the following two are equivalent:*

- *there is a dependent morphism into any model over M,*
- *there is a unique homomorphism from M into any model (initiality).*

Dependent models and morphisms were introduced in order to specify syntax and induction. Special cases of induction are iteration (fold, catamorphism, non-dependent eliminator, interpreter) and recursion (sometimes also called non-dependent eliminator). The syntax has iteration, which means that for any model M, there is a morphism from I to M. Recursion is the special case of induction where the $\mathsf{C}_\mathsf{I} \to \mathsf{Set}$ component in the dependent model is a constant function.

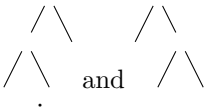
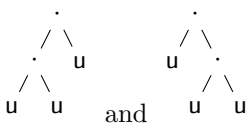
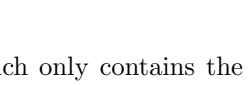
Exercise 8. *There is an identity morphism from any model to itself. Morphisms can be composed. An isomorphism between models M and N (denoted $M \cong N$) comprises morphisms $M \to N$ and $N \to M$ such that there composites are the identity morphisms. Show that any two syntaxes are isomorphic.*

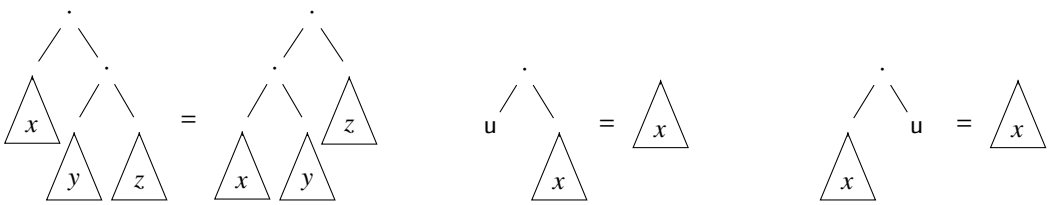
An *admissible* operation / equation is one that can be defined / proven for the syntax via induction. For the syntax of monoids, we prove that for any $x : \mathsf{C}_\mathsf{I}$ (note that I is the syntax), $x = \mathsf{u}_\mathsf{I}$ by defining the following dependent model over I:

$$\begin{aligned}
&\mathsf{C}\,x && := (x = \mathsf{u}_\mathsf{I})\\
&(e : x = \mathsf{u}_\mathsf{I}) \cdot (e' : x' = \mathsf{u}_\mathsf{I}) && : x \cdot_\mathsf{I} x' \overset{e}{=} \mathsf{u}_\mathsf{I} \cdot_\mathsf{I} x' \overset{e'}{=} \mathsf{u}_\mathsf{I} \cdot_\mathsf{I} \mathsf{u}_\mathsf{I} \overset{\mathsf{idl}_\mathsf{I}}{=} \mathsf{u}_\mathsf{I}\\
&\mathsf{u} && : \mathsf{u}_\mathsf{I} = \mathsf{u}_\mathsf{I}
\end{aligned}$$

Via induction, we obtain a function $(x : \mathsf{C}_\mathsf{I}) \to \mathsf{C}\,x$ which is the same as $(x : \mathsf{C}_\mathsf{I}) \to x = \mathsf{u}_\mathsf{I}$. We say that $x = \mathsf{u}$ is an admissible equation, but it is not derivable.

Exercise 9 (From [Moe22]). *Show that the inverse operation is admissible for monoids by defining a dependent model where* $\mathsf{C}\,x = (x^{-1} : \mathsf{C}) \times (x \cdot x^{-1} = \mathsf{u}) \times (x^{-1} \cdot x = \mathsf{u})$.

For monoids we were able to define the syntax in an ad-hoc way (Exercise 6), but there is a generic way to construct the syntax which works for any algebraic theory. For monoids, we first try to define $\mathsf{C_I}$ as the set of binary trees where the binary branching nodes denote $\cdot_\mathsf{I}$ and the leaves denote u. The trees  and  denote $(\mathsf{u} \cdot \mathsf{u}) \cdot \mathsf{u}$ and $\mathsf{u} \cdot (\mathsf{u} \cdot \mathsf{u})$, respectively. Actually, we like to draw them as  and to make the connection between the operations and the nodes / leaves of the tree explicit. This definition of $\mathsf{C_I}$ however does not suffice. We cannot prove e.g. that the tree  and the tree which only contains the leaf u are equal, so we cannot provide the component $\mathsf{idl_I}$. We solve this by *quotienting* the set of binary trees by the three equations ass, idl and idr, that is, the trees which have the following shapes will be identified (where a triangle denotes any tree):



We call these quotiented trees *syntax trees* (we could call them quotiented syntax trees – however we can consider the language of monoids without equations, and the syntax for that language provides the unquotiented monoid syntax trees). One can prove that the model defined like this is a syntax, that is, it has induction. However, we don't do this, we simply assume that there is a syntax (which just means a model with induction). For a generic construction of syntaxes, see e.g. [KKA19]. In a type theory with support for quotient inductive sets (quotient inductive types), one can define the syntax of monoids as the quotient inductive set with two point-constructors $(- \cdot -, \mathsf{u})$ and three equality (path) constructors (ass, idl and idr). The dependent eliminator for this quotient inductive set exactly says that this model has induction.

2.2 Pointed set with endofunction

This language is particularly simple, it does not have equations.

Definition 10 (PSE). *A model comprises the following components:*

$$\begin{aligned}\mathsf{N} &: \mathsf{Set}\\ \mathsf{z} &: \mathsf{N}\\ \mathsf{s} &: \mathsf{N} \to \mathsf{N}\end{aligned}$$

A derivable function is e.g. $\mathsf{sss}\,(n : \mathsf{N}) : \mathsf{N} := \mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,n))$. There are no interesting derivable equations.

The syntax is $(\mathbb{N}, 0, +1)$, a dependent model over this contains $\mathsf{N} : \mathbb{N} \to \mathsf{Set}$, $\mathsf{z} : \mathsf{N}\,0$ and $\mathsf{s} : \mathsf{N}\,n \to \mathsf{N}\,(1 + n)$. Induction for $(\mathbb{N}, 0, +1)$ thus says the usual notion of induction: given a (proof-relevant) predicate which holds for 0 and which preserves successor, the predicate holds for all natural numbers. Hence I is the natural numbers. The fact that natural numbers form an exponential semiring is admissible for the language PSE:

Exercise 11. *Define the admissible operations of addition, multiplication, exponentiation. Prove the admissible equations associativity of addition, left and right identity, commutativity, etc.*

Exercise 12. *Show via induction that* $0 \neq 1 + n$ *and that* $+1$ *is injective.*

When we draw syntax trees, they are just unary branching, like $\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{z}))$:

```
s
|
s
|
s
|
z
```

Exercise 13. *Show that induction is equivalent to initiality.*

2.3 An expression language

The following is a simple expression language which is not algebraic, but *generalised* algebraic. There are two sorts, and the second one is indexed over the first one.[1] We call the language Razor following [Hut23].

Definition 14 (Razor)**.** *A model comprises the following components:*

$$
\begin{array}{ll}
\mathsf{Ty} & : \mathsf{Set} \\
\mathsf{Tm} & : \mathsf{Ty} \to \mathsf{Set} \\
\mathsf{Bool} & : \mathsf{Ty} \\
\mathsf{Nat} & : \mathsf{Ty} \\
\mathsf{true} & : \mathsf{Tm}\,\mathsf{Bool} \\
\mathsf{false} & : \mathsf{Tm}\,\mathsf{Bool} \\
\mathsf{ite} & : \mathsf{Tm}\,\mathsf{Bool} \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A \\
\mathsf{num} & : \mathbb{N} \to \mathsf{Tm}\,\mathsf{Nat} \\
- + - & : \mathsf{Tm}\,\mathsf{Nat} \to \mathsf{Tm}\,\mathsf{Nat} \to \mathsf{Tm}\,\mathsf{Nat} \\
\mathsf{isZero} & : \mathsf{Tm}\,\mathsf{Nat} \to \mathsf{Tm}\,\mathsf{Bool} \\
\mathsf{ite}\beta_1 & : \mathsf{ite}\,\mathsf{true}\,u\,v = u \\
\mathsf{ite}\beta_2 & : \mathsf{ite}\,\mathsf{false}\,u\,v = v \\
+\beta & : \mathsf{num}\,m + \mathsf{num}\,n = \mathsf{num}\,(m+n) \\
\mathsf{isZero}\beta_1 & : \mathsf{isZero}\,(\mathsf{num}\,0) = \mathsf{true} \\
\mathsf{isZero}\beta_2 & : \mathsf{isZero}\,(\mathsf{num}\,(1+n)) = \mathsf{false}
\end{array}
$$

We call elements of Ty types and elements of $\mathsf{Tm}\,A$ terms of type A. There is no single set of terms, instead for each type, there is a separate set of terms of that type. Each operation is *typed*: their input types and output type is restricted. The operation ite has one implicit type argument A and its first explicit argument needs to have type Bool, while the second and third explicit arguments need to have (the same) type A.

The derivable operations can be seen as programs that can be defined in this language, e.g. $\mathsf{not}\,(b : \mathsf{Tm}\,\mathsf{Bool}) : \mathsf{Tm}\,\mathsf{Bool} := \mathsf{ite}\,b\,\mathsf{false}\,\mathsf{true}$. The equations explain how to run the programs, e.g. $\mathsf{not}\,\mathsf{true} = \mathsf{ite}\,\mathsf{true}\,\mathsf{false}\,\mathsf{true} \overset{\mathsf{ite}\beta_1}{=} \mathsf{false}$. Another example:

$$
\begin{array}{ll}
(\mathsf{num}\,1 + \mathsf{num}\,2) + (\mathsf{num}\,3 + \mathsf{num}\,4) & =(+\beta) \\
(\mathsf{num}\,(1+2)) + (\mathsf{num}\,3 + \mathsf{num}\,4) & = \\
\mathsf{num}\,3 + (\mathsf{num}\,3 + \mathsf{num}\,4) & =(+\beta) \\
\mathsf{num}\,3 + \mathsf{num}\,(3+4) & = \\
\mathsf{num}\,3 + \mathsf{num}\,7 & =(+\beta) \\
\mathsf{num}\,(3+7) & = \\
\mathsf{num}\,10 &
\end{array}
$$

Note that in the expression $\mathsf{num}\,(1+2)$, the $+$ refers to the metatheoretic addition, while in $\mathsf{num}\,1 + \mathsf{num}\,2$ the $+$ refers to object theoretic addition.

The standard (metacircular, set) model of Razor is where types are sets (metatheoretic types) and terms

[1] Certain such GATs can be reduced to ATs by replacing the Ty-indexing with a function from Tm to Ty. For Razor, this is not the case because after such a reduction, the ite operation becomes partial. The resulting theory is called an essentially algebraic theory (EAT). EATs are another extension of algebraic theories. GATs and EATs are equivalent in the sense that for each GAT, there is an EAT with an equivalent category of models, and for each EAT, there is a GAT with an equivalent category of models.

are elements of the sets (metatheoretic terms):

$$\begin{aligned}
\mathsf{Ty} &:= \mathsf{Set}\\
\mathsf{Tm}\,A &:= A\\
\mathsf{Bool} &:= \{\mathsf{tt}, \mathsf{ff}\}\\
\mathsf{Nat} &: \mathbb{N}\\
\mathsf{true} &:= \mathsf{tt}\\
\mathsf{false} &:= \mathsf{ff}\\
\mathsf{ite}\,b\,t\,f &:= \mathsf{match}\,b\,\{\mathsf{tt} \mapsto t; \mathsf{ff} \mapsto f\}\\
\mathsf{num}\,n &:= n\\
u + v &:= u + v\\
\mathsf{isZero}\,u &:= \mathsf{match}\,u\,\{0 \mapsto \mathsf{tt}; (1+n) \mapsto \mathsf{ff}\}
\end{aligned}$$

We do not write metatheoretic universe levels, hence informally $\mathsf{Set} : \mathsf{Set}$, but formally types in the standard model are a large metatheoretic type (they are in Set_1). In the definition of ite and isZero we used pattern matching on elements of metatheoretic booleans $\{\mathsf{tt}, \mathsf{ff}\}$. In the standard model, all equations hold by reflexivity, as they hold definitionally in the metatheory.

Exercise 15. *If in any model* $\mathsf{true} = \mathsf{false}$ *then for any* A *and* $u, v : \mathsf{Tm}\,A$*, we have* $u = v$*.*

Exercise 16. *If in any model* $\mathsf{num}\,0 = \mathsf{num}\,1$ *then for any* A *and* $u, v : \mathsf{Tm}\,A$*, we have* $u = v$*.*

Exercise 17. *There is a model where* $\mathsf{num}\,1 = \mathsf{num}\,2$*, but* $\mathsf{true} \neq \mathsf{false}$*.*

Exercise 18. *There is a model where* $\mathsf{Tm}\,\mathsf{Bool}$ *has three elements (nonstandard model).*

Exercise 19. *Prove or disprove the following statements.*

- *There is a model where* $\mathsf{true} \neq \mathsf{false}$ *and for every* t *and* u*,* $\mathsf{ite}\,t\,u\,u = u$*.*
- *In every model for every* t *and* u*,* $\mathsf{ite}\,t\,u\,u = u$*.*
- *There is a model in which* $\mathsf{isZero}\,(\mathsf{num}\,0) = \mathsf{false}$*.*
- *In every model* $\mathsf{isZero}\,(\mathsf{num}\,3) = \mathsf{isZero}\,(\mathsf{num}\,5)$*.*
- *There is no model in which* $\mathsf{Tm}\,\mathsf{Bool} = \mathbb{N}$*.*

A morphism $M \to N$ contains two functions, one for types and one for terms. The latter refers to the former:

$$\begin{aligned}
\mathsf{Ty} &: \mathsf{Ty}_M \to \mathsf{Ty}_N\\
\mathsf{Tm} &: \mathsf{Tm}_M\,A_M \to \mathsf{Tm}_N\,(\mathsf{Ty}\,A_M)\\
\mathsf{Bool} &: \mathsf{Ty}\,\mathsf{Bool}_M = \mathsf{Bool}_N\\
\mathsf{Nat} &: \mathsf{Ty}\,\mathsf{Nat}_M = \mathsf{Nat}_N\\
\mathsf{true} &: \mathsf{Tm}\,\mathsf{true}_M = \mathsf{true}_N\\
\mathsf{false} &: \mathsf{Tm}\,\mathsf{false}_M = \mathsf{false}_N\\
\mathsf{ite} &: (b_M : \mathsf{Tm}_M\,\mathsf{Bool}_M)(t_M\,f_M : \mathsf{Tm}_M\,A) \to \mathsf{Tm}\,(\mathsf{ite}_M\,b_M\,t_M\,f_M) = \mathsf{ite}_N\,(\mathsf{Tm}\,b_M)\,(\mathsf{Tm}\,t_M)\,(\mathsf{Tm}\,f_M)\\
\mathsf{num} &: (n : \mathbb{N}) \to \mathsf{Tm}\,(\mathsf{num}_M\,n) = \mathsf{num}_N\,n\\
- + - &: (u_M\,v_M : \mathsf{Tm}_M\,\mathsf{Nat}_M) \to \mathsf{Tm}\,(u_M +_M v_M) = \mathsf{Tm}\,u_M +_N \mathsf{Tm}\,v_M\\
\mathsf{isZero} &: (u_M : \mathsf{Tm}_M\,\mathsf{Nat}_M) \to \mathsf{Tm}\,(\mathsf{isZero}_M\,u_M) = \mathsf{isZero}_N\,(\mathsf{Tm}\,u_M)
\end{aligned}$$

The equality true depends on the equality Bool as its left hand side is in $\mathsf{Tm}_N\,(\mathsf{Ty}\,\mathsf{Bool}_M)$, while the right hand side is in $\mathsf{Tm}_N\,\mathsf{Bool}_N$. The situation is similar for other term equations.

Theorem 20. *Razor is logically inconsistent, that is, every sort in the syntax has an element.*

Proof. Ty_I has an element Bool_I, $\mathsf{Tm}_\mathsf{I}\,\mathsf{Bool}_\mathsf{I}$ has an element true_I, $\mathsf{Tm}_\mathsf{I}\,Nat_\mathsf{I}$ has an element $\mathsf{num}_\mathsf{I}\,0$. □

Theorem 21. *Razor is equationally consistent, that is, not all syntactic terms are equal, that is, there is a type* A_I *and terms* $a, a' : \mathsf{Tm}\,A_\mathsf{I}$ *such that* $a \neq a'$*.*

Proof. We choose $A_\mathsf{I} := \mathsf{Bool}_\mathsf{I}$ and $a := \mathsf{true}_\mathsf{I}$ and $a' := \mathsf{false}_\mathsf{I}$, then assuming $a = a'$, their iterations into the standard model are also equal, hence $\mathsf{tt} = \mathsf{ff}$. □

Iteration into the standard model can be called normalisation (recall that this is a morphism from I to the standard model). Normal forms are given by the Ty component of iteration which we rename to Nf, and we omit the names of the equations; this looks like a pattern-matching definition:

$$\begin{aligned}
&\mathsf{Nf} : \mathsf{Ty}_\mathsf{I} \to \mathsf{Set}\\
&\mathsf{Nf}\,\mathsf{Bool}_\mathsf{I} = \{\mathsf{tt}, \mathsf{ff}\}\\
&\mathsf{Nf}\,\mathsf{Nat}_\mathsf{I} \;= \mathbb{N}
\end{aligned}$$

The Tm component of the iteration morphism gives the normalisation function, we rename it to norm, its computation rules are the components of the morphism for the term operators:

$$\begin{aligned}
&\mathsf{norm} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I} \to \mathsf{Nf}\, A_\mathsf{I}\\
&\mathsf{norm}\,\mathsf{true}_\mathsf{I} &&= \mathsf{tt}\\
&\mathsf{norm}\,\mathsf{false}_\mathsf{I} &&= \mathsf{ff}\\
&\mathsf{norm}\,(\mathsf{ite}_\mathsf{I}\, b_\mathsf{I}\, t_\mathsf{I}\, f_\mathsf{I}) &&= \mathsf{match}\,(\mathsf{norm}\, b_\mathsf{I})\,\{\mathsf{tt} \mapsto \mathsf{norm}\, t_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{norm}\, f_\mathsf{I}\}\\
&\mathsf{norm}\,(\mathsf{num}_\mathsf{I}\, n) &&= n\\
&\mathsf{norm}\,(u_\mathsf{I} +_\mathsf{I} v_\mathsf{I}) &&= \mathsf{norm}\, u_\mathsf{I} + \mathsf{norm}\, v_\mathsf{I}\\
&\mathsf{norm}\,(\mathsf{isZero}_\mathsf{I}\, u_\mathsf{I}) &&= \mathsf{match}\,(\mathsf{norm}\, u_\mathsf{I})\,\{0 \mapsto \mathsf{tt}; (1+n) \mapsto \mathsf{ff}\}
\end{aligned}$$

A dependent model over a model M comprises the following components:

$$\begin{aligned}
&\mathsf{Ty} &&: \mathsf{Ty}_M \to \mathsf{Set}\\
&\mathsf{Tm} &&: \{A_M : \mathsf{Ty}_M\} \to \mathsf{Ty}\, A_M \to \mathsf{Tm}_M\, A_M \to \mathsf{Set}\\
&\mathsf{Bool} &&: \mathsf{Ty}\,\mathsf{Bool}_M\\
&\mathsf{Nat} &&: \mathsf{Ty}\,\mathsf{Nat}_M\\
&\mathsf{true} &&: \mathsf{Tm}\,\{\mathsf{Bool}_M\}\,\mathsf{Bool}\,\mathsf{true}_M\\
&\mathsf{false} &&: \mathsf{Tm}\,\{\mathsf{Bool}_M\}\,\mathsf{Bool}\,\mathsf{false}_M\\
&\mathsf{ite} &&: \mathsf{Tm}\,\mathsf{Bool}\, b_M \to \mathsf{Tm}\, A\, t_M \to \mathsf{Tm}\, A\, f_M \to \mathsf{Tm}\, A\,(\mathsf{ite}_M\, b_M\, t_M\, f_M)\\
&\mathsf{num} &&: (n : \mathbb{N}) \to \mathsf{Tm}\,\mathsf{Nat}\,(\mathsf{num}_M\, n)\\
&- + - &&: \mathsf{Tm}\,\mathsf{Nat}\, u_M \to \mathsf{Tm}\,\mathsf{Nat}\, v_M \to \mathsf{Tm}\,\mathsf{Nat}\,(u_M +_M v_M)\\
&\mathsf{isZero} &&: \mathsf{Tm}\,\mathsf{Nat}\, u_M \to \mathsf{Tm}\,\mathsf{Bool}\,(\mathsf{isZero}_M\, u_M)\\
&\mathsf{ite}\beta_1 &&: \mathsf{ite}\,\mathsf{true}\, u\, v = u\\
&\mathsf{ite}\beta_2 &&: \mathsf{ite}\,\mathsf{false}\, u\, v = v\\
&+\beta &&: \mathsf{num}\, m + \mathsf{num}\, n = \mathsf{num}\,(m+n)\\
&\mathsf{isZero}\beta_1 &&: \mathsf{isZero}\,(\mathsf{num}\, 0) = \mathsf{true}\\
&\mathsf{isZero}\beta_2 &&: \mathsf{isZero}\,(\mathsf{num}\,(1+n)) = \mathsf{false}
\end{aligned}$$

Note that the equations in the dependent model depend on the corresponding equations in M. For example, the left hand side of $\mathsf{ite}\beta_1$ is in $\mathsf{Tm}\,\{A_M\}\,A\,(\mathsf{ite}_M\,\mathsf{true}_M\, u_M\, v_M)$, while the right hand side is in $\mathsf{Tm}\,\{A_M\}\,A\, u_M$.

The syntax supports induction, which means that there is the following dependent morphism from the syntax to a dependent model D over it:

$$\begin{aligned}
&\mathsf{Ty} &&: (A_\mathsf{I} : \mathsf{Ty}_\mathsf{I}) \to \mathsf{Ty}_D\, A_\mathsf{I}\\
&\mathsf{Tm} &&: (a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}) \to \mathsf{Tm}_D\,(\mathsf{Ty}\, A_\mathsf{I})\, a_\mathsf{I}\\
&\mathsf{Bool} &&: \mathsf{Ty}\,\mathsf{Bool}_\mathsf{I} = \mathsf{Bool}_D\\
&\mathsf{Nat} &&: \mathsf{Ty}\,\mathsf{Nat}_\mathsf{I} = \mathsf{Nat}_D\\
&\mathsf{true} &&: \mathsf{Tm}\,\mathsf{true}_\mathsf{I} = \mathsf{true}_D\\
&\mathsf{false} &&: \mathsf{Tm}\,\mathsf{false}_\mathsf{I} = \mathsf{false}_D\\
&\mathsf{ite} &&: (b_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\,\mathsf{Bool}_\mathsf{I})(t_\mathsf{I}\, f_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}) \to \mathsf{Tm}\,(\mathsf{ite}_\mathsf{I}\, b_\mathsf{I}\, t_\mathsf{I}\, f_\mathsf{I}) = \mathsf{ite}_D\,(\mathsf{Tm}\, b_\mathsf{I})\,(\mathsf{Tm}\, t_\mathsf{I})\,(\mathsf{Tm}\, f_\mathsf{I})\\
&\mathsf{num} &&: (n : \mathbb{N}) \to \mathsf{Tm}\,(\mathsf{num}_\mathsf{I}\, n) = \mathsf{num}_D\, n\\
&- + - &&: (u_\mathsf{I}\, v_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\,\mathsf{Nat}_\mathsf{I}) \to \mathsf{Tm}\,(u_\mathsf{I} +_\mathsf{I} v_\mathsf{I}) = \mathsf{Tm}\, u_\mathsf{I} +_D \mathsf{Tm}\, v_\mathsf{I}\\
&\mathsf{isZero} &&: (u_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\,\mathsf{Nat}_\mathsf{I}) \to \mathsf{Tm}\,(\mathsf{isZero}_\mathsf{I}\, u_\mathsf{I}) = \mathsf{isZero}_D\,(\mathsf{Tm}\, u_\mathsf{I})
\end{aligned}$$

We define a dependent model where the term components are trivial (we don't list them):

$$\begin{aligned}
&\mathsf{Ty}\, A_\mathsf{I} &&:= \mathsf{Nf}\, A_\mathsf{I} \to \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}\\
&\mathsf{Tm}\, A\, a_\mathsf{I} &&:= \mathbb{1}\\
&\mathsf{Bool} : \underbrace{\mathsf{Nf}\, \mathsf{Bool}_\mathsf{I}}_{=\{\mathsf{tt},\mathsf{ff}\}} \to \mathsf{Tm}_\mathsf{I}\, \mathsf{Bool}_\mathsf{I}\\
&\mathsf{Bool}\, b &&:= \mathsf{match}\, b\, \{\mathsf{tt} \mapsto \mathsf{true}_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{false}_\mathsf{I}\}\\
&\mathsf{Nat} : \underbrace{\mathsf{Nf}\, \mathsf{Nat}_\mathsf{I}}_{=\mathbb{N}} \to \mathsf{Tm}_\mathsf{I}\, \mathsf{Nat}_\mathsf{I}\\
&\mathsf{Nat}\, n &&:= \mathsf{num}_\mathsf{I}\, n
\end{aligned}$$

Induction into this dependent model provides us with a quote function which maps normal forms back into syntactic terms:

$$\begin{aligned}
&\mathsf{quote} : (A_\mathsf{I} : \mathsf{Ty}_\mathsf{I}) \to \mathsf{Nf}\, A_\mathsf{I} \to \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}\\
&\mathsf{quote}\, \mathsf{Bool}_\mathsf{I}\, b = \mathsf{match}\, b\, \{\mathsf{tt} \mapsto \mathsf{true}_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{false}_\mathsf{I}\}\\
&\mathsf{quote}\, \mathsf{Nat}_\mathsf{I}\ \ n = \mathsf{num}_\mathsf{I}\, n
\end{aligned}$$

Completeness of normalisation says that for any $a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}$, $\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, a_\mathsf{I}) = a_\mathsf{I}$. We prove this by constructing another dependent model over I where the Ty component is trivial:

$$\begin{aligned}
&\mathsf{Ty}\, A_\mathsf{I} &&:= \mathbb{1}\\
&\mathsf{Tm}\, \{A_\mathsf{I}\} * a_\mathsf{I} &&:= (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, a_\mathsf{I}) = a_\mathsf{I})\\
&\mathsf{true} &&: \mathsf{quote}\, \mathsf{Bool}_\mathsf{I}\, (\mathsf{norm}\, \mathsf{true}_\mathsf{I}) = \mathsf{quote}\, \mathsf{Bool}_\mathsf{I}\, \mathsf{tt} = \mathsf{match}\, \mathsf{tt}\, \{\mathsf{tt} \mapsto \mathsf{true}_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{false}_\mathsf{I}\} = \mathsf{true}_\mathsf{I}\\
&\mathsf{false} &&: \mathsf{quote}\, \mathsf{Bool}_\mathsf{I}\, (\mathsf{norm}\, \mathsf{false}_\mathsf{I}) = \mathsf{quote}\, \mathsf{Bool}_\mathsf{I}\, \mathsf{ff} = \mathsf{match}\, \mathsf{ff}\, \{\mathsf{tt} \mapsto \mathsf{true}_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{false}_\mathsf{I}\} = \mathsf{false}_\mathsf{I}\\
&\mathsf{ite}\, (e_b : \mathsf{quote}\, \mathsf{Bool}_\mathsf{I}\, (\mathsf{norm}\, b_\mathsf{I}) = b_\mathsf{I})(e_t : \mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, t_\mathsf{I}) = t_\mathsf{I})(e_f : \mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, f_\mathsf{I}) = f_\mathsf{I})\\
&&&: \mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, (\mathsf{ite}_\mathsf{I}\, b_\mathsf{I}\, t_\mathsf{I}\, f_\mathsf{I}))\\
&&&\ \ \mathsf{quote}\, A_\mathsf{I}\, (\mathsf{match}\, (\mathsf{norm}\, b_\mathsf{I})\, \{\mathsf{tt} \mapsto \mathsf{norm}\, t_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{norm}\, f_\mathsf{I}\}) && =(\mathsf{norm}\, b_\mathsf{I} = \mathsf{tt})\\
&&&\ \ \mathsf{quote}\, A_\mathsf{I}\, (\mathsf{match}\, \mathsf{tt}\, \{\mathsf{tt} \mapsto \mathsf{norm}\, t_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{norm}\, f_\mathsf{I}) && =\\
&&&\ \ \mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, t_\mathsf{I}) && =(\mathsf{ite}\beta_{1\mathsf{I}})\\
&&&\ \ \mathsf{ite}_\mathsf{I}\, \mathsf{true}_\mathsf{I}\, (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, t_\mathsf{I}))\, (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, f_\mathsf{I})) && =\\
&&&\ \ \mathsf{ite}_\mathsf{I}\, (\mathsf{match}\, \mathsf{tt}\, \{\mathsf{tt} \mapsto \mathsf{true}_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{false}_\mathsf{I}\})\, (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, t_\mathsf{I}))\, (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, f_\mathsf{I})) && =(\mathsf{norm}\, b_\mathsf{I} = \mathsf{tt})\\
&&&\ \ \mathsf{ite}_\mathsf{I}\, (\mathsf{match}\, (\mathsf{norm}\, b_\mathsf{I})\, \{\mathsf{tt} \mapsto \mathsf{true}_\mathsf{I}; \mathsf{ff} \mapsto \mathsf{false}_\mathsf{I}\})\, (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, t_\mathsf{I}))\, (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, f_\mathsf{I})) && =\\
&&&\ \ \mathsf{ite}_\mathsf{I}\, (\mathsf{quote}\, \mathsf{Bool}_\mathsf{I}\, (\mathsf{norm}\, b_\mathsf{I}))\, (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, t_\mathsf{I}))\, (\mathsf{quote}\, A_\mathsf{I}\, (\mathsf{norm}\, f_\mathsf{I})) && =(e_b, e_t, e_f)\\
&&&\ \ \mathsf{ite}_\mathsf{I}\, b_\mathsf{I}\, t_\mathsf{I}\, f_\mathsf{I}
\end{aligned}$$

As terms in this dependent model are equations between elements of $\mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}$ for some A_I, we don't have to provide the equation components ($\mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}$ is a set in the sense of homotopy type theory, it has uniqueness of identity proofs).

Exercise 22. *Finish defining the dependent model: write the case* $\mathsf{norm}\, b_\mathsf{I} = \mathsf{ff}$ *for* ite *and define the components* num, $- + -$, isZero.

Exercise 23. *Merge* quote *and completeness, so that they both are parts of induction into a single dependent model.*

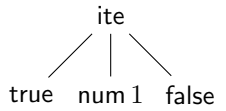
Exercise 24. *Prove* stability*: that is, given a normal form, if we quote it and then normalise it, we get back the same normal form. In summary, normalisation with completeness and stability says that there is an isomorphism between the sets* $\mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}$ *and* $\mathsf{Nf}\, A_\mathsf{I}$.

In general, *normal forms* are a complete representation of the syntax where equality is easily decidable (for example, they are given by an inductive definition, that is, the syntax of a GAT without equations). Decidable equality of normal forms implies decidable equality of terms by completeness: given $a_\mathsf{I}, a_\mathsf{I}' : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}$, if $e : \mathsf{norm}\, a_\mathsf{I} = \mathsf{norm}\, a_\mathsf{I}'$ then

$$a_\mathsf{I} \overset{\text{completeness}}{=} \mathsf{quote}\, (\mathsf{norm}\, a_\mathsf{I}) \overset{e}{=} \mathsf{quote}\, (\mathsf{norm}\, a_\mathsf{I}') \overset{\text{completeness}}{=} a_\mathsf{I}',$$

and given $\mathsf{norm}\, a_\mathsf{I} \neq \mathsf{norm}\, a_\mathsf{I}'$, from $a_\mathsf{I} = a_\mathsf{I}'$, by congruence we get $\mathsf{norm}\, a_\mathsf{I} = \mathsf{norm}\, a_\mathsf{I}'$. Stability is not needed for decidability of equality of the syntax, but it is useful to obtain a new induction principle for the syntax: one can prove things about the syntax by induction on normal forms.

Notation 25 (Derivation rules)**.** *Elements of the syntax can be depicted by syntax trees, but then the well-typedness of the tree is not immediately visible, it is not enforced by the drawing that the tree does not make sense. Instead, we first describe the notion of model using derivation rule notation which means that we uncurry all the operators, give names to the arguments and in each operator, we replace the remaining* $\to$ *with a horizontal line:*



$$\frac{}{\mathsf{Ty} : \mathsf{Set}} \qquad \frac{A : \mathsf{Ty}}{\mathsf{Tm}\, A : \mathsf{Set}} \qquad \frac{}{\mathsf{Bool} : \mathsf{Ty}} \qquad \frac{}{\mathsf{Nat} : \mathsf{Ty}} \qquad \frac{}{\mathsf{true} : \mathsf{Tm}\,\mathsf{Bool}} \qquad \frac{}{\mathsf{false} : \mathsf{Tm}\,\mathsf{Bool}}$$

$$\frac{b : \mathsf{Tm}\,\mathsf{Bool} \qquad t : \mathsf{Tm}\, A \qquad f : \mathsf{Tm}\, A}{\mathsf{ite}\, b\, t\, f : \mathsf{Tm}\, A} \qquad \frac{n : \mathbb{N}}{\mathsf{num}\, n : \mathsf{Tm}\,\mathsf{Nat}} \qquad \frac{u : \mathsf{Tm}\,\mathsf{Nat} \qquad v : \mathsf{Tm}\,\mathsf{Nat}}{u + v : \mathsf{Tm}\,\mathsf{Nat}} \qquad \frac{u : \mathsf{Tm}\,\mathsf{Nat}}{\mathsf{isZero}\, u : \mathsf{Tm}\,\mathsf{Bool}}$$

$$\frac{}{\mathsf{ite}\beta_1 : \mathsf{ite}\,\mathsf{true}\, u\, v = u} \qquad \frac{}{\mathsf{ite}\beta_2 : \mathsf{ite}\,\mathsf{false}\, u\, v = v} \qquad \frac{}{+\beta : \mathsf{num}\, m + \mathsf{num}\, n = \mathsf{num}\,(m + n)}$$

$$\frac{}{\mathsf{isZero}\beta_1 : \mathsf{isZero}\,(\mathsf{num}\, 0) = \mathsf{true}} \qquad \frac{}{\mathsf{isZero}\beta_2 : \mathsf{isZero}\,(\mathsf{num}\,(1 + n)) = \mathsf{false}}$$

Just as in the algebraic notation we have implicit arguments, for example $A : \mathsf{Ty}$ *is not written above the line for* ite*, but we assume it is (implicitly) there.*

Notation 26 (Derivation trees)**.** *Derivation trees are like upside-down syntax trees where at the nodes the full term and its set are both written; the children are subterms just as before. Example:*

$$\frac{\dfrac{\dfrac{}{\mathsf{num}\, 1 : \mathsf{Tm}\,\mathsf{Nat}}}{\mathsf{isZero}\,(\mathsf{num}\, 1) : \mathsf{Tm}\,\mathsf{Bool}} \qquad \frac{}{\mathsf{true} : \mathsf{Tm}\,\mathsf{Bool}} \qquad \frac{}{\mathsf{false} : \mathsf{Tm}\,\mathsf{Bool}}}{\mathsf{ite}\,(\mathsf{isZero}\,(\mathsf{num}\, 1))\,\mathsf{true}\,\mathsf{false} : \mathsf{Tm}\,\mathsf{Bool}}$$

Each horizontal line in the derivation tree is a special case of one of the derivation rules.

It is clear that the following tree cannot be finished:

$$\frac{\dfrac{}{\mathsf{true} : \mathsf{Tm}\,\mathsf{Bool}} \qquad \dfrac{}{\mathsf{num}\, 1 : \mathsf{Tm}\,\mathsf{Nat}} \qquad \dfrac{????}{\mathsf{false} : \mathsf{Tm}\,\mathsf{Nat}}}{\mathsf{ite}\,\mathsf{true}\,(\mathsf{num}\, 1)\,\mathsf{false} : \mathsf{Tm}\,\mathsf{Nat}}$$

Exercise 27. *Show that induction is equivalent to initiality.*

2.3.1 Type inference

The AST-level definition of Razor is the following.

Definition 28 (Untyped Razor)**.** *A model comprises the following components:*

$$\begin{aligned}
\mathsf{Tm} &: \mathsf{Set}\\
\mathsf{true} &: \mathsf{Tm}\\
\mathsf{false} &: \mathsf{Tm}\\
\mathsf{ite} &: \mathsf{Tm} \to \mathsf{Tm} \to \mathsf{Tm} \to \mathsf{Tm}\\
\mathsf{num} &: \mathbb{N} \to \mathsf{Tm}\\
_ + _ &: \mathsf{Tm} \to \mathsf{Tm} \to \mathsf{Tm}\\
\mathsf{isZero} &: \mathsf{Tm} \to \mathsf{Tm}
\end{aligned}$$

We define the following model of untyped Razor referring to the syntax of (typed) Razor. (Actually any model with decidability of equality for Ty suffices.)

Definition 29 (Type inference model)**.**

$$\begin{aligned}
\mathsf{Tm} &:= \mathsf{Maybe}\,((A : \mathsf{Ty}) \times \mathsf{Tm}\, A)\\
\mathsf{true} &:= \mathsf{just}\,(\mathsf{Bool}, \mathsf{true})\\
\mathsf{false} &:= \mathsf{just}\,(\mathsf{Bool}, \mathsf{false})\\
\mathsf{ite}\, t\, u\, v &:= \mathsf{match}\,(t, u, v)\,\{(\mathsf{just}\,(\mathsf{Bool}, t'), \mathsf{just}\,(\mathsf{Bool}, u'), \mathsf{just}(\mathsf{Bool}, v')) \mapsto \mathsf{just}\,(\mathsf{Bool}, \mathsf{ite}\, t'\, u'\, v');\\
&\qquad\qquad (\mathsf{just}\,(\mathsf{Bool}, t'), \mathsf{just}\,(\mathsf{Nat}, u'), \mathsf{just}(\mathsf{Nat}, v')) \mapsto \mathsf{just}\,(\mathsf{Nat}, \mathsf{ite}\, t'\, u'\, v');\\
&\qquad\qquad _ \mapsto \mathsf{nothing}\}\\
\mathsf{num}\, n &:= \mathsf{just}\,(\mathsf{Nat}, \mathsf{num}\, n)\\
u + v &:= \mathsf{match}\,(u, v)\,\{(\mathsf{just}\,(\mathsf{Nat}, u'), \mathsf{just}\,(\mathsf{Nat}, v')) \mapsto \mathsf{just}\,(\mathsf{Nat}, u' + v');\ _ \mapsto \mathsf{nothing}\}\\
\mathsf{isZero}\, t &:= \mathsf{match}\, t\,\{\mathsf{just}\,(\mathsf{Nat}, t') \mapsto \mathsf{just}\,(\mathsf{Bool}, \mathsf{isZero}\, t');\ _ \mapsto \mathsf{nothing}\}
\end{aligned}$$

Interpretation into this model is type inference. We apologise for the extreme overloading. It is clear from the absence of indices in the input of the function that we refer to the syntax of untyped Razor:

$$\mathsf{infer} : \mathsf{Tm_I} \to \mathsf{Maybe}\,((A : \mathsf{Ty}) \times \mathsf{Tm}\,A)$$

For example, we compute

$$\begin{aligned}
&\mathsf{infer}\,(\mathsf{ite}\,(\mathsf{num}\,1)\,\mathsf{true}\,\mathsf{false}) =\\
&\mathsf{match}\,(\mathsf{infer}\,(\mathsf{num}\,1), \mathsf{infer}\,\mathsf{true}, \mathsf{infer}\,\mathsf{false})\,\{(\mathsf{just}\,(\mathsf{Bool}, t'), \mathsf{just}\,(\mathsf{Bool}, u'), \mathsf{just}(\mathsf{Bool}, v')) \mapsto \mathsf{just}\,(\mathsf{Bool}, \mathsf{ite}\,t'\,u'\,v');\\
&\qquad\qquad (\mathsf{just}\,(\mathsf{Bool}, t'), \mathsf{just}\,(\mathsf{Nat}, u'), \mathsf{just}(\mathsf{Nat}, v')) \mapsto \mathsf{just}\,(\mathsf{Nat}, \mathsf{ite}\,t'\,u'\,v');\\
&\qquad\qquad _ \mapsto \mathsf{nothing}\} =\\
&\mathsf{match}\,(\mathsf{just}\,(\mathsf{Nat}, \mathsf{num}\,1), \mathsf{just}\,(\mathsf{Bool}, \mathsf{true}), \mathsf{just}\,(\mathsf{Bool}, \mathsf{false}))\,\{\\
&\quad (\mathsf{just}\,(\mathsf{Bool}, t'), \mathsf{just}\,(\mathsf{Bool}, u'), \mathsf{just}(\mathsf{Bool}, v')) \mapsto \mathsf{just}\,(\mathsf{Bool}, \mathsf{ite}\,t'\,u'\,v');\\
&\quad (\mathsf{just}\,(\mathsf{Bool}, t'), \mathsf{just}\,(\mathsf{Nat}, u'), \mathsf{just}(\mathsf{Nat}, v')) \mapsto \mathsf{just}\,(\mathsf{Nat}, \mathsf{ite}\,t'\,u'\,v');\\
&\quad _ \mapsto \mathsf{nothing}\} =\\
&\mathsf{nothing}.
\end{aligned}$$

Exercise 30. *Show that* $\mathsf{infer}\,(\mathsf{num}\,1 + \mathsf{num}\,2) = \mathsf{just}\,(\mathsf{Nat}, (\mathsf{num}\,1 + \mathsf{num}\,2))$. *Show that there is an* A *and* $t : \mathsf{Tm}\,A$ *such that* $\mathsf{infer}\,(\mathsf{num}\,0 + (\mathsf{ite}\,(\mathsf{isZero}\,(\mathsf{num}\,1))\,(\mathsf{num}\,1)\,(\mathsf{num}\,2))) = \mathsf{just}\,(A, t)$.

Exercise 31. *Define type erasure, a map from well-typed terms to untyped terms* $\mathsf{erase} : \mathsf{Tm}\,A \to \mathsf{Tm}$. *Give examples of* $t : \mathsf{Tm}$ *such that there is no* A *and* $t' : \mathsf{Tm}\,A$ *with* $\mathsf{erase}t' = t$. *Show soundness of inference, that is,* $\mathsf{infer}\,t = \mathsf{just}\,(A, t')$ *implies* $\mathsf{erase}\,t' = t$. *Show completeness of inference, that is, for any* $t : \mathsf{Tm}\,A$, *we have* $\mathsf{infer}\,(\mathsf{erase}\,t) = \mathsf{just}\,(A, t)$. *Using soundness and/or completeness, give generic ways to prove statements such as "for a given* t, *show that (doesn't) exist* A *and* $t' : \mathsf{Tm}\,A$ *such that* $\mathsf{erase}\,t' = t$*".*

2.4 Simply typed combinator calculus

Using Moses Schönfinkel's K and S combinators [Sch24], higher order functions can be described without variables.

Definition 32 (STCC)**.** *A model comprises the following components:*

$$\begin{aligned}
&\mathsf{Ty} && : \mathsf{Set}\\
&\mathsf{Tm} && : \mathsf{Ty} \to \mathsf{Set}\\
&\iota && : \mathsf{Ty}\\
&- \Rightarrow - && : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty}\\
&- \cdot - && : \mathsf{Tm}\,(A \Rightarrow B) \to \mathsf{Tm}\,A \to \mathsf{Tm}\,B\\
&\mathsf{K} && : \mathsf{Tm}\,(A \Rightarrow B \Rightarrow A)\\
&\mathsf{S} && : \mathsf{Tm}\,((A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C)\\
&\mathsf{K}\beta && : \mathsf{K} \cdot a \cdot b = a\\
&\mathsf{S}\beta && : \mathsf{S} \cdot f \cdot g \cdot a = f \cdot a \cdot (g \cdot a)
\end{aligned}$$

We added one base type so that the syntax is not empty. Note that $- \cdot -$ and K have two, S has three implicit arguments. $- \Rightarrow -$ is right-associative, application $- \cdot -$ is left-associative.

Exercise 33. *Derive the following combinators, their expected definitional behaviour is specified on the right:*

$$\begin{aligned}
&\mathsf{I} : \mathsf{Tm}\,(A \Rightarrow A) && \mathsf{I} \cdot a = a\\
&\mathsf{B} : \mathsf{Tm}\,((B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C) && \mathsf{B} \cdot f \cdot g \cdot a = f \cdot (g \cdot a)\\
&\mathsf{C} : \mathsf{Tm}\,((A \Rightarrow B \Rightarrow C) \Rightarrow B \Rightarrow A \Rightarrow C) && \mathsf{C} \cdot f \cdot b \cdot a = f \cdot a \cdot b
\end{aligned}$$

Exercise 34. *Define the standard model where* $\mathsf{Ty} = \mathsf{Set}$, $\mathsf{Tm}\,A = A$. *The interpretation of* ι *is a parameter of the standard model.*

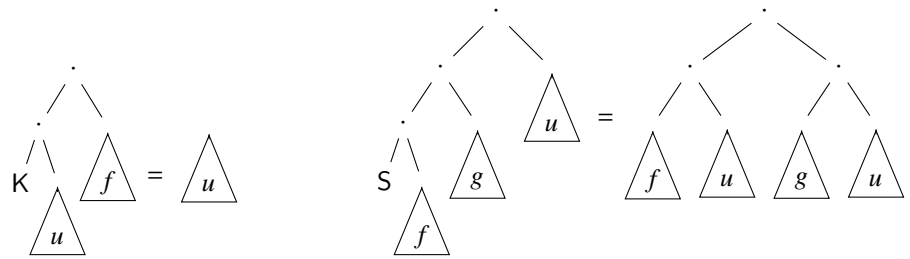
Theorem 35. *STCC is logically consistent, that is, there is a type* A_I *such that* $\mathsf{Tm_I}\,A_\mathsf{I}$ *is empty.*

Proof. Interpretating $\mathsf{Tm_I}\,\iota_\mathsf{I}$ into the standard model with $\iota = \mathbb{0}$ gives an element of $\mathbb{0}$. □

Theorem 36. *STCC is equationally consistent, meaning not all terms are equal. That is, there is a type* A_I *and terms* $a_\mathsf{I}, a'_\mathsf{I} : \mathsf{Tm}\,A_\mathsf{I}$ *such that* $a_\mathsf{I} \neq a'_\mathsf{I}$.

Proof. We choose $A_\mathsf{I} := (\iota_\mathsf{I} \Rightarrow_\mathsf{I} \iota_\mathsf{I} \Rightarrow_\mathsf{I} \iota_\mathsf{I})$ and $a_\mathsf{I} := \mathsf{lam}_\mathsf{I}\, \lambda x_\mathsf{I}.\, \mathsf{lam}\, \lambda y_\mathsf{I}.\, x_\mathsf{I}$ and $a'_\mathsf{I} := \mathsf{lam}_\mathsf{I}\, \lambda x_\mathsf{I}.\, \mathsf{lam}\, \lambda y_\mathsf{I}.\, y_\mathsf{I}$. We apply the iterator into the standard model with $\iota = \mathbb{2}$ both to a_I and a'_I. Asssuming $a_\mathsf{I} = a'_\mathsf{I}$, their interpretations are also equal, so we get $(\lambda x\, y.\, x) = (\lambda x\, y.\, y)$, and from this we get $\mathsf{tt} = (\lambda x\, y.\, x)\, \mathsf{tt}\, \mathsf{ff} = (\lambda x\, y.\, y)\, \mathsf{tt}\, \mathsf{ff} = \mathsf{ff}$. □

In the syntax of STCC, types are binary trees (nodes are $\Rightarrow$, leaves are ι). If we ignore types, terms are boolean-labelled binary trees quotiented by the following equations:



If we don't ignore types, then we can only build a node if the left hand subtree is in $\mathsf{Tm}\,(A \Rightarrow B)$ for some A, B, and the right hand subtree is in $\mathsf{Tm}\,A$ for the same A. Again, this is encforced by derivation trees, for example:

$$\dfrac{\dfrac{\overline{\mathsf{S} : \mathsf{Tm}\,((\iota \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota) \Rightarrow ((\iota \Rightarrow \iota \Rightarrow \iota) \Rightarrow \iota \Rightarrow \iota))} \qquad \overline{\mathsf{K} : \mathsf{Tm}\,(\iota \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota)}}{\mathsf{S} \cdot \mathsf{K} : \mathsf{Tm}\,((\iota \Rightarrow \iota \Rightarrow \iota) \Rightarrow \iota \Rightarrow \iota)} \qquad \overline{\mathsf{K} : \mathsf{Tm}\,(\iota \Rightarrow \iota \Rightarrow \iota)}}{\mathsf{S} \cdot \mathsf{K} \cdot \mathsf{K} : \mathsf{Tm}\,(\iota \Rightarrow \iota)}$$

Note that we didn't write implicit arguments. The two Ks on the right hand side of the tree are different: there is $\mathsf{K}\,\{\iota\}\,\{\iota\} : \mathsf{Tm}\,(\iota \Rightarrow \iota \Rightarrow \iota)$ and $\mathsf{K}\,\{\iota\}\,\{\iota \Rightarrow \iota\} : \mathsf{Tm}\,(\iota \Rightarrow (\iota \Rightarrow \iota) \Rightarrow \iota)$.

Exercise 37. *What is a nice minimal presyntax (untyped AST description) for STCC where types can be always inferred? That is, which implicit arguments have to be written explicitly?*

What are the normal forms for the syntax of STCC? We give an inductive description of the expressions where $\mathsf{K}\beta$ and $\mathsf{S}\beta$ cannot be applied because K and S does not have enough arguments:

Definition 38 (STCC normal forms)**.** *A normal form model comprises the following:*

$$\begin{aligned}
&\mathsf{Nf} : (A_\mathsf{I} : \mathsf{Ty}_\mathsf{I}) \to \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I} \to \mathsf{Set}\\
&\mathsf{K}_0 : \mathsf{Nf}\,(A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I} \Rightarrow_\mathsf{I} A_\mathsf{I})\, \mathsf{K}_\mathsf{I}\\
&\mathsf{K}_1 : \mathsf{Nf}\, A_\mathsf{I}\, a_\mathsf{I} \to \mathsf{Nf}\,(B_\mathsf{I} \Rightarrow_\mathsf{I} A_\mathsf{I})\,(\mathsf{K}_\mathsf{I} \cdot_\mathsf{I} a_\mathsf{I})\\
&\mathsf{S}_0 : \mathsf{Nf}\,((A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I} \Rightarrow_\mathsf{I} C_\mathsf{I}) \Rightarrow_\mathsf{I} (A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I}) \Rightarrow_\mathsf{I} A_\mathsf{I} \Rightarrow_\mathsf{I} C_\mathsf{I})\, \mathsf{S}_\mathsf{I}\\
&\mathsf{S}_1 : \mathsf{Nf}\,(A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I} \Rightarrow_\mathsf{I} C_\mathsf{I})\, f_\mathsf{I} \to \mathsf{Nf}\,((A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I}) \Rightarrow_\mathsf{I} A_\mathsf{I} \Rightarrow_\mathsf{I} C_\mathsf{I})\,(\mathsf{S}_\mathsf{I} \cdot_\mathsf{I} f_\mathsf{I})\\
&\mathsf{S}_2 : \mathsf{Nf}\,(A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I} \Rightarrow_\mathsf{I} C_\mathsf{I})\, f_\mathsf{I} \to \mathsf{Nf}\,(A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I})\, g_\mathsf{I} \to \mathsf{Nf}\,(A_\mathsf{I} \Rightarrow_\mathsf{I} C_\mathsf{I})\,(\mathsf{S}_\mathsf{I} \cdot_\mathsf{I} f_\mathsf{I} \cdot_\mathsf{I} g_\mathsf{I})
\end{aligned}$$

These normal forms are indexed not only by their syntactic type, but also by the term they correspond to (the result of quote).

Exercise 39. *Show that equality is decidable in the syntax of normal forms (this relies on equality of syntactic types). The nicest way is to do double-induction on normal forms to prove decidability of equality of the total space of normal forms, that is, the set* $(A_\mathsf{I} : \mathsf{Ty}_\mathsf{I}) \times (a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}) \times \mathsf{Nf}_\mathsf{I}\, A_\mathsf{I}\, a_\mathsf{I}$.

A dependent model over I comprises the following components:

$$\begin{aligned}
&\mathsf{Ty} && : \mathsf{Ty}_\mathsf{I} \to \mathsf{Set}\\
&\mathsf{Tm} && : \mathsf{Ty}\, A_\mathsf{I} \to \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I} \to \mathsf{Set}\\
&\iota && : \mathsf{Ty}\, \iota_\mathsf{I}\\
&- \Rightarrow - && : \mathsf{Ty}\, A_\mathsf{I} \to \mathsf{Ty}\, B_\mathsf{I} \to \mathsf{Ty}\,(A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I})\\
&- \cdot - && : \mathsf{Tm}\,(A \Rightarrow B)\, f_\mathsf{I} \to \mathsf{Tm}\, A\, a_\mathsf{I} \to \mathsf{Tm}\, B\,(f_\mathsf{I} \cdot_\mathsf{I} a_\mathsf{I})\\
&\mathsf{K} && : \mathsf{Tm}\,(A \Rightarrow B \Rightarrow A)\, \mathsf{K}_\mathsf{I}\\
&\mathsf{S} && : \mathsf{Tm}\,((A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C)\, S_\mathsf{I}\\
&\mathsf{K}\beta && : \mathsf{K} \cdot a \cdot b = a\\
&\mathsf{S}\beta && : \mathsf{S} \cdot f \cdot g \cdot a = f \cdot a \cdot (g \cdot a)
\end{aligned}$$

Given a dependent model D over I, a section comprises the following:

$$\begin{aligned}
&\mathsf{Ty} && : (A_\mathsf{I} : \mathsf{Ty}_\mathsf{I}) \to \mathsf{Ty}_D\, A_\mathsf{I}\\
&\mathsf{Tm} && : (a_\mathsf{I} : \mathsf{Tm}_A\, A_\mathsf{I}) \to \mathsf{Tm}_D\, (\mathsf{Ty}\, A_\mathsf{I})\, a_\mathsf{I}\\
&\iota && : \mathsf{Ty}\, \iota_\mathsf{I} = \iota_D\\
&- \Rightarrow - && : (A\, B : \mathsf{Ty}_\mathsf{I}) \to \mathsf{Ty}\, (A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I}) = \mathsf{Ty}\, A_\mathsf{I} \Rightarrow_D \mathsf{Ty}\, B_\mathsf{I}\\
&- \cdot - && : (f_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, (A_\mathsf{I} \Rightarrow_\mathsf{I} B_\mathsf{I}))(a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}) \Rightarrow \mathsf{Tm}\, (f_\mathsf{I} \cdot_\mathsf{I} a_\mathsf{I}) = \mathsf{Tm}\, f_\mathsf{I} \cdot_D \mathsf{Tm}\, a_\mathsf{I}\\
&\mathsf{K} && : \mathsf{Tm}\, \mathsf{K}_\mathsf{I} = K_D\\
&\mathsf{S} && : \mathsf{Tm}\ S_\mathsf{I} = S_D
\end{aligned}$$

We use Tait's method [Tai67] (also called logical predicate or reducibility method) to prove normalisation for STCC. We define a dependent model over I where types are a proof-relevant predicate over terms together with a reify function, and terms are witnesses of the predicate. The predicate for ι is constant false. The predicate for a function says that if the predicate holds for an input, it also holds for the output, and the function is in normal form.

$$\begin{aligned}
&\mathsf{Ty}\, A_\mathsf{I} && := (P_A : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I} \to \mathsf{Set}) \times \left(\{a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}\} \to P_A\, a_\mathsf{I} \to \mathsf{Nf}\, A_\mathsf{I}\, a_\mathsf{I}\right)\\
&\mathsf{Tm}\, (P_A, r_A)\, a_\mathsf{I} && := P_A\, a_\mathsf{I}\\
&\iota && := (\lambda_.\, \mathbb{0}, \lambda b.\, \mathsf{match}\, b\, \{\})\\
&(P_A, r_A) \Rightarrow (P_B, r_B) && := \left(\lambda f_\mathsf{I}.\, \left(\{a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}\} \to P_A\, a_\mathsf{I} \to P_B\, (f_\mathsf{I} \cdot_\mathsf{I} a_\mathsf{I})\right) \times \mathsf{Nf}\, (A_\mathsf{I} \Rightarrow B_\mathsf{I})\, f_\mathsf{I}, \lambda(p_f, n_f).n_f\right)\\
&(p_f, n_f) \cdot p_a && := p_f\, p_a\\
&\mathsf{K}\, \{P_A, r_A\}\{P_B, r_B\} && := \left(\lambda p_a.\, \left(\lambda p_b.\, p_a, \mathsf{K}_1\, (r_A\, a_\mathsf{I})\right), \mathsf{K}_0\right)\\
&\mathsf{S}\, \{P_A, r_A\}\{P_B, r_B\}\{P_C, r_C\} && := \left(\lambda(p_f, n_f).\, \left(\lambda(p_g, n_g).\, \left(\lambda p_a.\, (p_f\, p_a)._1\, (p_g\, p_a), \mathsf{S}_2\, n_f\, n_g\right), \mathsf{S}_1\, n_f\right), \mathsf{S}_0\right)
\end{aligned}$$

Equations $\mathsf{K}\beta$ and $\mathsf{S}\beta$ hold by definition. When defining K and S, we implicitly made use of $\mathsf{K}\beta_\mathsf{I}$ and $\mathsf{S}\beta_\mathsf{I}$, respectively. By induction into this model we obtain the predicate for each type, the reify function for each type and the witness of the predicate for each term:

$$\begin{aligned}
&P : (A_\mathsf{I} : \mathsf{Ty}_\mathsf{I}) \to \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I} \to \mathsf{Set}\\
&r\ : (A_\mathsf{I} : \mathsf{Ty}_\mathsf{I})\{a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}\} \to P\, A_\mathsf{I}\, a_\mathsf{I} \to \mathsf{Nf}\, A_\mathsf{I}\, a_\mathsf{I}\\
&p : (a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}) \to P\, A_\mathsf{I}\, a_\mathsf{I}
\end{aligned}$$

We put these together to obtain normalisation:

$$\mathsf{norm}\, (a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\, A_\mathsf{I}) := r\, A_\mathsf{I}\, a_\mathsf{I}\, (p\, a_\mathsf{I})$$

Note that in the normalisation dependent model, we only assumed an STCC normal form model, we did not rely on it being syntax. However, if it is the syntax of STCC normal forms, we can do the following:

Exercise 40. *Via normalisation, show decidability of equality for syntactic STCC terms.*

We were quite clever when coming up with the notion of normal forms. We don't need to be clever:

Exercise 41. *Show that the syntax of STCC without equations also suffices for normalisation and decidability of equality of (quotiented) STCC syntax.*

Exercise 42. *Intuitionistic logic with only implication as a connective is the same as STCC where we don't care about equations, a model is the following:*

$$\begin{aligned}
&\mathsf{For} && : \mathsf{Set}\\
&\mathsf{Pf} && : \mathsf{For} \to \mathsf{Set}\\
&\mathsf{irr} && : (a\, a' : \mathsf{Pf}\, A) \to a = a'\\
&\iota && : \mathsf{For}\\
&- \supset - && : \mathsf{For} \to \mathsf{For} \to \mathsf{For}\\
&- \cdot - && : \mathsf{Pf}\, (A \supset B) \to \mathsf{Pf}\, A \to \mathsf{Pf}\, B\\
&\mathsf{K} && : \mathsf{Pf}\, (A \supset B \supset A)\\
&\mathsf{S} && : \mathsf{Pf}\, ((A \supset B \supset C) \supset (A \supset B) \supset A \supset C)
\end{aligned}$$

Define normal forms as a proof-irrelevant predicate on Pf_I and show normalisation. Analysing the shape of normal forms, show that Peirce's law is not provable.

Food for thought 43. *Extend this logic with true, false, conjunction, disjunction, and show via normalisation that the law of excluded middle is not derivable.*

2.5 Other GATs

See [KKA19] for an algorithm which derives the notion of model, morphism, dependent model, dependent morphism from any GAT signature. Models and morphisms organise into a category (morphisms can be composed, composition is associative, etc), dependent models and dependent morphisms form a family structure over this category, together they produce a category with family, see Example 220.

Classes of inductive sets (meta-types) and classes of algebraic theories roughly correspond to each other. Inductive sets are initial models (syntaxes) for some algebraic theory, algebraic theories determine classes of algebras for some inductive types. The correspondence:

single sorted algebraic theory without equations	simple inductive type (W-type)
multi sorted algebraic theory without equations	mutually defined inductive types (indexed W-type)
single sorted algebraic theory	quotient inductive type (QIT, QW-type)
generalised algebraic theory without equations	inductive inductive type (IIT)
generalised algebraic theory (GAT)	quotient inductive inductive type (QIIT)
higher generalised algebraic theory	higher inductive inductive type (HIIT)

However algebraic theories are more fine-grained: for example, as inductive types $\mathsf{List}\,A$ and the free monoid over A are equivalent (Exercise 46), but as algebraic theories, the latter has a larger category of models. The situation is similar between the lambda calculus and combinatory calculus [AKSV23]. Mutual inductive types can be reduced to indexed inductive types [KvR20] which can again be reduced to simple inductive types [Kap19], but this is not the case for the corresponding classes of algebraic theories.

Definition 44 (Monoid over $A : \mathsf{Set}$)**.** *A model is a model of a monoid (Definition 1) extended with an operation* $\eta : A \to \mathsf{C}$.

The syntax of monoid over A is also called the free monoid over A.

Definition 45 (A-nil-cons)**.** *For an* $A : \mathsf{Set}$*, an* A*-nil-cons model comprises the following:*

$$\mathsf{L} : \mathsf{Set} \qquad \mathsf{nil} : \mathsf{L} \qquad \mathsf{cons} : A \to \mathsf{L} \to \mathsf{L}$$

The syntax of A-nil-cons is also called $\mathsf{List}\,A$.

Exercise 46. *Show that the syntax of* A*-nil-cons gives rise to a syntax of monoid over* A*. Show normalisation for monoids over* A *where normal forms are* $\mathsf{List}\,A$.

Definition 47 (Graph)**.** *A model comprises the following:*

$$\begin{aligned}\mathsf{Ob} &: \mathsf{Set}\\ \mathsf{Mor} &: \mathsf{Ob} \to \mathsf{Ob} \to \mathsf{Set}\end{aligned}$$

Exercise 48 (From [Moe22])**.** *Show that reflexivity (an operation* $(I : \mathsf{Ob}) \to \mathsf{Mor}\,I\,I$*) for graphs is admissible.*

Definition 49 (Category)**.** *A model is a graph with the following additional components:*

$$\begin{aligned}-\circ- &: \mathsf{Mor}\,J\,I \to \mathsf{Mor}\,K\,J \to \mathsf{Mor}\,K\,I\\ \mathsf{id} &: \mathsf{Mor}\,I\,I\\ \mathsf{ass} &: (f \circ g) \circ h = f \circ (g \circ h)\\ \mathsf{idl} &: \mathsf{id} \circ f = f\\ \mathsf{idr} &: f \circ \mathsf{id} = f\end{aligned}$$

Definition 50 (Monoid')**.** *A monoid is a category with an additional operation* $\mathsf{ob} : \mathsf{Ob}$ *and equation* $(I\,J : \mathsf{Ob}) \to I = J$.

Exercise 51. *What is the relationship between monoid and monoid'?*

Definition 52 (Preorder)**.** *A model is a category with the additional equation* $(f\,g : \mathsf{Mor}\,J\,I) \to f = g$.

Exercise 53. *What is the simplest definition of syntax for category? What about cartesian closed category?*

Normal forms and normalisation cannot be defined for arbitrary GATs. For example, in the syntax of untyped combinator calculus (Definition 186), equality is undecidable, hence it does not have normalisation.

3 Derivability in SOGATs

Second-order GATs (SOGATs) allow second-order operations. Second-order operations are also called binders. In this section, we define several languages with binders as SOGATs, and show how to define derivable operations and prove derivable equations in second-order models. The phrase "second-order" is a negative qualifier: second-order algebraic theories are not really algebraic (this is like people's democracy, which is not really democracy). For example, there is no good notion of morphism between second-order models. However, for derivability, second-order models are good enough. We will treat this problem in Section 4.

3.1 Simply typed lambda calculus

Definition 54 (Simply typed lambda calculus, STLC)**.** *A second-order model of STLC comprises the following components:*

$$\begin{aligned}
&\mathsf{Ty} && : \mathsf{Set}\\
&\mathsf{Tm} && : \mathsf{Ty} \to \mathsf{Set}\\
&\iota && : \mathsf{Ty}\\
&- \Rightarrow - && : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty}\\
&\mathsf{lam} && : (\mathsf{Tm}\, A \to \mathsf{Tm}\, B) \to \mathsf{Tm}\,(A \Rightarrow B)\\
&- \cdot - && : \mathsf{Tm}\,(A \Rightarrow B) \to \mathsf{Tm}\, A \to \mathsf{Tm}\, B\\
&\beta && : \mathsf{lam}\, b \cdot a = b\, a\\
&\eta && : f = \mathsf{lam}\, \lambda x.\, f \cdot x
\end{aligned}$$

The operators lam and $- \cdot -$ take *implicit arguments*, A and B. E.g. the explicit type of lam is $\{A : \mathsf{Ty}\}\{B : \mathsf{Ty}\} \to (\mathsf{Tm}\, A \to \mathsf{Tm}\, B) \to \mathsf{Tm}\,(A \Rightarrow B)$. All the arguments of the equations β and η are implicit. E.g. the explicit type of β is $\{A : \mathsf{Ty}\}\{B : \mathsf{Ty}\}\{b : \mathsf{Tm}\, A \to \mathsf{Tm}\, B\}\{a : \mathsf{Tm}\, A\} \to (- \cdot -)\,\{A\}\{B\}\,(\mathsf{lam}\,\{A\}\,\{B\}\, b)\, a = b\, a$. Note that in the equation η, f has to be in $\mathsf{Tm}\,(A \Rightarrow B)$ for some A and B for the equation to make sense.

The operation lam is second-order (also called a *binder*): its third input (after the two implicit inputs A, B) is a function.

Notation 55 (Isomorphism)**.** *The last four lines can be written more consisely:*

$$\mathsf{lam} : (\mathsf{Tm}\, A \to \mathsf{Tm}\, B) \cong \mathsf{Tm}\,(A \Rightarrow B) : - \cdot -$$

We say that the sets $(\mathsf{Tm}\, A \to \mathsf{Tm}\, B)$ *and* $\mathsf{Tm}\,(A \Rightarrow B)$ *are isomorphic. We write the maps on the two sides of* $\cong$*, the notation in general is*

$$(f : X \cong Y : g) \;:=\; (f : X \to Y) \times (g : Y \to X) \times \big((x : X) \to g\,(f\, x) = x\big) \times \big((y : Y) \to f\,(g\, y) = y\big).$$

Derivability is programming: we can program in a SOGAT by postulating a second-order model in Agda or Coq and combining the components to build new programs. We can run the programs by proving (deriving) equalities between them.

Example 56. *In any second-order model of STLC, we define the identity function on* ι *by* $\mathsf{lam}\,\{\iota\}\{\iota\}\,(\lambda x.\, x)$*, which is in* $\mathsf{Tm}\,(\iota \Rightarrow \iota)$*. Note the difference between the object theory* lam *and meta* λ*. Actually, we can define the identity function on any type using meta quantification:*

$$\lambda A.\, \mathsf{lam}\,\{A\}\{A\}\,(\lambda x.\, x) : (A : \mathsf{Ty}) \to \mathsf{Tm}\,(A \Rightarrow A).$$

We compute (run the programs) in a second-order model deriving equalities. E.g. given $u : \mathsf{Tm}\,\iota$ *we argue*

$$(\mathsf{lam}\,(\lambda x.\, x)) \cdot u \overset{\beta}{=} (\lambda x.\, x)\, u \overset{\text{meta function application}}{=} u.$$

This seems like cheating: we avoided writing substitution by pushing it into the metatheory. But this is OK, we do not want to fully bootstrap STLC, we just want to define it. In the metatheory we of course assume (higher-order) functions, function application, its β *rule, and so on.*

Notation 57 (Derivation rules for SOGATs)**.** *Similarly to GATs, the derivation rule notation uses uncurried function space, named parameters and horizontal lines instead of the arrow* $\to$ *symbol. For arrows in second-order positions, we use the turnstile* $\vdash$*, and we use named function application for the* $\vdash$ *function space. Second-order models of STLC are described by the following derivation rules:*

$$\frac{}{\mathsf{Ty} : \mathsf{Set}} \qquad \frac{A : \mathsf{Ty}}{\mathsf{Tm}\, A : \mathsf{Set}} \qquad \frac{}{\iota : \mathsf{Ty}} \qquad \frac{A : \mathsf{Ty} \qquad B : \mathsf{Ty}}{A \Rightarrow B : \mathsf{Ty}} \qquad \frac{x : \mathsf{Tm}\, A \vdash t : \mathsf{Tm}\, B}{\mathsf{lam}\, x.t : \mathsf{Tm}\,(A \Rightarrow B)}$$

$$\frac{t : \mathsf{Tm}\,(A \Rightarrow B) \qquad a : \mathsf{Tm}\,A}{t \cdot a : \mathsf{Tm}\,B} \qquad \frac{}{(\mathsf{lam}\,x.b) \cdot a = b[x \mapsto a]}\,\beta \qquad \frac{}{f = \mathsf{lam}\,x.f \cdot x}\,\eta$$

Some arguments of the horizontal line function space can be implicit. Sometimes the name of the rule is written on the right hand side of the horizontal line.

We can omit the $: \mathsf{Ty}$ *and* Tm *parts and more concisely write the following without sacrificing precision (this is always the case when there are two sorts* $\mathsf{Ty} : \mathsf{Set}$, $\mathsf{Tm} : \mathsf{Ty} \to \mathsf{Set}$*).*

$$\bar{\iota} \qquad \frac{A \qquad B}{A \Rightarrow B} \qquad \frac{x : A \vdash t : B}{\mathsf{lam}\,x.t : A \Rightarrow B} \qquad \frac{t : A \Rightarrow B \qquad a : A}{t \cdot a : B} \qquad \frac{}{(\mathsf{lam}\,x.b) \cdot a = b[x \mapsto a]} \qquad \frac{}{f = \mathsf{lam}\,x.f \cdot x}$$

Example 58 (Second-order operations in mathematics)**.** *For the integral operation, the derivation rule notation is on the left and the second-order algebraic notation is on the right:*

$$\frac{a : \mathbb{R} \qquad b : \mathbb{R} \qquad x : \mathbb{R} \vdash t : \mathbb{R}}{\int_a^b t\,dx} \qquad\qquad \int : \mathbb{R} \to \mathbb{R} \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R},$$

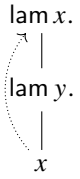
the expression $\int_0^1 \frac{1}{x^2}\,dx$ *corresponds to* $\int 0\,1\,(\lambda x.\,\frac{1}{x^2})$*.*

Notation 59 (Derivation trees for SOGATs)**.** *When we write derivation trees, the second-order arguments are collected on the left hand side of the* $\vdash$*. For example, given* $A : \mathsf{Ty}$ *and* $B : \mathsf{Ty}$*, we derive the constant function as follows: the verbose notation is on the left, the concise is on the right. In the concise notation we also omit the* λ*s.*

$$\frac{\dfrac{\dfrac{}{x : \mathsf{Tm}\,A, y : \mathsf{Tm}\,B \vdash x : \mathsf{Tm}\,A}}{x : \mathsf{Tm}\,A \vdash \mathsf{lam}\,\lambda y.x : \mathsf{Tm}\,(B \Rightarrow A)}}{\mathsf{lam}\,\lambda x.\,\mathsf{lam}\,\lambda y.\,x : \mathsf{Tm}\,(A \Rightarrow B \Rightarrow A)} \qquad\qquad \frac{\dfrac{\dfrac{}{x : A, y : B \vdash x : A}}{x : A \vdash \mathsf{lam}\,y.x : B \Rightarrow A}}{\mathsf{lam}\,x.\,\mathsf{lam}\,y.\,x : A \Rightarrow B \Rightarrow A}$$

Now the leaves of the tree can be assumptions from the left hand side of the turnstile in addition to derivation rules without arguments. The algebraic (or Coq/Agda) version of this is simply the conclusion $\mathsf{lam}\,\lambda x.\,\mathsf{lam}\,\lambda y.\,x$*, or* $\mathsf{lam}\,(\lambda x.\,\mathsf{lam}\,(\lambda y.\,x))$ *with more brackets.*

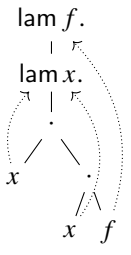
Notation 60 (Abstract binding trees (ABTs))**.** *Another notation for SOGAT-derivation trees is abstract* binding *trees where variables are pointers to the binders which are reachable through the path to the root. The constant function is drawn as follows.*



Abstract binding tree notation has the same issue as abstract syntax trees: the restriction that trees can only be put together in well-typed ways is not apparent.

Example 61. *We define the "double" function using short derivation rule and abstract binding tree notation.*

$$\frac{\dfrac{\dfrac{\dfrac{}{f : A \Rightarrow A, x : A \vdash f : A \Rightarrow A} \qquad \dfrac{\dfrac{}{f : A \Rightarrow A, x : A \vdash f : A \Rightarrow A} \qquad \dfrac{}{f : A \Rightarrow A, x : A \vdash x : A}}{f : A \Rightarrow A, x : A \vdash f \cdot x : A}}{f : A \Rightarrow A, x : A \vdash f \cdot (f \cdot x) : A}}{f : A \Rightarrow A \vdash \mathsf{lam}\,x.f \cdot (f \cdot x) : A \Rightarrow A}}{\mathsf{lam}\,f.\mathsf{lam}\,x.f \cdot (f \cdot x) : (A \Rightarrow A) \Rightarrow A \Rightarrow A}$$



Example 62. *Function composition:*

$$\frac{\dfrac{\dfrac{\dfrac{}{f : B \Rightarrow A, g : C \Rightarrow B, x : C \vdash f : B \Rightarrow A} \qquad \dfrac{\dfrac{}{f : B \Rightarrow A, g : C \Rightarrow B, x : C \vdash g : C \Rightarrow B} \qquad \dfrac{}{\ldots, x : C \vdash x : C}}{f : B \Rightarrow A, g : C \Rightarrow B, x : C \vdash g \cdot x : B}}{f : B \Rightarrow A, g : C \Rightarrow B, x : C \vdash f \cdot (g \cdot x) : A}}{f : B \Rightarrow A, g : C \Rightarrow B \vdash \mathsf{lam}\,x.f \cdot (g \cdot x) : C \Rightarrow A}}{f : B \Rightarrow A \vdash \mathsf{lam}\,g.\mathsf{lam}\,x.f \cdot (g \cdot x) : (C \Rightarrow B) \Rightarrow C \Rightarrow A}}{\mathsf{lam}\,f.\mathsf{lam}\,g.\mathsf{lam}\,x.f \cdot (g \cdot x) : (B \Rightarrow A) \Rightarrow (C \Rightarrow B) \Rightarrow C \Rightarrow A}$$

Example 63. *We define a model of STCC (Definition 32) assuming a second-order model of STLC.*

$$\begin{aligned}
\mathsf{Ty} &:= \mathsf{Ty}\\
\mathsf{Tm}\,A &:= \mathsf{Tm}\,A\\
\iota &:= \iota\\
A \Rightarrow B &:= A \Rightarrow B\\
\mathsf{K} &:= \mathsf{lam}\left(\lambda u.\,\mathsf{lam}\left(\lambda f.\,u\right)\right)\\
\mathsf{S} &:= \mathsf{lam}\left(\lambda f.\,\mathsf{lam}\left(\lambda g.\,\mathsf{lam}\left(\lambda u.\,f\cdot u\cdot(g\cdot u)\right)\right)\right)
\end{aligned}$$

The equations hold, e.g.:

$$\begin{aligned}
\mathsf{K}\beta : \mathsf{K}\cdot u\cdot f &&=&(\textit{definition of } \mathsf{K})\\
\left(\left(\mathsf{lam}\left(\lambda u.\,\mathsf{lam}\left(\lambda f.\,u\right)\right)\right)\cdot u\right)\cdot f &&=&(\beta)\\
\left(\left(\lambda u.\,\mathsf{lam}\left(\lambda f.\,u\right)\right)u\right)\cdot f &&=&(\textit{meta function application})\\
\left(\mathsf{lam}\left(\lambda f.\,u\right)\right)\cdot f &&=&(\beta)\\
(\lambda f.\,u)\,f &&=&(\textit{meta function application})\\
u
\end{aligned}$$

Exercise 64. *Prove that* $\mathsf{S}\beta$ *holds.*

Food for thought 65. *Can we derive a second-order model of STLC from a model of STCC? What if we start with a second-order model having some extra equations [AKSV23]?*

Example 66. *We can use derivation tree building to do informal type inference. For example, we don't know whether the expression*

$$\mathsf{lam}\,\lambda x.\mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z$$

has a type (makes sense in a second-order model of STLC), so we try to build its derivation tree from the bottom. Let's say it has an arbitrary type A*.*

$$\frac{?}{\mathsf{lam}\,\lambda x.\mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : A}$$

As the expression starts with lam*, we use its derivation rule and we learn that* $A = B \Rightarrow C$ *for some* B *and* C*:*

$$\frac{\dfrac{?}{x : B \vdash \mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : C}}{\mathsf{lam}\,\lambda x.\mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : B \Rightarrow C}$$

We use lam *again and we learn* $C = D \Rightarrow E$*, so we replace all occurrences of* C *with* $D \Rightarrow E$*:*

$$\frac{\dfrac{\dfrac{?}{x : B, y : D \vdash \mathsf{lam}\,\lambda z.y\cdot x\cdot z : E}}{x : B \vdash \mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : D \Rightarrow E}}{\mathsf{lam}\,\lambda x.\mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : B \Rightarrow (D \Rightarrow E)}$$

We use lam *again and we learn* $E = F \Rightarrow G$*:*

$$\frac{\dfrac{\dfrac{\dfrac{?}{x : B, y : D, z : F \vdash y\cdot x\cdot z : G}}{x : B, y : D \vdash \mathsf{lam}\,\lambda z.y\cdot x\cdot z : F \Rightarrow G}}{x : B \vdash \mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : D \Rightarrow (F \Rightarrow G)}}{\mathsf{lam}\,\lambda x.\mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : B \Rightarrow (D \Rightarrow (F \Rightarrow G))}$$

Now we can only use the derivation rule of application $(\cdot)$*, and we introduce a new metavariable* H *because we cannot read off the domain of the function from its conclusion. Recall that* $y\cdot x\cdot z = (y\cdot x)\cdot z$*.*

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{?}{x : B, y : D, z : F \vdash y\cdot x : H \Rightarrow G} \qquad \dfrac{?}{x : B, y : D, z : F \vdash z : H}}{x : B, y : D, z : F \vdash y\cdot x\cdot z : G}}{x : B, y : D \vdash \mathsf{lam}\,\lambda z.y\cdot x\cdot z : F \Rightarrow G}}{x : B \vdash \mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : D \Rightarrow (F \Rightarrow G)}}{\mathsf{lam}\,\lambda x.\mathsf{lam}\,\lambda y.\mathsf{lam}\,\lambda z.y\cdot x\cdot z : B \Rightarrow (D \Rightarrow (F \Rightarrow G))}$$

We first reach a leaf on the right hand side: z is to the left of the turnstile, so we learn $H = F$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{?}{x : B, y : D, z : F \vdash y \cdot x : F \Rightarrow G} \quad \overline{x : B, y : D, z : F \vdash z : F}}{x : B, y : D, z : F \vdash y \cdot x \cdot z : G}}{x : B, y : D \vdash \mathsf{lam}\, \lambda z.y \cdot x \cdot z : F \Rightarrow G}}{x : B \vdash \mathsf{lam}\, \lambda y.\mathsf{lam}\, \lambda z.y \cdot x \cdot z : D \Rightarrow (F \Rightarrow G)}}{\mathsf{lam}\, \lambda x.\mathsf{lam}\, \lambda y.\mathsf{lam}\, \lambda z.y \cdot x \cdot z : B \Rightarrow (D \Rightarrow (F \Rightarrow G))}$$

On the left hand side, there is an application $\cdot$, we introduce a new metavariable I:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{?}{x : B, y : D, z : F \vdash y : I \Rightarrow F \Rightarrow G} \quad \dfrac{?}{x : B, y : D, z : F \vdash x : I}}{x : B, y : D, z : F \vdash y \cdot x : F \Rightarrow G} \quad \overline{x : B, y : D, z : F \vdash z : F}}{x : B, y : D, z : F \vdash y \cdot x \cdot z : G}}{x : B, y : D \vdash \mathsf{lam}\, \lambda z.y \cdot x \cdot z : F \Rightarrow G}}{x : B \vdash \mathsf{lam}\, \lambda y.\mathsf{lam}\, \lambda z.y \cdot x \cdot z : D \Rightarrow (F \Rightarrow G)}}{\mathsf{lam}\, \lambda x.\mathsf{lam}\, \lambda y.\mathsf{lam}\, \lambda z.y \cdot x \cdot z : B \Rightarrow (D \Rightarrow (F \Rightarrow G))}$$

The solution to the right hand side ? forces $I = B$, because this is the type of x:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{?}{x : B, y : D, z : F \vdash y : B \Rightarrow F \Rightarrow G} \quad \overline{x : B, y : D, z : F \vdash x : B}}{x : B, y : D, z : F \vdash y \cdot x : F \Rightarrow G} \quad \overline{x : B, y : D, z : F \vdash z : F}}{x : B, y : D, z : F \vdash y \cdot x \cdot z : G}}{x : B, y : D \vdash \mathsf{lam}\, \lambda z.y \cdot x \cdot z : F \Rightarrow G}}{x : B \vdash \mathsf{lam}\, \lambda y.\mathsf{lam}\, \lambda z.y \cdot x \cdot z : D \Rightarrow (F \Rightarrow G)}}{\mathsf{lam}\, \lambda x.\mathsf{lam}\, \lambda y.\mathsf{lam}\, \lambda z.y \cdot x \cdot z : B \Rightarrow (D \Rightarrow (F \Rightarrow G))}$$

On the left hand side we learn $D = B \Rightarrow F \Rightarrow G$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{x : B, y : B \Rightarrow F \Rightarrow G, z : F \vdash y : B \Rightarrow F \Rightarrow G} \quad \overline{x : B, y : B \Rightarrow F \Rightarrow G, z : F \vdash x : B}}{x : B, y : B \Rightarrow F \Rightarrow G, z : F \vdash y \cdot x : F \Rightarrow G} \quad \overline{\ldots, z : F \vdash z : F}}{x : B, y : B \Rightarrow F \Rightarrow G, z : F \vdash y \cdot x \cdot z : G}}{x : B, y : B \Rightarrow F \Rightarrow G \vdash \mathsf{lam}\, \lambda z.y \cdot x \cdot z : F \Rightarrow G}}{x : B \vdash \mathsf{lam}\, \lambda y.\mathsf{lam}\, \lambda z.y \cdot x \cdot z : (B \Rightarrow F \Rightarrow G) \Rightarrow (F \Rightarrow G)}}{\mathsf{lam}\, \lambda x.\mathsf{lam}\, \lambda y.\mathsf{lam}\, \lambda z.y \cdot x \cdot z : B \Rightarrow ((B \Rightarrow F \Rightarrow G) \Rightarrow (F \Rightarrow G))}$$

So the most general type of the expression is $B \Rightarrow ((B \Rightarrow F \Rightarrow G) \Rightarrow (F \Rightarrow G))$ for some types B, F, G.

Exercise 67. *Using the above method, try to infer the type of the following epxressions (we assume Razor extended with functions merging Definitions 14 and 54):*

$$\mathsf{lam}\, \lambda x.\, x \cdot x$$
$$\mathsf{lam}\, \lambda f.\, f \cdot (\mathsf{num}\, 1 + f \cdot \mathsf{true})$$
$$\mathsf{lam}\, \lambda f.\, f \cdot (\mathsf{num}\, 1 + f \cdot \mathsf{num}\, 2)$$

Exercise 68. *Show that function composition (Example 62) is associative. That is, abbreviating* $\mathsf{comp} := \mathsf{lam}\, f.\mathsf{lam}\, g.\mathsf{lam}\, x.f \cdot (g \cdot x)$*, for every u, v, w we have* $\mathsf{comp} \cdot (\mathsf{comp} \cdot u \cdot v) \cdot w = \mathsf{comp} \cdot u \cdot (\mathsf{comp} \cdot v \cdot w)$.

Exercise 69. *Show that for any $A : \mathsf{Ty}$, $\mathsf{Tm}\,(A \Rightarrow A)$ is a monoid with function composition and identity.*

Exercise 70. *Show that we can define a category where objects are* Ty*, a morphism from A to B is* $\mathsf{Tm}\,(A \Rightarrow B)$*, composition is function composition (a refined version of the previous exercise).*

We cannot prove logical consistency just assuming a second-order model, because it might be the trivial model (where all sorts are $\mathbb{1}$). We also cannot prove equational consistency saying that there are terms a, a' such that $a = a' \to \mathbb{0}$, but we can get something similar: instead of obtaining a meta $\mathbb{0}$, we obtain that the second-order model is trivial:

Exercise 71. *Prove that for any $B : \mathsf{Ty}$ we have a $A : \mathsf{Ty}$ and $a, a' : \mathsf{Tm}\, A$ where $a = a'$ implies that any $b, b' : \mathsf{Tm}\, B$, $b = b'$.*

Exercise 72. *There is a second-order model of STLC where there is an $A : \mathsf{Ty}$ and $a, a' : \mathsf{Tm}\, A$ such that $a \neq a'$.*

3.1.1 Bidirectional type checking

The ABT-level (well-scoped) definition of STLC is the following.

Definition 73 (Well-scoped STLC)**.** *A first-order model of well-scoped STLC with De Bruijn indices comprises the following components:*

$$
\begin{array}{ll}
\mathsf{Con} & : \mathsf{Set} \\
\diamond & : \mathsf{Con} \\
- \triangleright & : \mathsf{Con} \to \mathsf{Con} \\
\mathsf{Var} & : \mathsf{Con} \to \mathsf{Set} \\
\mathsf{vz} & : \mathsf{Var}\,(\Gamma \triangleright) \\
\mathsf{vs} & : \mathsf{Var}\,\Gamma \to \mathsf{Var}\,(\Gamma \triangleright) \\
\mathsf{Tm} & : \mathsf{Con} \to \mathsf{Set} \\
\mathsf{var} & : \mathsf{Var}\,\Gamma \to \mathsf{Tm}\,\Gamma \\
\mathsf{lam} & : \mathsf{Tm}\,(\Gamma \triangleright) \to \mathsf{Tm}\,\Gamma \\
- \cdot - & : \mathsf{Tm}\,\Gamma \to \mathsf{Tm}\,\Gamma \to \mathsf{Tm}\,\Gamma \\
\mathsf{Ty} & : \mathsf{Set} \\
\iota & : \mathsf{Ty} \\
- \Rightarrow - & : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} \\
\mathsf{ann} & : \mathsf{Tm}\,\Gamma \to \mathsf{Ty} \to \mathsf{Tm}\,\Gamma
\end{array}
$$

Contexts (Con) give the information about the number of variables: $\diamond$ is the empty context, $\diamond \triangleright$ is the context containing one variable, $\diamond \triangleright \triangleright$ contains two variables, and so on. In the syntax of well-scoped STLC, Con is isomorphic to natural numbers. Var contains well-scoped De-Bruijn indices. In the syntax, $\mathsf{Var}\,\Gamma$ is the finite set containing Γ many elements. Terms are indexed by the context: in the syntax, $\mathsf{Tm}\,\Gamma$ is a term that can refer to Γ many variables. In particular, $\mathsf{Tm}\,\diamond$ is a closed term. Terms include annotated terms (ann) where a type for the term is specified.

Some examples for terms in the above well-scoped language (left) and the corresponding terms in a second-order model of STLC:

$\mathsf{lam}\,(\mathsf{var}\,\mathsf{vz})$	$\mathsf{lam}\,\lambda x.\,x$
$\mathsf{lam}\,(\mathsf{lam}\,(\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz})))$	$\mathsf{lam}\,\lambda x.\,\mathsf{lam}\,\lambda y.\,x$
$\mathsf{lam}\,(\mathsf{lam}\,(\mathsf{var}\,\mathsf{vz}))$	$\mathsf{lam}\,\lambda x.\,\mathsf{lam}\,\lambda y.\,y$
$\mathsf{lam}\,(\mathsf{lam}\,(\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}) \cdot (\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}) \cdot \mathsf{var}\,\mathsf{vz})))$	$\mathsf{lam}\,\lambda f.\,\mathsf{lam}\,\lambda x.\,f \cdot (f \cdot x)$

Type checking should map the left hand side untyped terms to the right hand side typed terms. However there is not enough information in these untyped terms to guess the type in the right hand side (which are implicit arguments). This is why we have annotated terms. The type inference algorithm will map the following untyped terms (left) to the typed terms (right). We added some implicit arguments.

$\mathsf{ann}\,(\mathsf{lam}\,(\mathsf{var}\,\mathsf{vz}))\,(\iota \Rightarrow \iota)$	$\mathsf{lam}\,\{\iota\}\,\lambda x.\,x$
$\mathsf{ann}\,(\mathsf{lam}\,(\mathsf{lam}\,(\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}))))\,(\iota \Rightarrow \iota \Rightarrow \iota)$	$\mathsf{lam}\,\{\iota\}\,\lambda x.\,\mathsf{lam}\,\{\iota\}\,\lambda y.\,x$
$\mathsf{ann}\,(\mathsf{lam}\,(\mathsf{lam}\,(\mathsf{var}\,\mathsf{vz})))\,(\iota \Rightarrow \iota \Rightarrow \iota)$	$\mathsf{lam}\,\{\iota\}\,\lambda x.\,\mathsf{lam}\,\{\iota\}\,\lambda y.\,y$
$\mathsf{ann}\,(\mathsf{lam}\,(\mathsf{lam}\,(\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}) \cdot (\mathsf{var}\,(\mathsf{vs}\,\mathsf{vz}) \cdot \mathsf{var}\,\mathsf{vz}))))\,((\iota \Rightarrow \iota) \Rightarrow \iota \Rightarrow \iota)$	$\mathsf{lam}\,\{\iota \Rightarrow \iota\}\lambda f.\,\mathsf{lam}\,\{\iota\}\,\lambda x.\,f \cdot (f \cdot x)$

Definition 74 (Typechecking and inference model)**.** *The following is a model of well-scoped STLC, referring to a second-order model of STLC. We rely on decidability of equality of* Ty *denoted by* $\stackrel{?}{=}$ *and by matching on its*

elements by λ, and we use the monadic operations of Maybe *denoted* $\gg\!=$, $\gg$, guard.

$$
\begin{array}{ll}
\mathsf{Con} & := (U : \mathsf{Set}) \times (El : U \to \mathsf{Set}) \\
\diamond & := (\mathbb{1}, \lambda\_.\, \mathbb{1}) \\
(U, El) \triangleright & := (U \times \mathsf{Ty}, \lambda(\Gamma, A).\, El\, \Gamma \times \mathsf{Tm}\, A) \\
\mathsf{Var}\, (U, El) & := (\Gamma : U) \to (A : \mathsf{Ty}) \times (El\, \Gamma \to \mathsf{Tm}\, A) \\
\mathsf{vz} & := \lambda(\Gamma, A).\, (A, \lambda(\gamma, a).\, a) \\
\mathsf{vs}\, x & := \lambda(\Gamma, B).\, \mathsf{match}\, (x\, \Gamma)\, \{(A, a) \mapsto (A, \lambda(\gamma, b).\, a\, \gamma)\} \\
\mathsf{Tm}\, (U, El) & := (\Gamma : U) \to \big((A : \mathsf{Ty}) \to \mathsf{Maybe}\, (\mathsf{El}\, \Gamma \to \mathsf{Tm}\, A)\big) \times \mathsf{Maybe}\, \big((A : \mathsf{Ty}) \times (\mathsf{El}\, \Gamma \to \mathsf{Tm}\, A)\big) \\
\mathsf{var}\, x\, \Gamma & := \mathsf{match}\, (x\, \Gamma)\, \big\{(A, a) \mapsto \big((\lambda A'.\, \mathsf{guard}\, (A \stackrel{?}{=} A') \gg \mathsf{just}\, (A, a)), \mathsf{just}\, (A, a)\big)\big\} \\
\mathsf{lam}\, t\, \Gamma & := \big((\lambda(A \Rightarrow B).\, (t\, (\Gamma, A))_{.1}\, B \gg\!= \lambda f.\, \mathsf{just}\, \lambda\gamma.\, \mathsf{lam}\, \lambda a.\, f\, (\gamma, a)), \mathsf{nothing}\big) \\
(t \cdot u)\, \Gamma & := \mathsf{match}\, (t\, \Gamma, u\, \Gamma)\, \big\{((\_, \mathsf{just}\, (A \Rightarrow B, t')), (chk_u, \_)) \mapsto \\
& \qquad \big((\lambda B'.\, chk_u\, A \gg\!= \lambda u'.\, \mathsf{guard}\, (B \stackrel{?}{=} B') \gg \mathsf{just}\, \lambda\gamma.\, t'\, \gamma \cdot u'\, \gamma), chk_u\, A \gg\!= \lambda u'.\, \mathsf{just}\, \lambda\gamma.\, t'\gamma \cdot u'\gamma)\big)\big\} \\
\mathsf{Ty} & := \mathsf{Ty} \\
\iota & := \iota \\
A \Rightarrow B & := A \Rightarrow B \\
\mathsf{ann}\, t\, A\, \Gamma & := \big(\lambda A'.\, \mathsf{guard}\, (A \stackrel{?}{=} A') \gg (t\, \Gamma)_{.1}\, A, (t\, \Gamma)_{.1}\, A \gg\!= \lambda a.\, \mathsf{just}\, (A, a)\big)
\end{array}
$$

Interpretation into this model are the following functions:

$$
\begin{array}{ll}
\mathsf{Tys} & : \mathsf{Con_I} \to \mathsf{Set} \\
\mathsf{Tms} & : (n : \mathsf{Con_I}) \to \mathsf{Tys}\, n \to \mathsf{Set} \\
\mathsf{check} & : \mathsf{Tm_I}\, n \to (\Gamma : \mathsf{Tys}\, n) \to (A : \mathsf{Ty}) \to \mathsf{Maybe}\, (\mathsf{Tms}\, \Gamma \to \mathsf{Tm}\, A) \\
\mathsf{infer} & : \mathsf{Tm_I}\, n \to (\Gamma : \mathsf{Tys}\, n) \to \mathsf{Maybe}\, \big((A : \mathsf{Ty}) \times (\mathsf{Tms}\, \Gamma \to \mathsf{Tm}\, A)\big)
\end{array}
$$

For a closed preterm $t : \mathsf{Tm_I}\, \diamond$, the last two specialise as follows:

$$
\begin{array}{ll}
\mathsf{check}\, t & : \mathbb{1} \to (A : \mathsf{Ty}) \to \mathsf{Maybe}\, (\mathbb{1} \to \mathsf{Tm}\, A) \\
\mathsf{infer}\, t & : \mathbb{1} \to \mathsf{Maybe}\, \big((A : \mathsf{Ty}) \times (\mathbb{1} \to \mathsf{Tm}\, A)\big)
\end{array}
$$

Exercise 75. *Do type checking and inference on some example preterms.*

3.2 System T

The following language has a proper type of natural numbers, it does not simply inherit $\mathbb{N}$ from the metatheory, like Razor did.

Definition 76 (System T)**.** *A second-order model of System T comprises the following components:*

$$
\begin{array}{ll}
\mathsf{Ty} & : \mathsf{Set} \\
\mathsf{Tm} & : \mathsf{Ty} \to \mathsf{Set} \\
- \Rightarrow - & : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} \\
\mathsf{lam} & : (\mathsf{Tm}\, A \to \mathsf{Tm}\, B) \cong \mathsf{Tm}\, (A \Rightarrow B) : - \cdot - \\
\mathsf{Nat} & : \mathsf{Ty} \\
\mathsf{zero} & : \mathsf{Tm}\, \mathsf{Nat} \\
\mathsf{suc} & : \mathsf{Tm}\, \mathsf{Nat} \to \mathsf{Tm}\, \mathsf{Nat} \\
\mathsf{ite} & : \mathsf{Tm}\, A \to (\mathsf{Tm}\, A \to \mathsf{Tm}\, A) \to \mathsf{Tm}\, \mathsf{Nat} \to \mathsf{Tm}\, A \\
\mathsf{Nat}\beta_1 & : \mathsf{ite}\, z\, s\, \mathsf{zero} = z \\
\mathsf{Nat}\beta_2 & : \mathsf{ite}\, z\, s\, (\mathsf{suc}\, t) = s\, (\mathsf{ite}\, z\, s\, t)
\end{array}
$$

Natural numbers are now given by zero and successor suc, for example, three is $\mathsf{suc}\, (\mathsf{suc}\, (\mathsf{suc}\, \mathsf{zero}))$. Functions out of natural numbers can be defined via ite (iterator, also called recursor, fold).

Definition 77 (Constructors, destructors, computation and uniqueness rules)**.** *Operators for each type can be grouped into constructors (introduction rules) and destructors (elimination rules, eliminators). Equations can be grouped into β (computation) and η (uniqueness) rules. β rules explain how to compute if a destructor is applied to a constructor, η rules express what happens if a constructor is applied to a destructor.*

We group our rules for System T and Razor for comparison.

language	type	constructor	destructor	computation	uniqueness
System T (Def. 76)	$- \Rightarrow -$	lam	$- \cdot -$	$\Rightarrow\beta$	$\Rightarrow\eta$
	Nat	$\mathsf{zero}, \mathsf{suc}$	ite	$\mathsf{Nat}\beta_1, \mathsf{Nat}\beta_2$	
Razor (Def. 14)	Bool	$\mathsf{true}, \mathsf{false}$	ite	$\mathsf{ite}\beta_1, \mathsf{ite}\beta_2$	
	Nat	num	$- + -, \mathsf{isZero}$	$+\beta, \mathsf{isZero}\beta_1, \mathsf{isZero}\beta_2$	

Nat does not come with an η rule, but this will be fixed in Section 3.5.

Example 78. *The double function:*

$$\begin{aligned}&\mathsf{double} : \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow \mathsf{Nat})\\&\mathsf{double} := \mathsf{lam}\,\lambda x.\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,x\end{aligned}$$

Computing the double of 2:

$$\begin{aligned}&\mathsf{double} \cdot \mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}) && =(\textit{abbreviation})\\&\big(\mathsf{lam}\,\lambda x.\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,x\big) \cdot \mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}) && =(\beta)\\&\big(\lambda x.\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,x\big)\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero})) && =(\textit{meta application})\\&\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero})) && =(\mathsf{Nat}\beta_2)\\&(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,\big(\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,(\mathsf{suc}\;\mathsf{zero})\big) && =(\textit{meta aplication})\\&\mathsf{suc}\,\Big(\mathsf{suc}\,\big(\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,(\mathsf{suc}\;\mathsf{zero})\big)\Big) && =(\mathsf{Nat}\beta_2)\\&\mathsf{suc}\,\Big(\mathsf{suc}\,\big((\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,(\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,\mathsf{zero})\big)\Big) && =(\textit{meta application})\\&\mathsf{suc}\,\Big(\mathsf{suc}\,\big(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,\mathsf{zero}))\big)\Big) && =(\mathsf{Nat}\beta_1)\\&\mathsf{suc}\,\Big(\mathsf{suc}\,\big(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero})\big)\Big)\end{aligned}$$

Definition 79 (Embedding $\mathbb{N}$ into $\mathsf{Tm}\,\mathsf{Nat}$)**.** *The meta function* $\ulcorner - \urcorner : \mathbb{N} \to \mathsf{Tm}\,\mathsf{Nat}$ *is defined as* $\ulcorner n \urcorner := \mathsf{suc}^n\,\mathsf{zero}$*, e.g.* $\ulcorner 3 \urcorner = \mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}))$*.*

Definition 80 (Definability)**.** *An* $f : \mathbb{N} \to \mathbb{N}$ *is definable in a second-order model of System T if there is a* $t : \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow \mathsf{Nat})$ *such that for all* n*,* $\ulcorner f\,n \urcorner = t \cdot \ulcorner n \urcorner$*. An* $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ *is definable if there is a* $t : \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow \mathsf{Nat} \Rightarrow \mathsf{Nat})$ *such that for all* m*,* n*,* $\ulcorner f\,m\,n \urcorner = t \cdot \ulcorner m \urcorner \cdot \ulcorner n \urcorner$*.*

Exercise 81. *Addition, multiplication, exponentiation, the Ackermann function are definable in any second-order model of System T.*

Exercise 82. *Which of the following terms define identity?*

$$\begin{aligned}&\mathsf{lam}\,\lambda x.x\\&\mathsf{lam}\,\lambda x.\mathsf{ite}\,\mathsf{zero}\,\mathsf{suc}\,x\\&\mathsf{lam}\,\lambda x.\mathsf{ite}\,x\,(\lambda y.y)\,x\\&\mathsf{lam}\,\lambda x.\mathsf{ite}\,\mathsf{zero}\,(\lambda y.y)\,x\\&\mathsf{lam}\,\lambda x.\mathsf{ite}\,(\mathsf{suc}\,\mathsf{zero})\,\mathsf{suc}\,x\\&\mathsf{lam}\,\lambda x.\mathsf{ite}\,(\mathsf{suc}\,\mathsf{zero})\,(\lambda y.y)\,x\end{aligned}$$

Exercise 83. *Which of the following terms define the function* $x \mapsto 2 * x + 1$*?*

$$\begin{aligned}&\mathsf{lam}\,\lambda x.\mathsf{suc}\,x\\&\mathsf{lam}\,\lambda x.\mathsf{suc}\,(\mathsf{ite}\,x\,\mathsf{suc}\,x)\\&\mathsf{lam}\,\lambda x.\mathsf{ite}\,(\mathsf{suc}\,x)\,\mathsf{suc}\,x\\&\mathsf{lam}\,\lambda x.\mathsf{ite}\,(\mathsf{suc}\,\mathsf{zero})\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,x\\&\mathsf{lam}\,\lambda x.\mathsf{suc}\,(\mathsf{ite}\,\mathsf{zero}\,(\lambda y.\mathsf{suc}\,(\mathsf{suc}\,y))\,x)\end{aligned}$$

Exercise 84. *Show that if* $\mathsf{zero} = \mathsf{suc}\,t$*, then for any* $u, v : \mathsf{Tm}\,A$*, we have* $u = v$*.*

Exercise 85. *Show that the predecessor function is definable (difficult, but it will become easier once we have product types).*

It is not possible to define a term $\mathsf{pred} : \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow \mathsf{Nat})$ such that $\mathsf{pred} \cdot \mathsf{suc}\, t = t$ for any $t : \mathsf{Tm}\,\mathsf{Nat}$. [Par89]. For this, we would need the recursor for Nat which in its method for successor receives the natural number, not only the result of the recursive call:

$$\begin{aligned}
&\mathsf{rec} : \mathsf{Tm}\,A \to (\mathsf{Tm}\,\mathsf{Nat} \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A) \to \mathsf{Tm}\,\mathsf{Nat} \to \mathsf{Tm}\,A\\
&\mathsf{rec}\,z\,s\,\mathsf{zero} = z\\
&\mathsf{rec}\,z\,s\,(\mathsf{suc}\,t) = s\,t\,(\mathsf{rec}\,z\,s\,t)
\end{aligned}$$

Now we can define $\mathsf{pred} := \mathsf{lam}\,\lambda x.\,\mathsf{rec}\,\mathsf{zero}\,(\lambda n\,\_.\,n)\,x$ and it satisfies

$$\begin{aligned}
&\mathsf{pred} \cdot\ \mathsf{suc}\,t && =\text{(abbreviation)}\\
&(\mathsf{lam}\,\lambda x.\mathsf{rec}\,\mathsf{zero}\,(\lambda y\,_.y)\,x) \cdot \mathsf{suc}\,t && =(\Rightarrow\beta)\\
&(\lambda x.\mathsf{rec}\,\mathsf{zero}\,(\lambda y\,_.y)\,x)\,(\mathsf{suc}\,t) && =\text{(meta application)}\\
&\mathsf{rec}\,\mathsf{zero}\,(\lambda y\,_.y)\,(\mathsf{suc}\,t) && =(\mathsf{Nat}\beta_2)\\
&(\lambda y\,_.y)\,t\,(\mathsf{rec}\,\mathsf{zero}\,(\lambda y\,_.y)\,t) && =\text{(meta application)}\\
&t.
\end{aligned}$$

Exercise 86. *Show that the iterator can be derived from the recursor.*

Exercise 87. *Show that the factorial is definable in System T with recursor.*

Exercise 88. *Define a model of Razor (Definition 14) assuming a second-order model of System T with recursor.*

Theorem 89. *There is an $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ function, and if this function is definable in System T, then there is a $v : \mathsf{Tm}\,\mathsf{Nat}$, such that $v = \mathsf{suc}\,v$.*

Proof. First we fix a Gödel coding which takes an arbitrary term of type $\mathsf{Nat} \Rightarrow \mathsf{Nat}$ and turns it into a natural number: $\mathsf{code} : \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow \mathsf{Nat}) \to \mathbb{N}$.

An evaluator for System T is an $\mathsf{eval} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ function with the following property: for every u and n, we have $\ulcorner \mathsf{eval}\,(\mathsf{code}\,u)\,n \urcorner = u \cdot \ulcorner n \urcorner$. Now eval being definable means that there is a t with $\ulcorner \mathsf{eval}\,m\,n \urcorner = t \cdot \ulcorner m \urcorner \cdot \ulcorner n \urcorner$ for every m and n. Now we define u as follows: $u := \mathsf{lam}\,\lambda x.\mathsf{suc}\,(t \cdot x \cdot x)$. And we reason:

$$\begin{aligned}
&t \cdot \ulcorner \mathsf{code}\,u \urcorner \cdot \ulcorner \mathsf{code}\,u \urcorner && =(\mathsf{eval}\text{ definable})\\
&\ulcorner \mathsf{eval}\,(\mathsf{code}\,u)\,(\mathsf{code}\,u) \urcorner && =\text{(property of eval)}\\
&u \cdot \ulcorner \mathsf{code}\,u \urcorner && =\text{(definition of } u)\\
&(\mathsf{lam}\,\lambda x.\mathsf{suc}\,(t \cdot x \cdot x)) \cdot \ulcorner \mathsf{code}\,u \urcorner && =(\Rightarrow\beta)\\
&\mathsf{suc}\,(t \cdot \ulcorner \mathsf{code}\,u \urcorner \cdot \ulcorner \mathsf{code}\,u \urcorner)
\end{aligned}$$

So we choose $v := t \cdot \ulcorner \mathsf{code}\,u \urcorner \cdot \ulcorner \mathsf{code}\,u \urcorner$ and then $v = \mathsf{suc}\,v$. □

Remark 90. *Unlike $\mathsf{zero} = \mathsf{suc}\,\mathsf{zero}$, the existence of a v with $v = \mathsf{suc}\,v$ in a second-order model does not imply that any two terms are equal. However, the above result implies that an evaluator for System T cannot be defined in the syntax of System T. For the notion of syntax for second-order languages (like System T), see Section 4. If we have $\mathsf{Nat}\eta$ in a second-order model, then the existence of v with $v = \mathsf{suc}\,v$ implies that any two terms are equal. However, in the syntax of System T with $\mathsf{Nat}\eta$, equality of terms is undecidable, and thus a Gödel-coding function is not possible.*

3.3 Finite types

We extend STLC with nullary and binary sums and products.

Definition 91 (Language with finite types)**.** *A second-order model of Fin is STLC (Definition 54) extended*

with the following components:

$$
\begin{array}{ll}
\bot & : \mathsf{Ty} \\
\mathsf{exfalso} & : \mathsf{Tm}\,\bot \to \mathsf{Tm}\,A \\
\bot\eta & : (t : \mathsf{Tm}\,\bot \to \mathsf{Tm}\,A) \to t = \mathsf{exfalso} \\
- + - & : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} \\
\mathsf{inl} & : \mathsf{Tm}\,A \to \mathsf{Tm}\,(A + B) \\
\mathsf{inr} & : \mathsf{Tm}\,B \to \mathsf{Tm}\,(A + B) \\
\mathsf{case} & : (\mathsf{Tm}\,A \to \mathsf{Tm}\,C) \to (\mathsf{Tm}\,B \to \mathsf{Tm}\,C) \to \mathsf{Tm}\,(A + B) \to \mathsf{Tm}\,C \\
+\beta_1 & : \mathsf{case}\,f\,g\,(\mathsf{inl}\,a) = f\,a \\
+\beta_2 & : \mathsf{case}\,f\,g\,(\mathsf{inr}\,b) = g\,b \\
+\eta & : (t : \mathsf{Tm}\,(A + B) \to \mathsf{Tm}\,C) \to t = \mathsf{case}\,(\lambda a.\,t\,(\mathsf{inl}\,a))\,(\lambda b.\,t\,(\mathsf{inr}\,b)) \\
\top & : \mathsf{Ty} \\
\mathsf{tt} & : \mathsf{Tm}\,\top \\
\top\eta & : t = tt \\
- \times - & : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} \\
-, - & : \mathsf{Tm}\,A \to \mathsf{Tm}\,B \to \mathsf{Tm}\,(A \times B) \\
\mathsf{fst} & : \mathsf{Tm}\,(A \times B) \to \mathsf{Tm}\,A \\
\mathsf{snd} & : \mathsf{Tm}\,(A \times B) \to \mathsf{Tm}\,B \\
\times\beta_1 & : \mathsf{fst}\,(a, b) = a \\
\times\beta_2 & : \mathsf{snd}\,(a, b) = b \\
\times\eta & : t = (\mathsf{fst}\,t, \mathsf{snd}\,t)
\end{array}
$$

Exercise 92. *Group the above operators into constructors and destructors. Check that the β and η rules adhere to their specification.*

Exercise 93. *Define* Bool *by* $\top + \top$ *and show that the following operations are definable and equations are provable:*

$$
\begin{array}{ll}
\mathsf{true} & : \mathsf{Tm}\,\mathsf{Bool} \\
\mathsf{false} & : \mathsf{Tm}\,\mathsf{Bool} \\
\mathsf{ite} & : \mathsf{Tm}\,A \to \mathsf{Tm}\,A \to \mathsf{Tm}\,\mathsf{Bool} \to \mathsf{Tm}\,A \\
\mathsf{Bool}\beta_1 & : \mathsf{ite}\,a\,a'\,\mathsf{true} = a \\
\mathsf{Bool}\beta_2 & : \mathsf{ite}\,a\,a'\,\mathsf{false} = a' \\
\mathsf{Bool}\eta & : (t : \mathsf{Tm}\,\mathsf{Bool} \to \mathsf{Tm}\,A) \to t = \mathsf{ite}\,(t\,\mathsf{true})\,(t\,\mathsf{false})
\end{array}
$$

Exercise 94. *Show that the terms* $\mathsf{lam}\,\lambda x.\mathsf{ite}\,x\,\mathsf{false}\,\mathsf{true}$ *and* $\mathsf{lam}\,\lambda x.(\mathsf{ite}\,x\,\mathsf{false}\,(\mathsf{ite}\,x\,\mathsf{true}\,\mathsf{true}))$ *are equal (you will need* $\mathsf{Bool}\eta$*).*

Exercise 95. *Define four terms of type* $\mathsf{Bool} \Rightarrow \mathsf{Bool}$ *such that if any two are equal, then* $\mathsf{true} = \mathsf{false}$*!*

Exercise 96. *Show that every term of type* $\mathsf{Bool} \Rightarrow \mathsf{Bool}$ *is equal to one of the four terms defined in the previous exercise.*

Exercise 97. *Define the three element type with a case operator and use it to define modulo 3 addition.*

Exercise 98. *Using* Bool *and product types, define homogeneous binary sums (that is, $A+A$ for any A). Define its constructors, destructor and derive its β and η rules.*

Exercise 99. *Define the option (maybe) type former which adds an extra element to an existing type.*

Exercise 100. *For every $n : \mathbb{N}$, define the* $\mathsf{Fin}\,n$ *type which contains exactly n elements. Define its constructors, destructor.*

Exercise 101. *For every n, define n-ary product and sum types.*

Exercise 102. *Derive* $h\,(\mathsf{case}\,f\,g\,w) = \mathsf{case}\,(\lambda a.\,h\,(f\,a))\,(\lambda b.\,h\,(g\,b))\,w$ *(this is called naturality of* case *in C).*

Exercise 103. *Show that there is exactly one term of the following types: $A \Rightarrow \top$ and $\bot \Rightarrow A$ (for arbitrary A).*

Definition 104 (Type isomorphism)**.** *A and B types are isomorphic, if there are $t : \mathsf{Tm}\,A \to \mathsf{Tm}\,B$ and $t' : \mathsf{Tm}\,B \to \mathsf{Tm}\,A$, such that $t'\,(t\,a) = a$ $t\,(t'\,b) = b$ for all a, b. We denote this by $A \cong B$.*

Exercise 105. *Show $(A \times \top) \cong A$ for every A.*

Exercise 106. *Show that for any A, B we have $A + B \cong B + A$.*

The next exercise is the type theory version of the algebraic equation $(a+b)^2 = a^2 + 2ab + b^2$.

Exercise 107. *Show that for arbitrary A, B types, $\mathsf{Bool} \Rightarrow (A + B) \cong \mathsf{Bool} \Rightarrow A + \mathsf{Bool} \times A \times B + \mathsf{Bool} \Rightarrow B$.*

Exercise 108. *Show that the $\cong$ relation is reflexive, symmetric and transitive (equivalence relation), and it is a congruence for $\times$ and $+$. (For example $A \cong A'$ implies $A \times B \cong A' \times B$.)*

Definition 109 (Commutative exponential semiring)**.** *A commutative exponential semiring (commutative exponential ring without negation, commutative exponential rig) is a $\mathsf{C} : \mathsf{Set}$ with the following operations and equations. As $+$ and $*$ are commutative, we only list one identity and distributivity law. We don't give names to the equations.*

$$
\begin{aligned}
- + - &: \mathsf{C} \to \mathsf{C} \to \mathsf{C}\\
0 &: \mathsf{C}\\
- * - &: \mathsf{C} \to \mathsf{C} \to \mathsf{C}\\
1 &: \mathsf{C}\\
-^{-} &: \mathsf{C} \to \mathsf{C} \to \mathsf{C}\\
&: (a+b)+c = a+(b+c)\\
&: a+b = b+a\\
&: 0+a = a\\
&: (a*b)*c = a*(b*c)\\
&: a*b = b*a\\
&: 1*a = a\\
&: a*(b+c) = a*b+a*c\\
&: a^{b+c} = a^b * a^c\\
&: a^0 = 1\\
&: (a*b)^c = a^c * b^c\\
&: 1^c = 1\\
&: a^{b*c} = (a^b)^c\\
&: a^1 = a
\end{aligned}
$$

Exercise 110. *(Meta) natural numbers form a commutative exponential semiring with addition, multiplication and exponentiation. They also form a commutative semiring with maximum and addition. Natural numbers extended with infinity form a commutative semiring with minimum and addition.*

Exercise 111. *In the language Fin, types form a commutative semiring where equations are isomorphisms.*

Remark 112. *Commutative exponential semirings are what Tarski calls high school algebra [BL93] (although they omit the operation 0). A related interesting completeness problem was settled by Wilkie [Wil01]. The dependently typed analogue is still open [Alt08].*

3.4 Generic programming

In this section we follow the book of Robert Harper [Har16].

Assume we have a $\mathsf{Bool} \Rightarrow \mathsf{Nat}$ function, one which maps true to 1 and false to 0 (e.g. $\mathsf{lam}\,(\mathsf{ite}\,(\mathsf{suc}\,\mathsf{zero})\,\mathsf{zero})$). We can apply this to a term of type Bool. But we can also apply it to a term in $\mathsf{Bool} \times \mathsf{Nat}$ in a way that the Nat part is not touched, i.e. (t, t') is mapped to $(\mathsf{ite}\,(\mathsf{suc}\,\mathsf{zero})\,\mathsf{zero}\,t, t')$. Or we can apply it to a term in $(\mathsf{Nat} + \mathsf{Bool}) \times \mathsf{Bool}$ where we apply it to the two Bools and we leave the Nat.

In general: we want to extend an $A \Rightarrow B$ function to an $F : \mathsf{Ty} \to \mathsf{Ty}$ type operator (a type depending on a type) in such a way that we obtain an $F\,A \Rightarrow F\,B$ function. This is what generic programming allows us for certain F type operators. An $F = \lambda X.\,A$, where A can only contain X, nullary and binary (co)products is called a polynomial type operator. If A contains function types, then the positive type operators are those for which the function can be extended to.

The type operator corresponding to the polynomial $\lambda x.2x^3 + x^2 + 3x + 1$ is $\lambda X.(\top + \top) \times (X \times X \times X) + X \times X + (\top + \top + \top) \times X + \top$.

Definition 113 (Map for polynomial type operators)**.** *For a second-order model of Fin (Definition 91), a model of map for polynomial type operators has one sort* Poly*, and some unnamed operators:*

$$\begin{aligned}
\mathsf{Poly} &: (\mathsf{Ty} \to \mathsf{Ty}) \to \mathsf{Set} \\
&: \mathsf{Poly}\,(\lambda X.X) \\
&: \mathsf{Poly}\,(\lambda X.\top) \\
&: \mathsf{Poly}\,F \to \mathsf{Poly}\,G \to \mathsf{Poly}\,(\lambda X.F\,X \times G\,X) \\
&: \mathsf{Poly}\,(\lambda X.\bot) \\
&: \mathsf{Poly}\,F \to \mathsf{Poly}\,G \to \mathsf{Poly}\,(\lambda X.F\,X + G\,X)
\end{aligned}$$

There is also an operator map *with some unnamed equations:*

$$\begin{aligned}
&\mathsf{map} : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{Poly}\,F\} \to (\mathsf{Tm}\,A \to \mathsf{Tm}\,B) \to \mathsf{Tm}\,(F\,A) \to \mathsf{Tm}\,(F\,B) \\
&\mathsf{map}\,(\lambda X.X)\quad\quad\quad\quad f\,a = f\,a \\
&\mathsf{map}\,(\lambda X.\top)\quad\quad\quad\quad f\,t = \mathsf{tt} \\
&\mathsf{map}\,(\lambda X.F\,X \times G\,X)\;f\,w = \big(\mathsf{map}\,F\,f\,(\mathsf{fst}\,w), \mathsf{map}\,G\,f\,(\mathsf{snd}\,w)\big) \\
&\mathsf{map}\,(\lambda X.\bot)\quad\quad\quad\quad f\,t = \mathsf{exfalso}\,t \\
&\mathsf{map}\,(\lambda X.F\,X + G\,X)\;f\,w = \mathsf{case}\,w\,\big(\lambda u.\mathsf{inl}\,(\mathsf{map}\,F\,f\,u)\big)\,\big(\lambda v.\mathsf{inr}\,(\mathsf{map}\,G\,f\,v)\big)
\end{aligned}$$

Exercise 114. *Show that the following type operator is polynomial.*

$$\lambda X.\bot + (\top \times X + X \times X)$$

Exercise 115. *Derive* $\mathsf{map}\,(\lambda X.\top + (\mathsf{Bool} \times X))\,f\,(\mathsf{inr}\,(\mathsf{true}, t)) = \mathsf{inr}\,(\mathsf{true}, f\,t)$.

Exercise 116. *Assuming* Poly *is the initial* Poly *model (that is, a meta inductive type), define* map *as iteration into a* Poly *model.*

Exercise 117. *Assuming* map *is defined as in Exercise 123, show that if* $A : \mathsf{Ty}$*, then* $\mathsf{map}\,(\lambda X.A)\,f\,w = w$.

Exercise 118. *Assuming* map *is defined as in Exercise 123, show that it is a functor:* $\mathsf{map}\,F\,(\lambda x.f\,(g\,x))\,w = \mathsf{map}\,F\,f\,(\mathsf{map}\,F\,g\,w)$ *and* $\mathsf{map}\,F\,(\lambda x.x)\,u = u$.

If we also allow functions in the type operator, then sometimes we cannot extend our type operator. For example there is no map function for the type operators

- $F\,X = X \Rightarrow \bot$,
- $F\,X = X \Rightarrow X$,

however there is a map function for

- $F\,X = \mathsf{Bool} \Rightarrow X$,
- $F\,X = ((X + X) \Rightarrow \mathsf{Bool}) \Rightarrow X$.

In the following language we introduce the notions of positive and negative type operators. We have a (covariant) map for the positive type operators while we have a contravariant map for the negative ones.

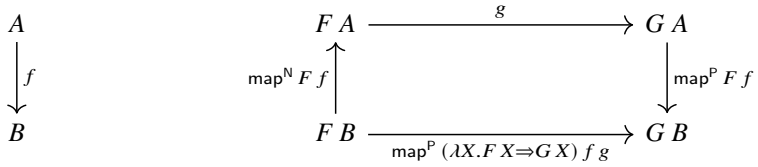
Definition 119 (Map for positive type operators)**.** *For a second-order model of Fin (Definition 91), a model of map for positive type operators has two sorts* Pos *and* Neg *and some unnamed operators:*

$$\begin{aligned}
\mathsf{Pos} &: (\mathsf{Ty} \to \mathsf{Ty}) \to \mathsf{Set} \\
\mathsf{Neg} &: (\mathsf{Ty} \to \mathsf{Ty}) \to \mathsf{Set} \\
&: \mathsf{Pos}\,(\lambda X.X) \\
&: (C : \mathsf{Ty}) \to \mathsf{Pos}\,(\lambda_.C) \\
&: (C : \mathsf{Ty}) \to \mathsf{Neg}\,(\lambda_.C) \\
&: \mathsf{Pos}\,F \to \mathsf{Pos}\,G \to \mathsf{Pos}\,(\lambda X.F\,X + G\,X) \\
&: \mathsf{Neg}\,F \to \mathsf{Neg}\,G \to \mathsf{Neg}\,(\lambda X.F\,X + G\,X) \\
&: \mathsf{Pos}\,F \to \mathsf{Pos}\,G \to \mathsf{Pos}\,(\lambda X.F\,X \times G\,X) \\
&: \mathsf{Neg}\,F \to \mathsf{Neg}\,G \to \mathsf{Neg}\,(\lambda X.F\,X \times G\,X) \\
&: \mathsf{Neg}\,F \to \mathsf{Pos}\,G \to \mathsf{Pos}\,(\lambda X.F\,X \Rightarrow G\,X) \\
&: \mathsf{Pos}\,F \to \mathsf{Neg}\,G \to \mathsf{Neg}\,(\lambda X.F\,X \Rightarrow G\,X)
\end{aligned}$$

There are also two operators satisfying the equations below:

$$\begin{aligned}
&\mathsf{map}^{\mathsf{P}} : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{Pos}\, F\} \to (\mathsf{Tm}\, A \to \mathsf{Tm}\, B) \to \mathsf{Tm}\,(F\, A) \to \mathsf{Tm}\,(F\, B)\\
&\mathsf{map}^{\mathsf{N}} : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{Neg}\, F\} \to (\mathsf{Tm}\, A \to \mathsf{Tm}\, B) \to \mathsf{Tm}\,(F\, B) \to \mathsf{Tm}\,(F\, A)\\
&\mathsf{map}^{\mathsf{P}}\ (\lambda X.X) \qquad\quad f\, a = f\, a\\
&\mathsf{map}^{\mathsf{P}}\ (\lambda X.C) \qquad\quad f\, c = c\\
&\mathsf{map}^{\mathsf{N}}\ (\lambda X.C) \qquad\quad f\, c = c\\
&\mathsf{map}^{\mathsf{P}}\ (\lambda X.F\, X + G\, X) \quad f\, w = \mathsf{case}\, w\, \big(\lambda u.\mathsf{inl}\,(\mathsf{map}^{\mathsf{P}}\, F\, f\, u)\big)\, \big(\lambda v.\mathsf{inr}\,(\mathsf{map}^{\mathsf{P}}\, G\, f\, v)\big)\\
&\mathsf{map}^{\mathsf{N}}\ (\lambda X.F\, X + G\, X) \quad f\, w = \mathsf{case}\, w\, \big(\lambda u.\mathsf{inl}\,(\mathsf{map}^{\mathsf{N}}\, F\, f\, u)\big)\, \big(\lambda v.\mathsf{inr}\,(\mathsf{map}^{\mathsf{N}}\, G\, f\, v)\big)\\
&\mathsf{map}^{\mathsf{P}}\ (\lambda X.F\, X \times G\, X) \quad f\, w = \big(\mathsf{map}^{\mathsf{P}}\, F\, f\, (\mathsf{fst}\, w), \mathsf{map}^{\mathsf{P}}\, G\, f\, (\mathsf{snd}\, w)\big)\\
&\mathsf{map}^{\mathsf{N}}\ (\lambda X.F\, X \times G\, X) \quad f\, w = \big(\mathsf{map}^{\mathsf{N}}\, F\, f\, (\mathsf{fst}\, w), \mathsf{map}^{\mathsf{N}}\, G\, f\, (\mathsf{snd}\, w)\big)\\
&\mathsf{map}^{\mathsf{P}}\ (\lambda X.F\, X \Rightarrow G\, X)\, f\, g = \mathsf{lam}\, \lambda x.\mathsf{map}^{\mathsf{P}}\, G\, f\, \big(g \cdot (\mathsf{map}^{\mathsf{N}}\, F\, f\, x)\big)\\
&\mathsf{map}^{\mathsf{N}}\ (\lambda X.F\, X \Rightarrow G\, X)\, f\, g = \mathsf{lam}\, \lambda x.\mathsf{map}^{\mathsf{N}}\, G\, f\, \big(g \cdot (\mathsf{map}^{\mathsf{P}}\, F\, f\, x)\big)
\end{aligned}$$

We illustrate the $\mathsf{map}^{\mathsf{P}}$-equation for $\Rightarrow$ by the following diagram.



Exercise 120. *Show that the following type operators are positive, negative or both.*

- $FX = X \Rightarrow \mathsf{Bool} \Rightarrow \mathsf{Bool}$
- $FX = (X \Rightarrow \mathsf{Bool}) \Rightarrow X$
- $FX = ((X \Rightarrow \mathsf{Bool}) \Rightarrow \mathsf{Bool}) \Rightarrow \mathsf{Bool}$
- $FX = (\mathsf{Bool} \Rightarrow \mathsf{Bool}) \Rightarrow X + X$
- $FX = (X \times (X + \top)) \Rightarrow \mathsf{Bool}$
- $FX = (\mathsf{Bool} \Rightarrow \mathit{Bool}) \Rightarrow \top$

Exercise 121. *Show that if the type operator $F\, X = X \Rightarrow \bot$ comes with a $\mathsf{map}^{\mathsf{P}}$ function, then $\mathsf{Tm}\, \bot$ has an element.*

Food for thought 122. *The following type operators are not positive, nor negative. What sort of contradiction do we get if they come equipped with $\mathsf{map}^{\mathsf{P}}$ or $\mathsf{map}^{\mathsf{N}}$ functions?*

- $FX = X \Rightarrow X$
- $FX = (\mathsf{Bool} \Rightarrow X) \Rightarrow X$

Exercise 123. *Assuming* Pos *and* Neg *are the initial* Pos–Neg *model (that is, two mutually defined meta inductive types), define $\mathsf{map}^{\mathsf{P}}$ and $\mathsf{map}^{\mathsf{N}}$ by iteration into a* Pos–Neg *model.*

Exercise 124. *Prove that $\mathsf{map}^{\mathsf{P}}$ is a covariant functor, while $\mathsf{map}^{\mathsf{N}}$ is contravariant:*

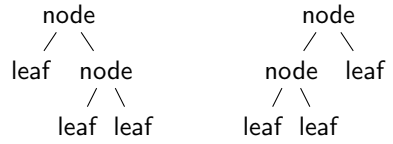
$$\begin{aligned}
&\mathsf{map}^{\mathsf{P}}\, F\, (\lambda x.f\, (g\, x))\, w = \mathsf{map}^{\mathsf{P}}\, F\, f\, (\mathsf{map}^{\mathsf{P}}\, F\, g\, w) \qquad &&\mathsf{map}^{\mathsf{P}}\, F\, (\lambda x.x)\, u = u\\
&\mathsf{map}^{\mathsf{N}}\, F\, (\lambda x.f\, (g\, x))\, w = \mathsf{map}^{\mathsf{N}}\, F\, g\, (\mathsf{map}^{\mathsf{N}}\, F\, f\, w) \qquad &&\mathsf{map}^{\mathsf{N}}\, F\, (\lambda x.x)\, u = u
\end{aligned}$$

3.5 Inductive types

Bool in Razor (Definition 14) and Nat in System T (Definition 76) were *inductive types*. Another example is the type of binary trees Tree which has two constructors:

$$\begin{aligned}
&\mathsf{Tree} : \mathsf{Ty}\\
&\mathsf{leaf} : \mathsf{Tm}\, \mathsf{Tree}\\
&\mathsf{node} : \mathsf{Tm}\, \mathsf{Tree} \to \mathsf{Tm}\, \mathsf{Tree} \to \mathsf{Tm}\, \mathsf{Tree}
\end{aligned}$$

The trees in Tree have binary branching and there is no information at the branching nodes or at the leaves. For example, the trees node leaf (node leaf leaf) and node (node leaf leaf) leaf are depicted as follows.



A Tree algebra (also called a Tree model inside our object language) is given by the following components:

$$A : \mathsf{Ty} \qquad u : \mathsf{Tm}\, A \qquad v : \mathsf{Tm}\, A \to \mathsf{Tm}\, A \to \mathsf{Tm}\, A,$$

A homomorphism from the algebra (Tree, leaf, node) to the algebra (A, u, v) is given by a function which respects the operations as follows:

$$\begin{aligned} &t : \mathsf{Tm}\,\mathsf{Tree} \to \mathsf{Tm}\, A \\ &t\,\mathsf{leaf} = u \\ &t\,(\mathsf{node}\, l\, r) = v\,(t\, l)\,(t\, r) \end{aligned}$$

The iterator of Tree says that for any algebra (A, u, v), there is such a homomorphism. The two equations that the homomorphism satisfies are called computation (β) rules. The uniqueness (η) rule says that every other homomorphism from (Tree, leaf, node) to (A, u, v) is equal to the iterator.

The two constructors of Tree can be merged into a single constructor along the following isomorphisms:

$$\begin{aligned} &\mathsf{Tm}\,\mathsf{Tree} \times (\mathsf{Tm}\,\mathsf{Tree} \to \mathsf{Tm}\,\mathsf{Tree} \to \mathsf{Tm}\,\mathsf{Tree}) && \cong \\ &(\mathsf{Tm}\,\top \to \mathsf{Tm}\,\mathsf{Tree}) \times (\mathsf{Tm}\,\mathsf{Tree} \times \mathsf{Tm}\,\mathsf{Tree} \to \mathsf{Tm}\,\mathsf{Tree}) && \cong \\ &\mathsf{Tm}\,(\top \Rightarrow \mathsf{Tree}) \times \mathsf{Tm}\,\big((\mathsf{Tree} \times \mathsf{Tree}) \Rightarrow \mathsf{Tree}\big) && \cong \\ &\mathsf{Tm}\,\Big((\top \Rightarrow \mathsf{Tree}) \times \big((\mathsf{Tree} \times \mathsf{Tree}) \Rightarrow \mathsf{Tree}\big)\Big) && \cong \\ &\mathsf{Tm}\,\big((\top + \mathsf{Tree} \times \mathsf{Tree}) \Rightarrow \mathsf{Tree}\big) && \cong \\ &\mathsf{Tm}\,(\top + \mathsf{Tree} \times \mathsf{Tree}) \to \mathsf{Tm}\,\mathsf{Tree} && = \\ &\mathsf{Tm}\,\big((\lambda X.\top + X \times X)\,\mathsf{Tree}\big) \to \mathsf{Tm}\,\mathsf{Tree} \end{aligned}$$

Definition 125 (Algebra of a type operator)**.** *Assume a type operator* $F : \mathsf{Ty} \to \mathsf{Ty}$*. An* F*-algebra consists of a type* $A : \mathsf{Ty}$ *and a function* $\mathsf{Tm}\,(F\,A) \to \mathsf{Tm}\,A$*.*

An inductive type consists of an F-algebra and an iterator. The type in the F-algebra is what we call the inductive type itself. The constructors of the inductive type are given by the function $\mathsf{Tm}\,(F\,A) \to \mathsf{Tm}\,A$ function in the F-algebra. The iterator encodes that there is a unique F-homomorphism from the inductive type into any other F-algebra. This is why the inductive type is also called the initial F-algebra. (It could be also called the syntax for F, but note that this syntax is internal to a model of our object theory.)

In order to obtain a well-defined notion of F-homomorphism, we need to restrict the type operator F to be positive. We will be even more restrictive, we will only allow strictly positive type operators. There are more strictly positive type operators than polynomial type operators, but less than positive type operators.

Food for thought 126. *Assuming that positive type operators have initial algebras is inconsistent with excluded middle.*

The Tree type is determined by the $\lambda X.\top + X \times X$ type operator, its constructor is

$$\mathsf{con} : \mathsf{Tm}\,(\top + \mathsf{Tree} \times \mathsf{Tree}) \to \mathsf{Tm}\,\mathsf{Tree},$$

its iterator and its computation rule are the following:

$$\begin{aligned} &\mathsf{ite} : \big(\mathsf{Tm}\,(\top + A \times A) \to \mathsf{Tm}\, A\big) \to \mathsf{Tm}\,\mathsf{Tree} \to \mathsf{Tm}\, A \\ &\mathsf{ite}\, f\,(\mathsf{con}\, w) = \mathsf{case}\, w\,\big(\lambda\_.f\,(\mathsf{inl}\,\mathsf{tt})\big)\,\Big(\lambda u.f\,\big(\mathsf{inr}\,\big(\mathsf{ite}\, f\,(\mathsf{fst}\, u), \mathsf{ite}\, f\,(\mathsf{snd}\, u)\big)\big)\Big) \end{aligned}$$

The following language contains an inductive type for all F strictly positive type operators. We express the equation for the homomorphism using map.

Definition 127 (Language with inductive types)**.** *A second-order model extends Fin (Definition 91) with the following components:*

$$\begin{aligned}
\mathsf{SPos} &: (\mathsf{Ty} \to \mathsf{Ty}) \to \mathsf{Set}\\
&: \mathsf{SPos}\,(\lambda X.X)\\
&: (A : \mathsf{Ty}) \to \mathsf{SPos}\,(\lambda_.A)\\
&: \mathsf{SPos}\,F \to \mathsf{SPos}\,G \to \mathsf{SPos}\,(\lambda X.F\,X + G\,X)\\
&: \mathsf{SPos}\,F \to \mathsf{SPos}\,G \to \mathsf{SPos}\,(\lambda X.F\,X \times G\,X)\\
&: (C : \mathsf{Ty}) \to \mathsf{SPos}\,F \to \mathsf{SPos}\,(\lambda X.C \Rightarrow F\,X)
\end{aligned}$$

We also have a map *operator:*

$$\begin{aligned}
&\mathsf{map} : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{SPos}\,F\} \to (f : \mathsf{Tm}\,A \to \mathsf{Tm}\,B) \to \mathsf{Tm}\,(F\,A) \to \mathsf{Tm}\,(F\,B)\\
&\mathsf{map}\,(\lambda X.X)\quad\quad f\,a = f\,a\\
&\mathsf{map}\,(\lambda_.C)\quad\quad f\,c = c\\
&\mathsf{map}\,(\lambda X.F\,X + G\,X)\;f\,w = \mathsf{case}\,w\,(\lambda u.\mathsf{inl}\,(\mathsf{map}\,F\,f\,u))\,(\lambda v.\mathsf{inr}\,(\mathsf{map}\,G\,f\,v))\\
&\mathsf{map}\,(\lambda X.F\,X \times G\,X)\;f\,w = \big(\mathsf{map}\,F\,f\,(\mathsf{fst}\,w), \mathsf{map}\,G\,f\,(\mathsf{snd}\,w)\big)\\
&\mathsf{map}\,(\lambda X.C \Rightarrow F\,X)\quad f\,g = \mathsf{lam}\,\lambda a.\mathsf{map}\,F\,f\,(g \cdot a)
\end{aligned}$$

And inductive types with their constructor, iterator, computation and uniqueness rules:

$$\begin{aligned}
&\mathsf{Ind} : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{SPos\,F}\} \to \mathsf{Ty}\\
&\mathsf{con} : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{SPos\,F}\} \to \mathsf{Tm}\,\big(F\,(\mathsf{Ind}\,F)\big) \to \mathsf{Tm}\,(\mathsf{Ind}\,F)\\
&\mathsf{ite} : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{SPos\,F}\} \to \big(\mathsf{Tm}\,(F\,A) \to \mathsf{Tm}\,A\big) \to \mathsf{Tm}\,(\mathsf{Ind}\,F) \to \mathsf{Tm}\,A\\
&\mathsf{Ind}\beta : \mathsf{ite}\,F\,f\,(\mathsf{con}\,F\,x) = f\,\big(\mathsf{map}\,F\,(\mathsf{ite}\,F\,f)\,x\big)\\
&\mathsf{Ind}\eta : (z : \mathsf{Tm}\,(\mathsf{Ind}\,F) \to \mathsf{Tm}\,A) \to \Big(\big(\lambda x.z\,(\mathsf{con}\,F\,x)\big) = \big(\lambda x.f\,(\mathsf{map}\,F\,z\,x)\big)\Big) \to z = \mathsf{ite}\,F\,f
\end{aligned}$$

Remark 128. *Usually η rules are omitted because (i) they are not needed for running programs; (ii) equality checking in the syntax becomes undecidable in the presence of η for natural numbers. This is not the case for η for finite types [Sch17].*

Inductive types are trees with finite depth. This means that they might have infinitely many branches, but if we follow a branch, then in finite steps we reach a leaf. Note that this does not mean that we can put an element of an inductive type into a hole of finite depth: there are infinitely branching trees where every branch is finite but it has branches of arbitrary length.

Exercise 129. *The inductive type* $\mathsf{Ind}\,(\lambda X.\top + (\mathsf{Nat} \Rightarrow X))$ *contains infinitely branching trees. Define a tree in which the i^{th} branch has height i.*

Exercise 130. *Natural numbers are given by* $\mathsf{Nat} := \mathsf{Ind}\,(\lambda X.\top + X)$*. Define* zero *and* suc *using* con*, define the iterator and prove the computation rules.*

Exercise 131. *Define the following inductive types: lists of natural numbers, binary trees with natural numbers at the leaves, binary trees with natural numbers at the nodes, ternary trees, untyped combinator calculus (without equations), Razor (without equations or types).*

Exercise 132. $\mathsf{Ind}\,F$ *is the fixpoint of the F function:* $F\,(\mathsf{Ind}\,F) \cong \mathsf{Ind}\,F$*.*

Exercise 133. *Show that for every A, we have* $\mathsf{Ind}\,(\lambda X.A) \cong A$*.*

The recursor of Nat can be defined via the iterator: we use the iterator in a way that it not only returns A, but also the concrete natural number on which it was called. For reference, we also list the rules for the iterator:

$$\begin{aligned}
&\mathsf{ite} : \{A : \mathsf{Ty}\} \to \mathsf{Tm}\,A \to (\mathsf{Tm}\,A \to \mathsf{Tm}\,A) \to \mathsf{Tm\,Nat} \to \mathsf{Tm}\,A\\
&\mathsf{ite}\beta_1 : \mathsf{ite}\,z\,s\,\mathsf{zero} = z\\
&\mathsf{ite}\beta_2 : \mathsf{ite}\,z\,s\,(\mathsf{suc}\,n) = s\,(\mathsf{ite}\,z\,s\,n)\\
&\mathsf{ite}\eta : (z : \mathsf{Tm\,Nat} \to \mathsf{Tm}\,A) \to (z\,\mathsf{zero} = z) \to ((n : \mathsf{Tm\,Nat}) \to z\,(\mathsf{suc}\,n) = s\,(z\,n)) \to z = \mathsf{ite}\,z\,s\\
&\mathsf{rec} : \{A : \mathsf{Ty}\} \to \mathsf{Tm}\,A \to (\mathsf{Tm\,Nat} \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A) \to \mathsf{Tm\,Nat} \to \mathsf{Tm}\,A\\
&\mathsf{rec}\,\{A\}\,z\,s\,n := \mathsf{snd}\,\Big(\mathsf{ite}\{\mathsf{Nat} \times A\}\,(\mathsf{zero}, z)\,\big(\lambda w.(\mathsf{suc}\,(\mathsf{fst}\,w), s\,(\mathsf{fst}\,w)\,(\mathsf{snd}\,w))\big)\,n\Big)
\end{aligned}$$

Exercise 134. *Prove the β rules of the recursor:*

$$\mathsf{rec}\beta_1 : \mathsf{rec}\, z\, s\, \mathsf{zero} = z$$
$$\mathsf{rec}\beta_2 : \mathsf{rec}\, z\, s\, (\mathsf{suc}\, n) = s\, n\, (\mathsf{rec}\, z\, s\, n)$$

For the latter we need the η law of the iterator (without this, $\mathsf{rec}\beta_2$ is not derivable [Par89]).

Food for thought 135. *Define the η rule of the recursor. Can it be derived from that of the iterator?*

Food for thought 136. *Define the recursor for an arbitrary inductive type and show that it can be derived from the iterator.*

If we only have inductive types for polynomial type operators, we obtain finitely branching trees.

Food for thought 137. *For every F polynomial type operator, $\mathsf{Ind}\, F$ is either isomorphic to a finite type or isomorphic to* Nat.

Exercise 138 (Cantor's diagonal argument)**.** *Show that if* $(\mathsf{Nat} \Rightarrow \mathsf{Bool}) \cong \mathsf{Nat}$*, then* $\mathsf{true} = \mathsf{false}$.

3.6 Coinductive types

Coinductive types are dual to inductive types.

- Inductive types are determined by their constructors, coinductive types by their destructors.
- The functions which we define out of inductive types using the iterator always *terminate*, the generators creating elements of coinductive types are always *productive*.
- Elements of inductive types are trees with finite depth, while coinductive types generate trees of potentially infinite depth.

Programs running forever (e.g. servers, operating systems, interactive programs), automata, Turing-machines and other machines are elements of coinductive types. They also model parts of object oriented programmign: a class is a coinductive type, the fields/member functions are the destructors, an object is an element of the coinductive type. In the 1990s Turner suggested strong functional programming [**?**] as a style where we separate inductive and coinductive types and there is no general recursion. No one has ever tried programming in such a setting.

The most well-known example of a coinductive type is the type of streams (lists of infinite length).

$$\begin{aligned}
&\mathsf{Stream} && : \mathsf{Ty} \to \mathsf{Ty}\\
&\mathsf{head} && : \mathsf{Tm}\, (\mathsf{Stream}\, A) \to \mathsf{Tm}\, A\\
&\mathsf{tail} && : \mathsf{Tm}\, (\mathsf{Stream}\, A) \to \mathsf{Tm}\, (\mathsf{Stream}\, A)\\
&\mathsf{gen} && : (\mathsf{Tm}\, C \to \mathsf{Tm}\, A) \to (\mathsf{Tm}\, C \to \mathsf{Tm}\, C) \to \mathsf{Tm}\, C \to \mathsf{Tm}\, (\mathsf{Stream}\, A)\\
&\mathsf{Stream}\beta_1 && : \mathsf{head}\, (\mathsf{gen}\, h\, t\, c) = h\, c\\
&\mathsf{Stream}\beta_2 && : \mathsf{tail}\, (\mathsf{gen}\, h\, t\, c) = \mathsf{gen}\, h\, t\, (t\, c)
\end{aligned}$$

Streams have two destructors: head and tail. The tail can be applied any number of times to the stream, so indeed this gives us infinite number of $\mathsf{Tm}\, A$s from a $u : \mathsf{Tm}\, (\mathsf{Stream}\, A)$:

$$\mathsf{head}\, u,\ \ \mathsf{head}\, (\mathsf{tail}\, u),\ \ \mathsf{head}\, (\mathsf{tail}\, (\mathsf{tail}\, u)),\ \ \mathsf{head}\, (\mathsf{tail}\, (\mathsf{tail}\, (\mathsf{tail}\, u))),\ \ \ldots$$

Streams can be made using the generator (coiterator, consructor, corecursor): the input of the generator is a type of seeds (C), functions h and t and a seed (an element of $\mathsf{Tm}\, C$) from which the stream grows out. The h tells us how to compute the current head element from the seed, while t says how to advance the seed (make a step): this is what the β rules express.

The stream containing the natural numbers $0, 1, 2, 3, \ldots$ is given by the following term where $C := \mathsf{Nat}$:

$$\begin{aligned}
&\mathsf{nats} : \mathsf{Tm}\, (\mathsf{Stream}\, \mathsf{Nat})\\
&\mathsf{nats} := \mathsf{gen}\, (\lambda n.n)\, \mathsf{suc}\, \mathsf{zero}
\end{aligned}$$

Indeed,

$$
\begin{aligned}
&\mathsf{head}\ \mathsf{nats} && = \\
&\mathsf{head}\ (\mathsf{gen}\ (\lambda n.n)\ \mathsf{suc}\ \mathsf{zero}) && =(\mathsf{Stream}\beta_1) \\
&(\lambda n.n)\ \mathsf{zero} && = \\
&\mathsf{zero}, \\
&\mathsf{head}\ (\mathsf{tail}\ \mathsf{nats}) && = \\
&\mathsf{head}\ (\mathsf{tail}\ (\mathsf{gen}\ (\lambda n.n)\ \mathsf{suc}\ \mathsf{zero})) && =(\mathsf{Stream}\beta_2) \\
&\mathsf{head}\ (\mathsf{gen}\ (\lambda n.n)\ \mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})) && =(\mathsf{Stream}\beta_1) \\
&(\lambda n.n)\ (\mathsf{suc}\ \mathsf{zero}) && = \\
&\mathsf{suc}\ \mathsf{zero}, \\
&\mathsf{head}\ (\mathsf{tail}\ (\mathsf{tail}\ \mathsf{nats})) && = \\
&\mathsf{head}\ (\mathsf{tail}\ (\mathsf{tail}\ (\mathsf{gen}\ (\lambda n.n)\ \mathsf{suc}\ \mathsf{zero}))) && =(\mathsf{Stream}\beta_2) \\
&\mathsf{head}\ (\mathsf{tail}\ (\mathsf{gen}\ (\lambda n.n)\ \mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero}))) && =(\mathsf{Stream}\beta_2) \\
&\mathsf{head}\ (\mathsf{gen}\ (\lambda n.n)\ \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero}))) && =(\mathsf{Stream}\beta_1) \\
&(\lambda n.n)\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})) && = \\
&\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero}), &&
\end{aligned}
$$

and so on.

Exercise 139. *Define pointwise addition on two* Stream Nat*s, and define the* map *function on streams.*

We cannot define the filter function on streams because it is not necessarily productive.

Every strictly positive type operator determines a coinductive type. In the following, we extend the language of inductive type with coinductive types (actually we don't need inductive types, only SPos and map).

Definition 140 (Language with inductive and coinductive types)**.** *A second-order model extends Definition 127 with the following components.*

$$
\begin{aligned}
&\mathsf{Coind} && : \mathsf{SPos}\,\mathsf{F} \to \mathsf{Ty} \\
&\mathsf{des} && : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{SPos}\,\mathsf{F}\} \to \mathsf{Tm}\,(\mathsf{Coind}\,F) \to \mathsf{Tm}\,\big(F\,(\mathsf{Coind}\,F)\big) \\
&\mathsf{gen} && : (F : \mathsf{Ty} \to \mathsf{Ty}) \to \{\mathsf{SPos}\,\mathsf{F}\} \to \big(\mathsf{Tm}\,A \to \mathsf{Tm}\,(F\,A)\big) \to \mathsf{Tm}\,A \to \mathsf{Tm}\,(\mathsf{Coind}\,F) \\
&\mathsf{Coind}\beta && : \mathsf{des}\,F\,(\mathsf{gen}\,F\,f\,a) = \mathsf{map}\,F\,(\mathsf{gen}\,F\,f)\,(f\,a) \\
&\mathsf{Coind}\eta && : (z : \mathsf{Tm}\,A \to \mathsf{Tm}\,(\mathsf{Coind}\,F)) \to \Big(\big(\lambda z.\mathsf{des}\,F\,(z\,a)\big) = \big(\lambda z.\mathsf{map}\,F\,z\,(f\,a)\big)\Big) \to z = \mathsf{gen}\,F\,f
\end{aligned}
$$

Exercise 141. *Show that the destructors and the generator of streams can be given using* $\mathsf{Stream}\,A := \mathsf{Coind}\,(\lambda X.\,A \times X)$.

Exercise 142. *Define addition and multiplication on conatural numbers (*$\mathsf{Coind}\,(\lambda X.ldotp\top + X)$*)!.*

Exercise 143. *Which inductive type is the dual of streams?*

Exercise 144. *What is the dual of lists, that is,* $\mathsf{Coind}\,(\lambda X.\,\top + A \times X)$*?*

Exercise 145. *Define the coinductive type of machines which can receive a natural number, they can print out a number and there is a button on them. Using the generator, define an summation machine, which adds all the input numbers, returns the sum when asked, and the button resets the number to 0. Define another machine in the same type which prints the maximum of all the input numbers.*

Exercise 146. *A* $\mathsf{Coind}\,F$ *típus az* F *operátor fixpontja: mutassuk meg, hogy* $F\,(\mathsf{Coind}\,F) \cong \mathsf{Coind}\,F$.

Exercise 147. *Show* $\mathsf{Coind}\,(\lambda X.\,A) \cong A$ *for every* A.

Food for thought 148. *The dual of iterator is generator. What is the dual of the recursor?*

3.7 Polymorphism

In the above languages we had to separately define function composition $(B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$ for each types A, B and C; we had to separately define the $\mathsf{filter} : \mathsf{Tm}\,((A \Rightarrow \mathsf{Bool}) \Rightarrow \mathsf{List}\,A \Rightarrow \mathsf{List}\,A)$ function for each type A. We introduce polymorphic types in order to define these terms once and for all for every type. The type of general function composition becomes $\forall\lambda A.\forall\lambda B.\forall\lambda C.(B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$, the type of filter becomes $\forall\lambda A.(A \Rightarrow \mathsf{Bool}) \Rightarrow \mathsf{List}\,A \Rightarrow \mathsf{List}\,A$.

3.7.1 Hindley–Milner

In the following type system we distinguish monorphic (MTy) and polymorphic types (Ty), and the universal quantifier can only appear at the front (at the top of) a polymorphic type. Base types such as Nat are monomorphic.

Definition 149 (A Hindley–Milner style language)**.** *A second-order model contains the following components.*

$$\begin{array}{ll}
\mathsf{MTy} & : \mathsf{Set} \\
\mathsf{Ty} & : \mathsf{Set} \\
\mathsf{Tm} & : \mathsf{Ty} \to \mathsf{Set} \\
- \Rightarrow - & : \mathsf{MTy} \to \mathsf{MTy} \to \mathsf{MTy} \\
\forall & : (\mathsf{MTy} \to \mathsf{Ty}) \to \mathsf{Ty} \\
\mathsf{i} & : \mathsf{MTy} \to \mathsf{Ty} \\
\mathsf{lam} & : \big(\mathsf{Tm}\,(\mathsf{i}\,A) \to \mathsf{Tm}\,(\mathsf{i}\,B)\big) \cong \mathsf{Tm}\,(\mathsf{i}\,(A \Rightarrow B)) : - \cdot - \\
\mathsf{Lam} & : \big((A : \mathsf{MTy}) \to \mathsf{Tm}\,(B\,A)\big) \cong \mathsf{Tm}\,(\forall\,B) : - \bullet -
\end{array}$$

In the above language we have not only term variables, but type variables as well: the operators $\forall$ and Lam bind type variables.

The polymorphic identity function is defined as follows.

$$\begin{array}{l}
\mathsf{id} : \mathsf{Tm}\,\big(\forall\,\lambda A.\mathsf{i}\,(A \Rightarrow A)\big) \\
\mathsf{id} := \mathsf{Lam}\,\lambda A.\mathsf{lam}\,\lambda a.a
\end{array}$$

Although this is not immediately apparent, the above function uses the A input as well because the implicit parameters of lam refer to it. The explicit definition is the following.

$$\mathsf{id} := \mathsf{Lam}\,\{\lambda A.\mathsf{i}\,(A \Rightarrow A)\}\,\big(\lambda A.\mathsf{lam}\,\{A\}\,\{A\}\,(\lambda a.a)\big)$$

Assuming we extend the above language with $\mathsf{Bool} : \mathsf{Ty}$ and $\mathsf{true} : \mathsf{Tm}\,\mathsf{Bool}$ operators, we can run the id program with those inputs as follows.

$$\begin{array}{ll}
\mathsf{id} \bullet \mathsf{Bool} \cdot \mathsf{true} & = \\
(\mathsf{Lam}\,\lambda A.\mathsf{lam}\,\{A\}\,\{A\}\,\lambda a.a) \bullet \mathsf{Bool} \cdot \mathsf{true} & =(\forall\beta) \\
\big((\lambda A.\mathsf{lam}\,\{A\}\,\{A\}\,\lambda a.a)\,\mathsf{Bool}\big) \cdot \mathsf{true} & = \\
(\mathsf{lam}\,\{\mathsf{Bool}\}\,\{\mathsf{Bool}\}\,\lambda a.a) \cdot \mathsf{true} & =(\Rightarrow\beta) \\
(\lambda a.a)\,\mathsf{true} & = \\
\mathsf{true} &
\end{array}$$

Exercise 150. *Define the following programs (derive them in any second-order model).*

$$\begin{array}{ll}
\mathsf{const} & : \mathsf{Tm}\,\big(\forall\lambda A.\forall\lambda B.\mathsf{i}\,(A \Rightarrow B \Rightarrow A)\big) \\
\mathsf{comp} & : \mathsf{Tm}\,\Big(\forall\lambda A.\forall\lambda B.\forall\lambda C.\mathsf{i}\,\big((B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C\big)\Big) \\
\mathsf{flip} & : \mathsf{Tm}\,\Big(\forall\lambda A.\forall\lambda B.\forall\lambda C.\mathsf{i}\,\big((A \Rightarrow B \Rightarrow C) \Rightarrow B \Rightarrow A \Rightarrow C\big)\Big)
\end{array}$$

This language does not support (co)inductive types, we have to put them in by hand. Haskell was originally designed as Hindley-Milner extended with a fixpoint operator and recursive types (see next section).

A Hindley–Milner type in general looks like this:

$$\forall\,\lambda A_1.\forall\,\lambda A_2.\ldots.\forall\,\lambda A_n.\mathsf{i}\,(\underbrace{\ldots\ldots\ldots\ldots\ldots\ldots}_{\text{there is no } \forall \text{ in here}})$$

The type $\mathsf{Bool} \Rightarrow \forall\lambda A.\,A$ does not make sense in this language, the $\forall$ has to be in the front. The Hindley-Milner type system is very convenient for programming because types never have to be written out.

Food for thought 151. *Define a type inference algorithm for Hindley–Milner which takes preterms without typing information and produces well-typed terms which have the most general type (E.g. for* const *we obtain* $\forall\lambda A.\,\forall\lambda B.\,A \Rightarrow B \Rightarrow A$ *instead of* $\forall\lambda A.\,A \Rightarrow A \Rightarrow A$*). Preterms do not contain* Lam *or* $- \bullet -$*, these are inserted by the type inference algorithm. See [Hin69, Mil78, Csö12].*

3.7.2 System F

In the next language, $\forall$ can appear anywhere, not only at the front of a type. It was discovered independently by Jean-Yves Girard [Gir72] and John C. Reynolds [Rey74], the former called it System F, the latter called it polymorphic lambda calculus.

Definition 152 (System F)**.** *A second-order model contains the following components:*

$$\begin{array}{ll} \mathsf{Ty} : \mathsf{Set} & \mathsf{Tm} : \mathsf{Ty} \to \mathsf{Set} \\ - \Rightarrow - : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} & \mathsf{lam} : (\mathsf{Tm}\, A \to \mathsf{Tm}\, B) \cong \mathsf{Tm}\,(A \Rightarrow B) : - \cdot - \\ \forall : (\mathsf{Ty} \to \mathsf{Ty}) \to \mathsf{Ty} & \mathsf{Lam} : ((X : \mathsf{Ty}) \to \mathsf{Tm}\,(A\,X)) \cong \mathsf{Tm}\,(\forall\, A) : - \bullet - \end{array}$$

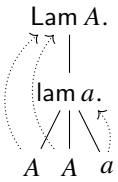
For the benefit of the reader, we also list the operations in concice derivation rule style (we don't write Ty and Tm anywhere):

$$\frac{A \qquad B}{A \Rightarrow B} \qquad \frac{x : A \vdash b : B}{\mathsf{lam}\, x.b : A \Rightarrow B} \qquad \frac{f : A \Rightarrow B \qquad a : A}{f \cdot a : B} \qquad \frac{}{(\mathsf{lam}\, x.b) \cdot a = b[x \mapsto a]} \qquad \frac{f : A \Rightarrow B}{f = \mathsf{lam}\, x.f \cdot x}$$

$$\frac{X \vdash A}{\forall X.A} \qquad \frac{X \vdash a : A}{\mathsf{Lam}\, X.a : \forall X.A} \qquad \frac{f : \forall X.A \qquad C}{f \bullet C : A[X \mapsto C]} \qquad \frac{}{(\mathsf{Lam}\, X.a) \bullet C = a[X \mapsto C]} \qquad \frac{f : \forall X.A}{f = \mathsf{Lam}\, X.f \bullet X}$$

Derivation of the polymorphic identity function (we made the implicit arguments of lam explicit):

$$\frac{\dfrac{\overline{A, a : A \vdash a : A}}{A \vdash \mathsf{lam}\,\{A\}\,\{A\}\,(a.a) : A \Rightarrow A}}{\mathsf{Lam}\, A.\, \mathsf{lam}\,\{A\}\,\{A\}\,(a.a) : \forall A.\, A \Rightarrow A}$$



Exercise 153. *Define the following programs (derive them in any second-order model).*

$$\begin{array}{ll} \mathsf{const} & : \mathsf{Tm}\,(\forall \lambda A.\, \forall \lambda B.\, A \Rightarrow B \Rightarrow A) \\ \mathsf{comp} & : \mathsf{Tm}\,\big(\forall \lambda A.\, \forall \lambda B.\, \forall \lambda C.\, (B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C\big) \\ \mathsf{comp}' & : \mathsf{Tm}\,\big(\forall \lambda B.\, \forall \lambda C.\, (B \Rightarrow C) \Rightarrow \forall \lambda A.\, (A \Rightarrow B) \Rightarrow A \Rightarrow C\big) \\ \mathsf{S} & : \mathsf{Tm}\,\big(\forall \lambda A.\, \forall \lambda B.\, \forall \lambda C.\, (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C\big) \\ \mathsf{flip} & : \mathsf{Tm}\,\big(\forall \lambda A.\, \forall \lambda B.\, \forall \lambda C.\, (A \Rightarrow B \Rightarrow C) \Rightarrow B \Rightarrow A \Rightarrow C\big) \end{array}$$

Exercise 154. *Show that any second-order model of System F gives rise to a second-order model of Hindley-Milner.*

Remark 155. *To rule out trivial solutions to the previous exercise, redo it inside the theory of signatures (see Section 3.12): define a morphism of from the signature of System F to the signature of Hindley–Miler; this also gives rise to a functor between the categories of models.*

All inductive and coiductive types described in Section 3.5, 3.6 are already supported by System F (with their β rules but without η rules). This method is called *Church encoding*, it is a generalisation of how Alonzo Church encoded natural numbers in the untyped lambda calculus (see the next section). The two-element type is defined as follows.

$$\begin{array}{ll} \mathsf{Bool} & := \forall \lambda A.\, A \Rightarrow A \Rightarrow A \\ \mathsf{true} & := \mathsf{Lam}\, \lambda A.\, \mathsf{lam}\, \lambda t.\, \mathsf{lam}\, \lambda f.\, t \\ \mathsf{false} & := \mathsf{Lam}\, \lambda A.\, \mathsf{lam}\, \lambda t.\, \mathsf{lam}\, \lambda f.\, f \\ \mathsf{ite} & : \mathsf{Tm}\,\mathsf{Bool} \to \mathsf{Tm}\,(\forall\, \lambda A.\, A \Rightarrow A \Rightarrow A) \\ \mathsf{ite} & := \lambda b.\, b \\ \mathsf{Bool}\beta_1 & : \mathsf{ite}\,\mathsf{true} \bullet A \cdot t \cdot f = (\lambda b.\, b)\,\mathsf{true} \bullet A \cdot t \cdot f = \mathsf{true} \bullet A \cdot t \cdot f = (\mathsf{Lam}\, \lambda A.\, \mathsf{lam}\, \lambda t.\, \mathsf{lam}\, \lambda f.\, t) \bullet A \cdot t \cdot f = t \\ \mathsf{Bool}\beta_2 & : \mathsf{ite}\,\mathsf{false} \bullet A \cdot t \cdot f = (\lambda b.\, b)\,\mathsf{false} \bullet A \cdot t \cdot f = \mathsf{false} \bullet A \cdot t \cdot f = (\mathsf{Lam}\, \lambda A.\, \mathsf{lam}\, \lambda t.\, \mathsf{lam}\, \lambda f.\, f) \bullet A \cdot t \cdot f = f \end{array}$$

The intuition is that a term in the above Bool type can only do two things: return its first or the second argument. In particular, it cannot look into the type A, it has to work in a uniform (parametric [Rey83, Wad89], natural [Awo10]) way for all As.

Note that Bool cannot be encoded in Hindley–Milner.

Exercise 156. *We defined the type of itertor above so that it can be easily implemented. Show that its type is isomorphic to the type of the usual if-then-else operator:*

$$\big(\mathsf{Tm}\,\mathsf{Bool} \to \mathsf{Tm}\,(\forall\,\lambda A.A \Rightarrow A \Rightarrow A)\big) \cong \big(\{A : \mathsf{Ty}\} \to \mathsf{Tm}\,\mathsf{Bool} \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A \to \mathsf{Tm}\,A\big).$$

Church encoding of a few more types:

empty	$\bot$	$\forall\,\lambda C.C$
unit	$\top$	$\forall\,\lambda C.C \Rightarrow C$
three-element		$\forall\,\lambda C.C \Rightarrow C \Rightarrow C \Rightarrow C$
sum	$A + B$	$\forall\,\lambda C.(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$
product	$A \times B$	$\forall\,\lambda C.(A \Rightarrow B \Rightarrow C) \Rightarrow C$
natural numbers	Nat	$\forall\,\lambda C.C \Rightarrow (C \Rightarrow C) \Rightarrow C$
list of As	$\mathsf{List}\,A$	$\forall\,\lambda C.C \Rightarrow (A \Rightarrow C \Rightarrow C) \Rightarrow C$
binary trees	Tree	$\forall\,\lambda C.C \Rightarrow (C \Rightarrow C \Rightarrow C) \Rightarrow C$

Exercise 157. *Derive the constructors and iterators for the above types, show that their computation rules hold.*

Food for thought 158. *Show that any inductive type encoded by a strictly positive functor can be derived in System F (without η rule).*

We can encode the other quantifier:

$$\begin{aligned}&\exists : (\mathsf{Ty} \to \mathsf{Ty}) \to \mathsf{Ty}\\ &\exists\,F := \forall\,\lambda C.(\forall\,\lambda X.F\,X \Rightarrow C) \Rightarrow C\end{aligned}$$

Exercise 159. *Derive the constuctor, destructor and β rule:*

$$\begin{aligned}&\mathsf{mk}\exists : (X : \mathsf{Ty}) \to \mathsf{Tm}\,(F\,X) \to \mathsf{Tm}\,(\exists\,F)\\ &\mathsf{case} : ((X : \mathsf{Ty}) \to \mathsf{Tm}\,(F\,X) \to \mathsf{Tm}\,C) \to \mathsf{Tm}\,(\exists\,F) \to \mathsf{Tm}\,C\\ &\exists\beta \quad : \mathsf{case}\,w\,(\mathsf{mk}\exists\,X\,f) = w\,X\,f\end{aligned}$$

We exemplify the encoding of coinductive types through streams. A stream is given by a type C, a coalgebra on C and an element of C called the seed rfom which the stream grows out.

$$\begin{aligned}&\mathsf{Stream} : \mathsf{Ty} \to \mathsf{Ty}\\ &\mathsf{Stream}\,A := \exists\,\lambda C.\big((C \Rightarrow A) \times (C \Rightarrow C)\big) \times C\\ &\mathsf{head} : \mathsf{Tm}\,(\mathsf{Stream}\,A) \to \mathsf{Tm}\,A\\ &\mathsf{head} := \mathsf{case}\,\big(\lambda X\,w.\mathsf{fst}\,(\mathsf{fst}\,w) \cdot \mathsf{snd}\,w\big)\\ &\mathsf{tail} : \mathsf{Tm}\,(\mathsf{Stream}\,A) \to \mathsf{Tm}\,(\mathsf{Stream}\,A)\\ &\mathsf{tail} := \mathsf{case}\,\Big(\lambda X\,w.\mathsf{mk}\exists\,X\,\big(\mathsf{fst}\,w, \mathsf{snd}\,(\mathsf{fst}\,w) \cdot \mathsf{snd}\,w\big)\Big)\\ &\mathsf{gen} : \{C : \mathsf{Ty}\} \to (\mathsf{Tm}\,C \to \mathsf{Tm}\,A) \to (\mathsf{Tm}\,C \to \mathsf{Tm}\,C) \to \mathsf{Tm}\,C \to \mathsf{Tm}\,(\mathsf{Stream}\,A)\\ &\mathsf{gen}\,\{C\}\,h\,t\,c := \mathsf{mk}\exists\,C\,\big((\mathsf{lam}\,h, \mathsf{lam}\,t), c\big)\end{aligned}$$

Exercise 160. *Prove the computation rules of Church-encoded streams:*

$$\begin{aligned}&\mathsf{Stream}\beta_1 : \mathsf{head}\,(\mathsf{gen}\,h\,t\,c) = h\,c\\ &\mathsf{Stream}\beta_2 : \mathsf{tail}\,(\mathsf{gen}\,h\,t\,c) = \mathsf{gen}\,h\,t\,(t\,c)\end{aligned}$$

Food for thought 161. *Show that any coinductive type encoded by a strictly positive functor can be derived in System F (without η rule).*

Food for thought 162. *If we have internal parametricity [ACKS24, Lor25], the η rules also hold [Wad90]. Can we extend System F with internal parametricity without adding dependent types?*

Abstract types and interfaces can be encoded using existential types [MP88], and thus in System F. For example, if S is a type of Nat-stacks, then we want the following features:

empty	$: \mathsf{Tm}\,S$	create an empty stack
push	$: \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow S \Rightarrow S)$	put a Nat on the top of the stack
pop	$: \mathsf{Tm}\,(S \Rightarrow S)$	remove an element from the top of the stack
top	$: \mathsf{Tm}\,(S \Rightarrow \mathsf{Nat})$	look at what is in the top of the stack
$\mathsf{isEmpty}$	$: \mathsf{Tm}\,(S \Rightarrow \mathsf{Bool})$	let us know if the stack is empty

(Probably top should have type $\mathsf{Tm}\,(S \Rightarrow (\mathsf{Nat}+\top))$, but that is not the point here.) A program which computes a Nat using an abstract stack interface has the following type:

$$\big(\exists\lambda S.\, S \times (\mathsf{Nat} \Rightarrow S \Rightarrow S) \times (S \Rightarrow S) \times (S \Rightarrow \mathsf{Nat}) \times (S \Rightarrow \mathsf{Bool})\big) \Rightarrow \mathsf{Nat}.$$

Exercise 163. *Define a term of the above type which creates a stack, then pushes* 1, 2, 3 *on it, pops one element, and returns what is on the top of the stack. Show that if the following equations hold for our stack implementation, the result will be* 2*:*

$$\mathsf{top} \cdot (\mathsf{pop} \cdot (\mathsf{push} \cdot n \cdot (\mathsf{push} \cdot m \cdot s))) = m.$$

This program behaves uniformly for any stack implementation. Of course, there can be trivial stack implementations which don't behave well. If we want to statically restrict our stack implementation to correct ones, we need dependent types (Section 3.9). Otherwise, we can just assume that the stack implementation behaves well.

Exercise 164. *How do you describe with equations such as the above that a stack implementation behaves correctly?*

3.7.3 Kinds

TODO: translate.

System F-ben vannak polimorf függvények, de nincsenek típus szintű függvények. Például a listák nem adhatók meg, ami egy típusról típusra képező függvény. A következő rendszer ezt kijavítja. A típusokon és a termeken túl van egy újabb szortunk, a fajták (Kind-ok) szortja, a típusok pedig indexelve vannak az ő "típusukkal" (fajtájukkal). A $*$ fajtájú típusok a tulajdonképpeni típusok, nekik lehetnek termei. A $* \Rrightarrow *$ fajtájú típusok a típus-függvények, melyek egy tulajdonképpeni típusból képeznek egy tulajdonképpeni típusba.

Definition 165 (System F_ω)**.**

$$\begin{array}{ll}
\mathsf{Kind} & : \mathsf{Set} \\
\mathsf{Ty} & : \mathsf{Kind} \to \mathsf{Set} \\
- \Rrightarrow - & : \mathsf{Kind} \to \mathsf{Kind} \to \mathsf{Kind} \\
\mathsf{LAM} & : (\mathsf{Ty}\,K \to \mathsf{Ty}\,L) \cong \mathsf{Ty}\,(K \Rrightarrow L) : - \bullet - \\
* & : \mathsf{Kind} \\
\mathsf{Tm} & : \mathsf{Ty} * \to \mathsf{Set} \\
\forall & : (\mathsf{Ty}\,K \to \mathsf{Ty} *) \to \mathsf{Ty} * \\
\mathsf{Lam} & : ((X : \mathsf{Ty}\,K) \to \mathsf{Tm}\,(A\,X)) \cong \mathsf{Tm}\,(\forall\,A) : - \bullet - \\
- \Rightarrow - & : \mathsf{Ty} * \to \mathsf{Ty} * \to \mathsf{Ty} * \\
\mathsf{lam} & : (\mathsf{Tm}\,A \to \mathsf{Tm}\,B) \cong \mathsf{Tm}\,(A \Rightarrow B) : - \cdot -
\end{array}$$

A fajták és típusok a $\Rrightarrow$-val úgy viselkednek, mint egyszerű típuselméletben a típusok és a termek. A System F_ω érdekessége, hogy az egyik szort felhasznál egy operátort ($*$-ot), tehát a szortok nem adhatók meg az operátoroktól elkülönülten.

A lista típus a következőképp Church-kódolható:

$$\begin{array}{l}
\mathsf{List} : \mathsf{Ty}\,(* \Rrightarrow *) \\
\mathsf{List} := \mathsf{LAM}\,\lambda A.\forall\,\lambda B.B \Rightarrow (A \Rightarrow B \Rightarrow B) \Rightarrow B
\end{array}$$

A Haskell bind operátorának a következő a típusa:

$$\forall\,\lambda M.\forall\,\lambda A.\forall\,\lambda B.(A \Rightarrow M \bullet B) \Rightarrow M \bullet A \Rightarrow M \bullet B$$

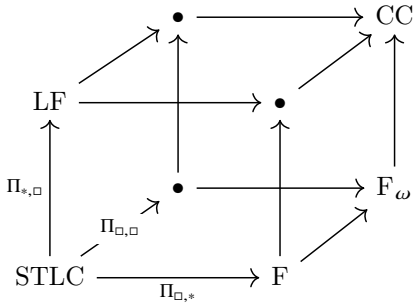
A λ-kat kihagyva és a $\forall$ által kötött típusok fajtáit odaírva a következőt kapjuk:

$$\forall(M : * \Rrightarrow *).\forall(A : *).\forall(B : *).(A \Rightarrow M \bullet B) \Rightarrow M \bullet A \Rightarrow M \bullet B$$

Exercise 166. *Definiáljuk a Church-kódolt lista típus konstruktorait és iterátorát (fold operátorát)!*

Exercise 167. *Adjuk meg a Church-kódolt lista típus return és bind operátorait és vizsgáljuk meg, hogy teljesítik-e a monád törvényeket!*

The following definition shows that all the languages in the lambda cube [Bar91] can be given as SOGATs. The simply typed lambda calculus (STLC) only includes $\Pi_{*,*}$, and the edges in each dimension add one of the other three Π types, respectively. The calculus of constructions (CC) includes all four Π types.



We don't give names to the maps in the universal properties.

Definition 168 (CC)**.**

$$
\begin{array}{lll}
\square & : \mathsf{Set} & \\
\mathsf{Ty} & : \square \to \mathsf{Set} & \\
* & : \square & \\
\mathsf{Tm} & : \mathsf{Ty}\, * \to \mathsf{Set} & \\
\Pi_{*,*} & : (A : \mathsf{Ty}\, *) \to (\mathsf{Tm}\, A \to \mathsf{Ty}\, *) \to \mathsf{Ty}\, * & \mathsf{Tm}\,(\Pi_{*,*}\, A\, B) \cong (a : \mathsf{Tm}\, A) \to \mathsf{Tm}\,(B\, a) \\
\Pi_{*,\square} & : (A : \mathsf{Ty}\, *) \to (\mathsf{Tm}\, A \to \square) \to \square & \mathsf{Ty}\,(\Pi_{*,\square}\, A\, L) \cong (a : \mathsf{Tm}\, A) \to \mathsf{Ty}\,(L\, a) \\
\Pi_{\square,*} & : (K : \square) \to (\mathsf{Ty}\, K \to \mathsf{Ty}\, *) \to \mathsf{Ty}\, * & \mathsf{Tm}\,(\Pi_{\square,*}\, K\, B) \cong (A : \mathsf{Ty}\, K) \to \mathsf{Tm}\,(B\, A) \\
\Pi_{\square,\square} & : (K : \square) \to (\mathsf{Ty}\, K \to \square) \to \square & \mathsf{Ty}\,(\Pi_{\square,\square}, K\, L) \cong (A : \mathsf{Ty}\, K) \to \mathsf{Ty}\,(L\, A)
\end{array}
$$

Food for thought 169. *Show that all pure type systems are definable as SOGATs. See [CD07].*

3.8 Fixpoint operator for types and terms, untyped calculi

If we have System F-style polymorphism, we don't need to add (co)inductive types by hand into our language. There is another very common way to add features to our language: adding noise, or in other words, making the language less safe. On the other hand, these languages are Turing-complete.

S. 3.8.1 We can add a fixpoint-operator to our language and then we do not need iterators anymore, but we lose that every program terminates: we add meaningless recursive programs which simply loop. The language which has Nat without iterator but with a fixpoint operator is called PCF.

S. 3.8.2 We can add recursive types by saying that every type operator has a fixpoint: we lose the distinction between inductive and coinductive types and now not only terms, but types contain rubbish.

S. 3.8.3 We can remove all types, then we don't even need to add a fixpoint operator, we can just define it. This is the untyped lambda calculus. We have no guarantee that programs are meaningful, but we can implement anything.

3.8.1 PCF

A *fixpoint-operator* is given by $\mathsf{fix} : (\mathsf{Tm}\, A \to \mathsf{Tm}\, A) \to \mathsf{Tm}\, A$ which satisfies the equation $\mathsf{fix}\, f = f\,(\mathsf{fix}\, f)$.

Definition 170 (PCF)**.** *A second-order model of PCF has two sorts (types and type-indexed terms) and the following additional components using concise derivation rule notation:*

$$\frac{A \qquad B}{A \Rightarrow B} \qquad \frac{x : A \vdash b : B}{\mathsf{lam}\, x.b : A \Rightarrow B} \qquad \frac{f : A \Rightarrow B \qquad a : A}{f \cdot a : B} \qquad \frac{}{(\mathsf{lam}\, x.b) \cdot a = b[x \mapsto a]} \qquad \frac{x : A \vdash t : A}{\mathsf{fix}\, x.t : A} \qquad \frac{}{\mathsf{fix}\, x.t = t[x \mapsto \mathsf{fix}\, x.t]}$$

$$\frac{}{\mathsf{Nat}} \qquad \frac{}{\mathsf{zero} : \mathsf{Nat}} \qquad \frac{n : \mathsf{Nat}}{\mathsf{suc}\, n : \mathsf{Nat}} \qquad \frac{b : \mathsf{Nat} \qquad t : A \qquad f : A}{\mathsf{ifZero}\, b\, t\, f : A} \qquad \frac{}{\mathsf{ifZero}\, \mathsf{zero}\, t\, f = t} \qquad \frac{}{\mathsf{ifZero}\,(\mathsf{suc}\, n)\, t\, f = f}$$

$$\frac{n : \mathsf{Nat}}{\mathsf{pred}\, n : \mathsf{Nat}} \qquad \frac{}{\mathsf{pred}\,(\mathsf{suc}\, n) = n} \qquad \frac{}{\mathsf{pred}\, \mathsf{zero} = \mathsf{zero}}$$

Example 171. *We want to define the $x \mapsto x * 2 + 1$ function in a way that it satisfies the following equation:*

$$f = \mathsf{lam}\,\lambda n.\,\mathsf{ifZero}\,n\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(f \cdot \mathsf{pred}\,n))).$$

This cannot be seen as a definition because f appears on the right hand side of the equation. We fix this by abstracting over the recursive call:

$$g : \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow \mathsf{Nat}) \to \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow \mathsf{Nat})$$
$$g\,f := \mathsf{lam}\,\lambda n.\,\mathsf{ifZero}\,n\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(f \cdot \mathsf{pred}\,n)))$$

The fixpoint of g satisfies the desired equation:

$$\begin{array}{ll}
\mathsf{fix}\,g : \mathsf{Tm}\,(\mathsf{Nat} \Rightarrow \mathsf{Nat}) & \\
\mathsf{fix}\,g & =(\beta \textit{ rule for } \mathsf{fix}) \\
g\,(\mathsf{fix}\,g) & =(\textit{definition of } g) \\
\mathsf{lam}\,\lambda n.\,\mathsf{ifZero}\,n\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,n))) &
\end{array}$$

We also check that $\mathsf{fix}\,g$ *computes as expected:*

$$\begin{array}{ll}
\mathsf{fix}\,g \cdot \mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}) & =(\beta \textit{ rule for } \mathsf{fix}) \\
\big(\mathsf{lam}\,\lambda n.\,\mathsf{ifZero}\,n\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,n)))\big) \cdot \mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}) & =(\Rightarrow\beta) \\
\mathsf{ifZero}\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}))\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero})))))\quad & =(\mathsf{ifZero}\textit{'s } \beta \textit{ rule}) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero}))))\quad & =(\mathsf{pred}\textit{'s } \beta \textit{ rule}) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{suc}\,\mathsf{zero}))\quad & =(\beta \textit{ rule for } \mathsf{fix}) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{lam}\,\lambda n.\,\mathsf{ifZero}\,n\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,n)))\cdot \mathsf{suc}\,\mathsf{zero}))\quad & =(\Rightarrow\beta) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{ifZero}\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,(\mathsf{suc}\,\mathsf{zero})))))) & =(\Rightarrow\beta) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,(\mathsf{suc}\,\mathsf{zero})))))\quad & =(\mathsf{pred}\textit{'s } \beta \textit{ rule}) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{zero}))))\quad & =(\beta \textit{ rule for } \mathsf{fix}) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{lam}\,\lambda n.\,\mathsf{ifZero}\,n\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,n)))\cdot \mathsf{zero}))))\quad & =(\Rightarrow\beta) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{ifZero}\,\mathsf{zero}\,(\mathsf{suc}\,\mathsf{zero})\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,g \cdot \mathsf{pred}\,\mathsf{zero})))))))\quad & =(\mathsf{ifZero}\textit{'s } \beta \textit{ rule}) \\
\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero})))) &
\end{array}$$

Exercise 172. *Define the iterator of natural numbers and show its computation rules.*

Exercise 173. *Show weak equational consistency for any second-order model: that is, if* $\mathsf{zero} = \mathsf{suc}\,\mathsf{zero}$*, then for all A and $u, v : \mathsf{Tm}\,A$, $u = v$.*

Exercise 174. *Show that using a second-order model of PCF, a second-order model of System T without η can be given. See also Remark 155.*

Exercise 175. *Which functions do the following PCF terms define?*

$$\mathsf{fix}\,(\lambda t.\,\mathsf{lam}\,\lambda x.\,x)$$
$$\mathsf{fix}\,(\lambda t.\,\mathsf{lam}\,\lambda x.\,\mathsf{ifZero}\,x\,x\,x)$$
$$\mathsf{fix}\,\big(\lambda t.\,\mathsf{lam}\,\lambda x.\,\mathsf{ifZero}\,x\,x\,(\mathsf{suc}\,\mathsf{zero})\big)$$
$$\mathsf{fix}\,\big(\lambda t.\,\mathsf{lam}\,\lambda x.\,\mathsf{ifZero}\,x\,(\mathsf{suc}\,\mathsf{zero})\,\mathsf{zero}\big)$$
$$\mathsf{fix}\,\Big(\lambda t.\,\mathsf{lam}\,\lambda x.\,\mathsf{ifZero}\,x\,x\,\big(\mathsf{suc}\,(t \cdot \mathsf{pred}\,x)\big)\Big)$$
$$\mathsf{fix}\,\Big(\lambda t.\,\mathsf{lam}\,\lambda x.\,\mathsf{ifZero}\,x\,\mathsf{zero}\,\big(\mathsf{suc}\,(t \cdot \mathsf{pred}\,x)\big)\Big)$$
$$\mathsf{fix}\,\Big(\lambda t.\,\mathsf{lam}\,\lambda x.\,\mathsf{ifZero}\,x\,(\mathsf{suc}\,\mathsf{zero})\,\big(\mathsf{suc}\,(t \cdot \mathsf{pred}\,x)\big)\Big)$$

The price we had to pay for having all recursive functions is that we have non-productive terms: $\mathsf{fix}\,(\lambda x.\,x) : \mathsf{Tm}\,A$ for any A, and we have $\mathsf{fix}\,(\lambda x.\,x) = (\lambda x.\,x)(\mathsf{fix}\,(\lambda x.\,x)) = \mathsf{fix}\,(\lambda x.\,x)$, so we don't obtain any new information from computing. Similarly, $\mathsf{fix}\,\mathsf{suc} = \mathsf{suc}\,(\mathsf{fix}\,\mathsf{suc}) = \mathsf{suc}\,(\mathsf{suc}\,(\mathsf{fix}\,\mathsf{suc})) = \ldots$, which will be a very large term never terminating. Compare this with $\mathsf{fix}\,(\lambda\_.\,t) = (\lambda\_.\,t)\,(\mathsf{fix}\,(\lambda\_.\,t)) = t$.

3.8.2 Recursive types

We have a fixpoint operator for $F : \mathsf{Ty} \to \mathsf{Ty}$ functions, that is, an operator $\mathsf{Fix} : (\mathsf{Ty} \to \mathsf{Ty}) \to \mathsf{Ty}$. There are two ways to express that $\mathsf{Fix}\, F$ is the fixpoint of F:

- *equirecursive types*, that is, we add the equation $\mathsf{Fix}\, F = F\, (\mathsf{Fix}\, F)$. This makes equality of types undecidable (and thus typechecking undecidable).
- *isorecursive types*: instead of adding an equation in types, we add an isomorphism $\mathsf{Fix}\, F \cong F\, (\mathsf{Fix}\, F)$.

We follow the latter route.

Definition 176 (A language with recursive types)**.** *A second-order model extends Fin (Definition 91) with the following components:*

$$\begin{aligned}
&\mathsf{fix} &&: (\mathsf{Tm}\, A \to \mathsf{Tm}\, A) \to \mathsf{Tm}\, A\\
&\mathsf{fix}\beta &&: \mathsf{fix}\, f = f\, (\mathsf{fix}\, f)\\
&\mathsf{Fix} &&: (\mathsf{Ty} \to \mathsf{Ty}) \to \mathsf{Ty}\\
&\mathsf{con} &&: \mathsf{Tm}\, (F\, (\mathsf{Fix}\, F)) \cong \mathsf{Tm}\, (\mathsf{Fix}\, F) : \mathsf{des}
\end{aligned}$$

Example 177 (Nat)**.** *We define natural numbers as a recursive type. We only write the implicit argument* $\{\lambda X.\, \top + X\}$ *of* con *once.*

$$\begin{aligned}
&\mathsf{Nat} &&:= \mathsf{Fix}\, \lambda X.\, \top + X\\
&\mathsf{zero} &&:= \mathsf{con}\, \{\lambda X.\, \top + X\}\, (\mathsf{inl}\, \mathsf{tt})\\
&\mathsf{suc} &&: \mathsf{Tm}\, \mathsf{Nat} \to \mathsf{Tm}\, \mathsf{Nat}\\
&\mathsf{suc}\, n &&:= \mathsf{con}\, (\mathsf{inr}\, n)\\
&\mathsf{ite} &&: \mathsf{Tm}\, A \to (\mathsf{Tm}\, A \to \mathsf{Tm}\, A) \to \mathsf{Tm}\, (\mathsf{Nat} \Rightarrow A)\\
&\mathsf{ite}\, z\, s &&:= \mathsf{fix}\, \lambda f.\, \mathsf{lam}\, \lambda n.\, \mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (f \cdot m))\, (\mathsf{des}\, n)\\
&\mathsf{Nat}\beta_1 : &&\mathsf{ite}\, z\, s \cdot \mathsf{zero} && =(\textit{definition})\\
&&&\big(\mathsf{fix}\, \lambda f.\, \mathsf{lam}\, \lambda n.\, \mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (f \cdot m))\, (\mathsf{des}\, n)\big) \cdot \mathsf{zero} && =(\mathsf{fix}\beta)\\
&&&\big(\mathsf{lam}\, \lambda n.\, \mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (\mathsf{ite}\, z\, s \cdot m))\, (\mathsf{des}\, n)\big) \cdot \mathsf{zero} && =(\Rightarrow\beta)\\
&&&\mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (\mathsf{ite}\, z\, s \cdot m))\, (\mathsf{des}\, \mathsf{zero}) && =(\textit{definition})\\
&&&\mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (\mathsf{ite}\, z\, s \cdot m))\, (\mathsf{des}\, (\mathsf{con}\, (\mathsf{inl}\, \mathsf{tt}))) && =(\mathsf{Fix}\beta)\\
&&&\mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (\mathsf{ite}\, z\, s \cdot m))\, (\mathsf{inl}\, \mathsf{tt}) && =(+\beta_1)\\
&&&z\\
&\mathsf{Nat}\beta_2 : &&\mathsf{ite}\, z\, s \cdot \mathsf{suc}\, n && =(\textit{definition})\\
&&&\big(\mathsf{fix}\, \lambda f.\, \mathsf{lam}\, \lambda n.\, \mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (f \cdot m))\, (\mathsf{des}\, n)\big) \cdot \mathsf{suc}\, n && =(\mathsf{fix}\beta)\\
&&&\big(\mathsf{lam}\, \lambda n.\, \mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (\mathsf{ite}\, z\, s \cdot m))\, (\mathsf{des}\, n)\big) \cdot \mathsf{suc}\, n && =(\Rightarrow\beta)\\
&&&\mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (\mathsf{ite}\, z\, s \cdot m))\, (\mathsf{des}\, (\mathsf{suc}\, n)) && =(\textit{definition})\\
&&&\mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (\mathsf{ite}\, z\, s \cdot m))\, (\mathsf{des}\, (\mathsf{con}\, (\mathsf{inr}\, n))) && =(\mathsf{Fix}\beta)\\
&&&\mathsf{case}\, (\lambda _.\, z)\, (\lambda m.\, s\, (\mathsf{ite}\, z\, s \cdot m))\, (\mathsf{inr}\, n) && =(+\beta_2)\\
&&&s\, (\mathsf{ite}\, z\, s \cdot n)
\end{aligned}$$

Food for thought 178. *Does* $\mathsf{Nat}\eta$ *hold?*

Example 179 (Stream)**.** *We define streams as a recursive type.*

$$\begin{aligned}
&\mathsf{Stream} &&: \mathsf{Ty} \to \mathsf{Ty}\\
&\mathsf{Stream}\, A &&:= \mathsf{Fix}\, \lambda A.\, A \times X\\
&\mathsf{head} &&: \mathsf{Tm}\, (\mathsf{Stream}\, A) \to \mathsf{Tm}\, A\\
&\mathsf{head}\, t &&:= \mathsf{fst}\, (\mathsf{des}\, t)\\
&\mathsf{tail} &&: \mathsf{Tm}\, (\mathsf{Stream}\, A) \to \mathsf{Tm}\, (\mathsf{Stream}\, A)\\
&\mathsf{tail}\, t &&:= \mathsf{snd}\, (\mathsf{des}\, t)\\
&\mathsf{gen} &&: (\mathsf{Tm}\, C \to \mathsf{Tm}\, A) \to (\mathsf{Tm}\, C \to \mathsf{Tm}\, C) \to \mathsf{Tm}\, (C \Rightarrow \mathsf{Stream}\, A)\\
&\mathsf{gen}\, h\, t &&:= \mathsf{fix}\, \lambda f.\, \mathsf{lam}\, \lambda c.\, \mathsf{con}\, (h\, c, f \cdot t\, c)
\end{aligned}$$

Exercise 180 (long)**.** *Prove the* β *rules for streams.*

Exercise 181 (long). *Show that for a stictly positive* $F : \mathsf{Ty} \to \mathsf{Ty}$*,* $\mathsf{Fix}\, F$ *satisfies all the rules of* $\mathsf{Ind}\, F$ *and* $\mathsf{Coind}\, F$*.*

We have meaningless types such as $D := \mathsf{Fix}\,(\lambda X.X)$ which gives as a $\mathsf{con} : \mathsf{Tm}\, D \to \mathsf{Tm}\, D$ and $\mathsf{des} : \mathsf{Tm}\, D \to \mathsf{Tm}\, D$, but we cannot create any element of D and cannot obtain any information from assumed elements of D.

3.8.3 Untyped lambda calculus

The simplest programming language with binders is the untyped lambda calculus.

Definition 182 (Untyped lambda calculus).

$$\begin{aligned}
&\mathsf{Tm} &&: \mathsf{Set}\\
&- \cdot - &&: \mathsf{Tm} \to \mathsf{Tm} \to \mathsf{Tm}\\
&\mathsf{lam} &&: (\mathsf{Tm} \to \mathsf{Tm}) \to \mathsf{Tm}\\
&\beta &&: \mathsf{lam}\, f \cdot u = f\, u
\end{aligned}$$

A fixpoint combinator:

$$\mathsf{Y} : \mathsf{Tm} := \mathsf{lam}\, \lambda f.\big(\mathsf{lam}\, \lambda x.f \cdot (x \cdot x)\big) \cdot \big(\mathsf{lam}\, \lambda x.f \cdot (x \cdot x)\big).$$

Exercise 183. *Show that* $\mathsf{Y} \cdot f = f \cdot (\mathsf{Y} \cdot f)$*.*

Exercise 184. *Show that there are terms* $\mathsf{zero}, \mathsf{suc}, \mathsf{ite} : \mathsf{Tm}$ *such that* $\mathsf{ite} \cdot z \cdot s \cdot \mathsf{zero} = z$ *and* $\mathsf{ite} \cdot z \cdot s \cdot (\mathsf{suc}\, t) = s \cdot (\mathsf{ite} \cdot z \cdot s \cdot t)$*.*

Exercise 185. *Show that we have similar encoding for all inductive and coinductive types. What about* η *rules?*

We have another untyped language which has one more operator and one more equation, but is first-order:

Definition 186 (Untyped combinator calculus).

$$\begin{aligned}
&\mathsf{Tm} &&: \mathsf{Set}\\
&- \cdot - &&: \mathsf{Tm} \to \mathsf{Tm} \to \mathsf{Tm}\\
&\mathsf{K} &&: \mathsf{Tm}\\
&\mathsf{S} &&: \mathsf{Tm}\\
&\mathsf{K}\beta &&: \mathsf{K} \cdot a \cdot b = a\\
&\mathsf{S}\beta &&: \mathsf{S} \cdot f \cdot g \cdot a = f \cdot a \cdot (g \cdot a)
\end{aligned}$$

See [AKSV23] for the relationship of untyped lambda and combinator calculi.

It is remarkable that Schönfinkel's untyped combinator calculus is still the simplest Turing complete language, even though it was the first such. Note that in contrast with STCC, untyped combinator calculus does not have normalisation.

TODO: add exercises from Hungarian course notes, including fixpoint operator for SK.

3.9 Martin-Löf type theory

Definition 187 (Minimal Martin-Löf type theory (mini-MLTT)).

$$\begin{aligned}
&\mathsf{Ty} &&: \mathsf{Set}\\
&\mathsf{Tm} &&: \mathsf{Ty} \to \mathsf{Set}\\
&\iota &&: \mathsf{Ty}\\
&\Pi &&: (A : \mathsf{Ty}) \to (\mathsf{Tm}\, A \to \mathsf{Ty}) \to \mathsf{Ty}\\
&\mathsf{lam} &&: ((a : \mathsf{Tm}\, A) \to \mathsf{Tm}\,(B\, a)) \cong \mathsf{Tm}\,(\Pi\, A\, B) : - \cdot -
\end{aligned}$$

Definition 188 (Martin-Löf type theory with Π and universes).

$$\begin{aligned}
&\mathsf{Ty} &&: \mathbb{N} \to \mathsf{Set} && \qquad &&\mathsf{U} &&: (i : \mathbb{N}) \to \mathsf{Ty}\,(1 + i)\\
&\mathsf{Tm} &&: \mathsf{Ty}\, i \to \mathsf{Set} && \qquad &&\mathsf{c} &&: \mathsf{Ty}\, i \cong \mathsf{Tm}\,(\mathsf{U}\, i) : \mathsf{El}\\
&\Pi &&: (A : \mathsf{Ty}\, i) \to (\mathsf{Tm}\, A \to \mathsf{Ty}\, i) \to \mathsf{Ty}\, i && \qquad &&\mathsf{Lift} &&: \mathsf{Ty}\, i \to \mathsf{Ty}\,(1 + i)\\
&\mathsf{lam} &&: ((a : \mathsf{Tm}\, A) \to \mathsf{Tm}\,(B\, a)) \cong \mathsf{Tm}\,(\Pi\, A\, B) : - \cdot - && \qquad &&\mathsf{mk} &&: \mathsf{Tm}\, A \cong \mathsf{Tm}\,(\mathsf{Lift}\, A) : \mathsf{un}
\end{aligned}$$

Definition 189 (Martin-Löf type theory with inductive types)**.** *We extend Definition 188 with the following.*

$$
\begin{array}{ll}
\Sigma & : (A : \mathsf{Ty}\, i) \to (\mathsf{Tm}\, A \to \mathsf{Ty}\, i) \to \mathsf{Ty}\, i \\
(-,-) & : (a : \mathsf{Tm}\, A) \times \mathsf{Tm}\,(B\, a) \cong \mathsf{Tm}\,(\Sigma\, A\, B) : \mathsf{fst}, \mathsf{snd} \\
\bot & : \mathsf{Ty}\, 0 \\
\mathsf{exfalso} & : \mathsf{Tm}\, \bot \to \mathsf{Tm}\, A \\
\top & : \mathsf{Ty}\, 0 \\
\mathsf{tt} & : \top \cong \mathsf{Tm}\, \top \\
\mathsf{Bool} & : \mathsf{Ty}\, 0 \\
\mathsf{true} & : \mathsf{Tm}\, \mathsf{Bool} \\
\mathsf{false} & : \mathsf{Tm}\, \mathsf{Bool} \\
\mathsf{indBool} & : (C : \mathsf{Tm}\, \mathsf{Bool} \to \mathsf{Ty}\, i) \to \mathsf{Tm}\,(C\, \mathsf{true}) \to \mathsf{Tm}\,(C\, \mathsf{false}) \to (b : \mathsf{Tm}\, \mathsf{Bool}) \to \mathsf{Tm}\,(C\, b) \\
\mathsf{Bool}\beta_1 & : \mathsf{indBool}\, t\, f\, \mathsf{true} = t \\
\mathsf{Bool}\beta_2 & : \mathsf{indBool}\, t\, f\, \mathsf{false} = f \\
\mathsf{Id} & : (A : \mathsf{Ty}\, i) \to \mathsf{Tm}\, A \to \mathsf{Tm}\, A \to \mathsf{Ty}\, i \\
\mathsf{refl} & : (a : \mathsf{Tm}\, A) \to \mathsf{Tm}\,(\mathsf{Id}\, A\, a\, a) \\
\mathsf{J} & : \big(C : (x : \mathsf{Tm}\, A) \to \mathsf{Tm}\,(\mathsf{Id}\, A\, a\, x) \to \mathsf{Ty}\, i\big) \to \\
 & \quad \mathsf{Tm}\,\big(C\, a\,(\mathsf{refl}\, a)\big) \to (x : \mathsf{Tm}\, A)(e : \mathsf{Tm}\,(\mathsf{Id}\, A\, a\, x)) \to \mathsf{Tm}\,(C\, x\, e) \\
\mathsf{Id}\beta & : \mathsf{J}\, C\, w\, a\,(\mathsf{refl}\, a) = w \\
\mathsf{W} & : (S : \mathsf{Ty}\, i) \to (\mathsf{Tm}\, S \to \mathsf{Ty}\, i) \to \mathsf{Ty}\, i \\
\mathsf{sup} & : (s : \mathsf{Tm}\, S) \to (\mathsf{Tm}\,(P\, s) \to \mathsf{Tm}\,(\mathsf{W}\, S\, P)) \to \mathsf{Tm}\,(\mathsf{W}\, S\, P) \\
\mathsf{indW} & : (C : \mathsf{Tm}\,(\mathsf{W}\, S\, P) \to \mathsf{Ty}\, i) \to \Big(\big((p : \mathsf{Tm}\,(P\, s)) \to \mathsf{Tm}\,(C\,(f\, p))\big) \to \mathsf{Tm}\,\big(C\,(\mathsf{sup}\, s\, f)\big)\Big) \to \\
 & \quad (w : \mathsf{Tm}\,(\mathsf{W}\, \mathsf{S}\, \mathsf{P})) \to \mathsf{Tm}\,\big(C\, w\big) \\
\mathsf{W}\beta & : \mathsf{indW}\, C\, h\,(\mathsf{sup}\, s\, f) = h\,(\lambda p.\mathsf{indW}\, C\, h\,(f\, p))
\end{array}
$$

TODO: extensional t.t. (equality reflection). Quotient types.

3.10 Primitive recursive arithmetic

We separate first-order types from general types so that we can restrict the iterator to not return functions. First-order types are Nat and products, general types include functions.

Definition 190 (Primitive recursive arithmetic (PRA))**.** *A second-order model comprises the following com-*

ponents:

$$
\begin{array}{ll}
\mathsf{Ty} & : \mathsf{Set} \\
\mathsf{Tm} & : \mathsf{Ty} \to \mathsf{Set} \\
\mathsf{FTy} & : \mathsf{Set} \\
\ulcorner - \urcorner & : \mathsf{FTy} \to \mathsf{Ty} \\
\mathsf{Nat} & : \mathsf{FTy} \\
\mathsf{zero} & : \mathsf{Tm}\,\ulcorner \mathsf{Nat} \urcorner \\
\mathsf{suc} & : \mathsf{Tm}\,\ulcorner \mathsf{Nat} \urcorner \to \mathsf{Tm}\,\ulcorner \mathsf{Nat} \urcorner \\
\mathsf{ite} & : \mathsf{Tm}\,\ulcorner A \urcorner \to (\mathsf{Tm}\,\ulcorner A \urcorner \to \mathsf{Tm}\,\ulcorner A \urcorner) \to \mathsf{Tm}\,\ulcorner \mathsf{Nat} \urcorner \to \mathsf{Tm}\,\ulcorner A \urcorner \\
\mathsf{Nat}\beta_1 & : \mathsf{ite}\,z\,s\,\mathsf{zero} = z \\
\mathsf{Nat}\beta_2 & : \mathsf{ite}\,z\,s\,(\mathsf{suc}\,t) = s\,(\mathsf{ite}\,z\,s\,t) \\
- \times - & : \mathsf{FTy} \to \mathsf{FTy} \to \mathsf{FTy} \\
-, - & : \mathsf{Tm}\,\ulcorner A \urcorner \to \mathsf{Tm}\,\ulcorner B \urcorner \to \mathsf{Tm}\,\ulcorner A \times B \urcorner \\
\mathsf{fst} & : \mathsf{Tm}\,\ulcorner A \times B \urcorner \to \mathsf{Tm}\ulcorner A \urcorner \\
\mathsf{snd} & : \mathsf{Tm}\,\ulcorner A \times B \urcorner \to \mathsf{Tm}\ulcorner B \urcorner \\
\times\beta_1 & : \mathsf{fst}\,(a, b) = a \\
\times\beta_2 & : \mathsf{snd}\,(a, b) = b \\
- \Rightarrow - & : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} \\
\mathsf{lam} & : (\mathsf{Tm}\,A \to \mathsf{Tm}\,B) \to \mathsf{Tm}\,(A \Rightarrow B) \\
- \cdot - & : \mathsf{Tm}\,(A \Rightarrow B) \to \mathsf{Tm}\,A \to \mathsf{Tm}\,B \\
\Rightarrow\beta & : \mathsf{lam}\,f \cdot a = f\,a
\end{array}
$$

The Ackermann function [Ack28] is not definable in this language.

Food for thought 191. *Show that the* $\ulcorner \mathsf{Nat} \urcorner \Rightarrow \ulcorner \mathsf{Nat} \urcorner$ *functions definable in our theory are the same as in the traditional presentation of primitive recursive arithmetic. See [BSvB24] for a PRA similar to ours which also features dependent types.*

3.11 First-order logic

Definition 192 (Minimal intuitionistic first-order logic with natural deduction style proof theory)**.**

$$\frac{}{\mathsf{For} : \mathsf{Set}} \qquad \frac{}{\mathsf{Tm} : \mathsf{Set}} \qquad \frac{A : \mathsf{For} \qquad B : \mathsf{For}}{A \supset B : \mathsf{For}} \qquad \frac{x : \mathsf{Tm} \vdash A : \mathsf{For}}{\forall x.A : \mathsf{For}} \qquad \frac{t : \mathsf{Tm} \qquad t' : \mathsf{Tm}}{\mathsf{Eq}\,t\,t' : \mathsf{For}} \qquad \frac{A : \mathsf{For}}{\mathsf{Pf}\,A : \mathsf{Set}}$$

$$\frac{p : \mathsf{Pf}\,A \qquad q : \mathsf{Pf}\,A}{p = q} \qquad \frac{\mathsf{Pf}\,A \vdash \mathsf{Pf}\,B}{\mathsf{Pf}\,(A \supset B)} \qquad \frac{\mathsf{Pf}\,(A \supset B) \qquad \mathsf{Pf}\,A}{\mathsf{Pf}\,B} \qquad \frac{x : \mathsf{Tm} \vdash \mathsf{Pf}\,A}{\mathsf{Pf}\,(\forall x.A)} \qquad \frac{\mathsf{Pf}\,(\forall x.A) \qquad t : \mathsf{Tm}}{\mathsf{Pf}\,(A[x \mapsto t])}$$

$$\frac{t : \mathsf{Tm}}{\mathsf{Pf}\,(\mathsf{Eq}\,t\,t)} \qquad \frac{x : \mathsf{Tm} \vdash A : \mathsf{For} \qquad \mathsf{Pf}\,(\mathsf{Eq}\,t\,t') \qquad \mathsf{Pf}\,(A[x \mapsto t])}{\mathsf{Pf}\,(A[x \mapsto t'])}$$

Exercise 193. *Define full intuitionistic first-order logic as a SOGAT. Add something to make it classical.*

3.12 Theories of signatures for (SO)(G)ATs

Theories of signatures (ToSs) are languages for describing signatures for algebraic theories. Above we defined (SO)GATs by giving the notion of (second-order) model, signatures are a more precise language which enforces that all operators result in a sort, strict positivity etc.

Definition 194 (AT)**.** *A second-order model of the ToS for ATs:*

$$
\begin{array}{ll}
\mathsf{Ty} & : \mathsf{Set} \\
\mathsf{Tm} & : \mathsf{Ty} \to \mathsf{Set} \\
\Sigma & : (A : \mathsf{Ty}) \to (\mathsf{Tm}\,A \to \mathsf{Ty}) \to \mathsf{Ty} \\
\mathsf{Srt} & : \mathsf{Ty} \\
\Pi\mathsf{Srt} & : (\mathsf{Tm}\,\mathsf{Srt} \to \mathsf{Ty}) \to \mathsf{Ty} \\
- \cdot - & : \mathsf{Tm}\,(\Pi\mathsf{Srt}\,B) \to (x : \mathsf{Tm}\,\mathsf{Srt}) \to \mathsf{Tm}\,(B\,x) \\
\mathsf{Id} & : \mathsf{Tm}\,\mathsf{Srt} \to \mathsf{Tm}\,\mathsf{Srt} \to \mathsf{Ty}
\end{array}
$$

A signature is an element of Ty*.*

For a $B : \mathsf{Ty}$, we introduce the abbreviation $\mathsf{Srt} \Rightarrow B := \Pi \mathsf{Srt}\,(\lambda\_.\, B)$.

Example 195. *The AT of monoids (see Definition 1) is given by the following signature:*

$$\Sigma\,(\mathsf{Srt} \Rightarrow \mathsf{Srt} \Rightarrow \mathsf{Srt})\,\lambda op.\,\Pi \mathsf{Srt}\,\lambda x.\,\Pi \mathsf{Srt}\,\lambda y.\,\Pi \mathsf{Srt}\,\lambda z.\,\mathsf{Id}\,\big(op \cdot (op \cdot x \cdot y) \cdot z\big)\,\big(op \cdot x \cdot (op \cdot y \cdot z)\big) \times$$
$$\Sigma\,\mathsf{Srt}\,\lambda u.\,\big(\Pi \mathsf{Srt}\,\lambda x.\,\mathsf{Id}\,(op \cdot u \cdot x)\,x\big) \times \big(\Pi \mathsf{Srt}\,\lambda x.\,\mathsf{Id}\,(op \cdot x \cdot u)\,x\big)$$

Exercise 196. *Define the signature for the following ATs: pointed set with an endofunction (see Definition 10), untyped combinator calculus.*

Definition 197 (GAT)**.** *A second-order model of the ToS for GATs:*

$$\begin{aligned}
&\mathsf{Ty} && : \mathsf{Set}\\
&\mathsf{Tm} && : \mathsf{Ty} \to \mathsf{Set}\\
&\Sigma && : (A : \mathsf{Ty}) \to (\mathsf{Tm}\,A \to \mathsf{Ty}) \to \mathsf{Ty}\\
&(_,_) && : (a : \mathsf{Tm}\,A) \times \mathsf{Tm}\,(B\,a) \cong \mathsf{Tm}\,(\Sigma\,A\,B) : \mathsf{fst}, \mathsf{snd}\\
&\mathsf{U} && : \mathsf{Ty}\\
&\mathsf{El} && : \mathsf{Tm}\,\mathsf{U} \to \mathsf{Ty}\\
&\Pi && : (a : \mathsf{Tm}\,\mathsf{U}) \to (\mathsf{Tm}\,(\mathsf{El}\,a) \to \mathsf{Ty}) \to \mathsf{Ty}\\
&_ \cdot _ && : \mathsf{Tm}\,(\Pi\,a\,B) \to (x : \mathsf{Tm}\,(\mathsf{El}\,a)) \to \mathsf{Tm}\,(B\,x)\\
&\mathsf{Id} && : (a : \mathsf{Tm}\,\mathsf{U}) \to \mathsf{Tm}\,(\mathsf{El}\,a) \to \mathsf{Tm}\,(\mathsf{El}\,a) \to \mathsf{Ty}\\
&\mathsf{reflect} && : \mathsf{Tm}\,(\mathsf{Id}\,a\,u\,v) \to u = v
\end{aligned}$$

A signature is an element of Ty.

We will use the abbreviations $A \Rightarrow B := \Pi\,A\,(\lambda\_.B)$ and $A \times B := \Sigma\,A\,(\lambda\_.B)$.

Example 198. *The GAT of preorders (see Definition 52) is given by the following signature:*

$$\begin{aligned}
&\Sigma\,\mathsf{U}\,\lambda Ob.\Sigma\\
&\quad (Ob \Rightarrow Ob \Rightarrow \mathsf{U})\,\lambda Mor.\\
&\quad \big(\Pi\,Ob\,\lambda I.\Pi\,Ob\,\lambda J.\Pi\,Ob\,\lambda K.Mor \cdot J \cdot I \Rightarrow Mor \cdot K \cdot J \Rightarrow \mathsf{El}\,(Mor \cdot K \cdot I)\big)\times\\
&\quad \big(\Pi\,Ob\,\lambda I.\mathsf{El}\,(Mor \cdot I \cdot I)\big)\times\\
&\quad \big(\Pi\,Ob\,\lambda I.\Pi\,Ob\,\lambda J.\Pi\,(Mor \cdot J \cdot I)\,\lambda f.\Pi\,(Mor \cdot J \cdot I)\,\lambda g.\mathsf{Id}\,(Mor \cdot J \cdot I)\,f\,g\big)
\end{aligned}$$

Exercise 199. *Define the signature for the following GATs: category, Razor, STCC.*

Definition 200 (SOGAT)**.** *A second-order model of the ToS for SOGATs is a second-order model of the ToS for GATs (Definition 197) extended with the following components:*

$$\begin{aligned}
&\mathsf{U}^+ && : \mathsf{Ty}\\
&\mathsf{el}^+ && : \mathsf{Tm}\,\mathsf{U}^+ \to \mathsf{Tm}\,\mathsf{U}\\
&\pi^+ && : (a^+ : \mathsf{Tm}\,\mathsf{U}^+) \to \Big(\mathsf{Tm}\,\big(\mathsf{El}\,(\mathsf{el}^+\,a^+)\big) \to \mathsf{Tm}\,\mathsf{U}\Big) \to \mathsf{Tm}\,\mathsf{U}\\
&\mathsf{lam}^+ && : \Big(\big(x : \mathsf{El}\,(\mathsf{el}^+\,a^+)\big) \to \mathsf{Tm}\,(\mathsf{El}\,(b\,x))\Big) \cong \mathsf{Tm}\,\big(\mathsf{El}\,(\pi^+\,a^+\,b)\big) : _ \cdot^+ _
\end{aligned}$$

U^+ is the universe of those sorts which can appear at the left hand side of an arrow in an argument of an operator. In STLC, Ty is in U, while Tm is in U^+.

Example 201. *The signature for STLC (see Definition 54):*

$$
\begin{aligned}
&\Sigma\, \mathsf{U} \lambda Ty.\, \Sigma \\
&(Ty \Rightarrow \mathsf{U}^+)\, \lambda Tm. \\
&\mathsf{El}\, Ty \times \Sigma \\
&(Ty \Rightarrow Ty \Rightarrow \mathsf{El}\, Ty)\, \lambda arr.\, \Sigma \\
&\Big(\Pi\, Ty\, \lambda A.\, \Pi\, Ty\, \lambda B.\, \big(Tm \cdot A \Rightarrow^+ \mathsf{el}^+\, (Tm \cdot B)\big) \Rightarrow \mathsf{El}\, \big(\mathsf{el}^+\, (Tm \cdot (arr \cdot A \cdot B))\big)\Big)\, \lambda \mathsf{lam}.\, \Sigma \\
&\Big(\Pi\, Ty\, \lambda A.\, \Pi\, Ty\, \lambda B.\, \mathsf{el}^+\, (Tm \cdot (arr \cdot A \cdot B)) \Rightarrow \mathsf{el}^+\, (Tm \cdot A) \Rightarrow \mathsf{El}\, \big(\mathsf{el}^+\, (Tm \cdot B)\big)\Big)\, \lambda app. \\
&\Big(\Pi\, Ty\, \lambda A.\, \Pi\, Ty\, \lambda B.\, \Pi\, \big(Tm \cdot A \Rightarrow^+ \mathsf{el}^+\, (Tm \cdot B)\big)\, \lambda b.\, \Pi\, \big(\mathsf{el}^+\, (Tm \cdot A)\big)\, \lambda a. \\
&\quad \mathsf{Id}\, \big(\mathsf{el}^+\, (Tm \cdot B)\big)\, \big(app \cdot A \cdot B \cdot (lam \cdot A \cdot B \cdot b) \cdot a\big)\, (b \cdot^+ a)\Big) \times \\
&\Big(\Pi\, Ty\, \lambda A.\, \Pi\, Ty\, \lambda B.\, \Pi\, \big(Tm \cdot (arr \cdot A \cdot B)\big)\, \lambda f. \\
&\quad \mathsf{Id}\, \Big(\mathsf{el}^+\, \big(Tm \cdot (arr \cdot A \cdot B)\big)\Big)\, f\, \big(lam \cdot A \cdot B \cdot (\mathsf{lam}^+\, \lambda x.\, app \cdot A \cdot B \cdot f \cdot x)\big)\Big)
\end{aligned}
$$

Exercise 202. *Define the signature for the following SOGATs: untyped lambda calculus, System T, PCF, first-order logic, System F, System F_ω, Martin-Löf type theory, ToS of ATs, ToS of GATs, ToS of SOGATs. Carefully consider which sorts should be in* U *and which should be in* U^+.

Remark: we only showed how to define *closed* SOGATs, but e.g. Definition 188 is open as it refers to the external set $\mathbb{N}$. The ToS of (SO)GATs can be extended to allow open signatures, see [KX24].

4 Converting SOGATs into GATs

As mentioned in the beginning of Section 3, second-order algebraic theories are not really algebraic, there is no way to define morphisms of second-order models:

Example 203. *A second-order model of the untyped lambda calculus consists of a* $\mathsf{Tm} : \mathsf{Set}$ *together with*

$$\mathsf{lam} : (\mathsf{Tm} \to \mathsf{Tm}) \cong \mathsf{Tm} : - \cdot -.$$

A morphism between second-order models M and N would be a function $\mathsf{Tm} : \mathsf{Tm}_M \to \mathsf{Tm}_N$ *which preserves* $- \cdot -$ *by* $\mathsf{Tm}\, (t_M \cdot_M a_M) = \mathsf{Tm}\, t_M \cdot_N \mathsf{Tm}\, a_M$*, but how do we express preservation of* lam*? This would be an equation such as* $\mathsf{Tm}\, (\mathsf{lam}_M\, f_M) = \mathsf{lam}_N\, (\mathsf{Tm} \circ f_M \circ ?)$*, but we don't know what to put in place of the* $?$ *which is in* $\mathsf{Tm}_N \to \mathsf{Tm}_M$*. If we wanted to only define isomorphisms, this would work.*

Instead, we translate SOGATs to GATs and work with the resulting GAT. For GATs, we have good notions of morphism, dependent model, syntax, and so on, as we have seen in Section 2. That is, by a category of models of a SOGAT, we mean the category of models of the GAT which is the result of the translation.

Food for thought 204. *Actually, there are multiple different translations which end up in different notions of models, e.g. instead of parallel substitution calculus, one can use single substitutions [KX24], models can be contextual (where contexts are inductively built, which is a special case of models where certain sorts are inductively generated), contexts can come with concatenation, and so on. For each SOGAT, there should also be a combinatory first-order GAT, just like there is the combinatory first-order version of the SOGAT of lambda calculus.*

4.1 STLC

The idea of the translation is that we introduce a new sort of contexts which are a list of the free variables. We index the sorts of our theory with contexts, the context index lists the possible free variables in a term. This way we can get rid of second-order function spaces. For example, the $\mathsf{lam} : (\mathsf{Tm}\, A \to \mathsf{Tm}\, B) \to \mathsf{Tm}\, (A \Rightarrow B)$ second-order operation becomes $\mathsf{lam} : \mathsf{Tm}\, (\Gamma \triangleright A)\, B \to \mathsf{Tm}\, \Gamma\, (A \Rightarrow B)$. That is, the $\mathsf{Tm}\, A$ dependency of the $\mathsf{Tm}\, B$ argument of lam becomes an extra variable of type A in the context (we read $\Gamma \triangleright A$ as the context Γ extended with an extra free variable of type A). We also introduce a sort of substitutions with their action on terms called instantiation. These allow expressing the β law in STLC: $\mathsf{lam}\, b \cdot a = b\, a$ becomes $\mathsf{lam}\, b \cdot a = b[\mathsf{id}, a]$ where $-[-]$ is the instantiation operator for Tm, (id, a) is a substitution from Γ to $\Gamma \triangleright A$ which leaves Γ untouched and substitutes a for the free variable of type A. The GAT that we obtain is called an explicit substitution calculus with parallel substitutions.

We first present and explain the result of the translation for STLC.

Definition 205 (First-order model of STLC (optimised version), see Definition 54 and Example 201)**.**

$$
\begin{array}{ll|ll}
\mathsf{Con} & : \mathsf{Set} & - \triangleright - & : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Con} \\
\mathsf{Sub} & : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set} & -, - & : \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Tm}\,\Delta\,A \to \mathsf{Sub}\,\Delta\,(\Gamma \triangleright A) \\
- \circ - & : \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Sub}\,\Theta\,\Delta \to \mathsf{Sub}\,\Theta\,\Gamma & \mathsf{p} & : \mathsf{Sub}\,(\Gamma \triangleright A)\,\Gamma \\
\mathsf{ass} & : (\gamma \circ \delta) \circ \theta = \gamma \circ (\delta \circ \theta) & \mathsf{q} & : \mathsf{Tm}\,(\Gamma \triangleright A)\,A \\
\mathsf{id} & : \mathsf{Sub}\,\Gamma\,\Gamma & \triangleright\beta_1 & : \mathsf{p} \circ (\gamma, a) = \gamma \\
\mathsf{idl} & : \mathsf{id} \circ \gamma = \gamma & \triangleright\beta_2 & : \mathsf{q}[\gamma, a] = a \\
\mathsf{idr} & : \gamma \circ \mathsf{id} = \gamma & \triangleright\eta & : \sigma = (\mathsf{p} \circ \sigma, \mathsf{q}[\sigma]) \\
\diamond & : \mathsf{Con} & \iota & : \mathsf{Ty} \\
\epsilon & : \mathsf{Sub}\,\Gamma\,\diamond & - \Rightarrow - & : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty} \\
\diamond\eta & : (\sigma : \mathsf{Sub}\,\Gamma\,\diamond) \to \sigma = \epsilon & \mathsf{lam} & : \mathsf{Tm}\,(\Gamma \triangleright A)\,B \to \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) \\
\mathsf{Ty} & : \mathsf{Set} & \mathsf{lam}[] & : (\mathsf{lam}\,b)[\gamma] = \mathsf{lam}\,(b[\gamma \circ \mathsf{p}, \mathsf{q}]) \\
\mathsf{Tm} & : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set} & - \cdot - & : \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,B \\
-[-] & : \mathsf{Tm}\,\Gamma\,A \to \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Tm}\,\Delta\,A & \cdot[] & : (f \cdot a)[\gamma] = (f[\gamma]) \cdot (a[\gamma]) \\
[\circ] & : a[\gamma \circ \delta] = a[\gamma][\delta] & \beta & : \mathsf{lam}\,b \cdot a = b[\mathsf{id}, a] \\
[\mathsf{id}] & : a[\mathsf{id}] = a & \eta & : f = \mathsf{lam}\,(f[\mathsf{p}] \cdot \mathsf{q})
\end{array}
$$

Contexts (lists of types) and substitutions (context morphisms, lists of terms) form a category ($\mathsf{Con}, \ldots, \mathsf{idr}$) with a terminal object ($\diamond$ is the empty context denoting no free variables, ϵ is the empty substitution into $\diamond$, $\diamond\eta$ expresses that it is unique). The sort Ty is unchanged compared to the second-order version. The sort of terms however has an extra context-index. $\mathsf{Tm}\,\Gamma\,A$ is a term of type A which might have free variables which are declared in Γ. The $-\triangleright-$ context extension operator allows us to put extra variables in a context. For example, the context $\diamond \triangleright \iota \triangleright \iota \Rightarrow \iota$ has two free variables of types ι and $\iota \Rightarrow \iota$, respectively. The sort $\mathsf{Sub}\,\Delta\,\Gamma$ is a substitution from Δ to Γ which means a list of Γ-many terms, all in context Δ. For example, if $\Gamma = \diamond \triangleright \iota \triangleright \iota \Rightarrow \iota$, then $\mathsf{Sub}\,\Delta\,\Gamma \cong \mathsf{Tm}\,\Delta\,\iota \times \mathsf{Tm}\,\Delta\,(\iota \Rightarrow \iota)$. This is ensured by the components $-, -, \ldots, \triangleright\eta$ which can be summarised as

$$(\mathsf{p} \circ -, \mathsf{q}[-]) : \mathsf{Sub}\,\Delta\,(\Gamma \triangleright A) \cong \mathsf{Sub}\,\Delta\,\Gamma \times \mathsf{Tm}\,\Delta\,A : (-, -).$$

Instantiation $-[-]$ gives meaning to variables via substitutions. For example, assume a term which depends only on a variable of type $\iota \Rightarrow \iota$, that is, $t : \mathsf{Tm}\,(\diamond \triangleright \iota \Rightarrow \iota)\,A$, and assume a closed term of type $\iota \Rightarrow \iota$, that is $u : \mathsf{Tm}\,\diamond\,(\iota \Rightarrow \iota)$. Now we can substitute u into t making it closed: $t[\epsilon, u] : \mathsf{Tm}\,\diamond\,A$. The rules $\mathsf{lam}[], \cdot[]$ explain how to commute instantiation and term formers, the rule $\triangleright\beta_2$ explains how to instantiate the last variable (De Bruijn index 0) in the context. Further De Bruijn indices are given by weakening: $1 = \mathsf{q}[\mathsf{p}]$, $2 = \mathsf{q}[\mathsf{p}][\mathsf{p}]$, $3 = \mathsf{q}[\mathsf{p}][\mathsf{p}][\mathsf{p}]$, and so on. The rule $\mathsf{lam}[]$ is interesting: here $\gamma : \mathsf{Sub}\,\Delta\,\Gamma$ and $b : \mathsf{Tm}\,(\Gamma \triangleright A)\,B$, so $b[\gamma]$ does not make sense. We have to *lift* γ so that we obtain a substitution which does not touch the last variable, $(\gamma \circ \mathsf{p}, \mathsf{q}) : \mathsf{Sub}\,(\Delta \triangleright A)\,(\Gamma \triangleright A)$, and apply this under the lam.

Example 206 (Naturality of $-, -$)**.** *We prove that in a first-order model of STLC, $(\gamma, a) \circ \delta = (\gamma \circ \delta, a[\delta])$ for any γ, a and δ.*

$$
\begin{array}{ll}
(\gamma, a) \circ \delta & = (\triangleright\eta) \\
(\mathsf{p} \circ ((\gamma, a) \circ \delta), \mathsf{q}[(\gamma, a) \circ \delta]) & = (\mathsf{ass}) \\
((\mathsf{p} \circ (\gamma, a)) \circ \delta, \mathsf{q}[(\gamma, a) \circ \delta]) & = (\triangleright\beta_1) \\
(\gamma \circ \delta, \mathsf{q}[(\gamma, a) \circ \delta]) & = ([\circ]) \\
(\gamma \circ \delta, \mathsf{q}[\gamma, a][\delta]) & = (\triangleright\beta_2) \\
(\gamma \circ \delta, a[\delta]) &
\end{array}
$$

Exercise 207. *Prove that assuming naturality of $-, -$ (i.e. $(\gamma, a) \circ \delta = (\gamma \circ \delta, a[\delta])$), the functor laws for Tm ($[\circ]$, $[\mathsf{id}]$) can be derived.*

Exercise 208. *Define substitutions which swap variables, duplicate variables, forget variables somewhere in the middle of the context, i.e. elements of the following sets:*

$$
\begin{array}{l}
\mathsf{Sub}\,(\Gamma \triangleright A \triangleright B_1 \triangleright \ldots \triangleright B_n \triangleright C \triangleright D_1 \triangleright \ldots \triangleright D_m)\,(\Gamma \triangleright C \triangleright B_1 \triangleright \ldots \triangleright B_n \triangleright A \triangleright D_1 \triangleright \ldots \triangleright D_m) \\
\mathsf{Sub}\,(\Gamma \triangleright A \triangleright B_1 \triangleright \ldots \triangleright B_n \triangleright C_1 \triangleright \ldots \triangleright C_m)\,(\Gamma \triangleright A \triangleright B_1 \triangleright \ldots \triangleright B_n \triangleright A \triangleright C_1 \triangleright \ldots \triangleright C_m) \\
\mathsf{Sub}\,(\Gamma \triangleright A \triangleright B_1 \triangleright \ldots \triangleright B_n)\,(\Gamma \triangleright B_1 \triangleright \ldots \triangleright B_n)
\end{array}
$$

Exercise 209. *Lifting of $\gamma : \mathsf{Sub}\,\Delta\,\Gamma$ is $\gamma^\uparrow := (\gamma \circ \mathsf{p}, \mathsf{q})$. Show $(\gamma \circ \delta)^\uparrow = (\gamma^\uparrow) \circ (\delta^\uparrow)$, $\mathsf{id}^\uparrow = \mathsf{id}$, $\mathsf{p}^\uparrow \circ (\mathsf{id}, \mathsf{q}) = \mathsf{id}$.*

Example 210 (C.f. Example 56)**.** *In any first-order model of STLC, we have* $\mathsf{lam}\,\mathsf{q} : \mathsf{Tm}\ \diamond\ (\iota \Rightarrow \iota)$*. If we write the implicit arguments, this is* $\mathsf{lam}\,\{\diamond\}\,\{\iota\}\,\{\iota\}\,(\mathsf{q}\,\{\diamond\}\,\{\iota\})$*. We can move this program into any context* Γ *by instantiating it by* $\epsilon\,\{\Gamma\} : \mathsf{Sub}\,\Gamma\,\diamond$*, and obtain* $(\mathsf{lam}\,\mathsf{q})[\epsilon] \overset{\mathsf{lam}[]}{=} \mathsf{lam}\,(\mathsf{q}[\epsilon \circ \mathsf{p}, \mathsf{q}]) \overset{\triangleright\beta_2}{=} \mathsf{lam}\,\mathsf{q}$*. The original and the instantiated terms have different implicit arguments: the right hand side* $\mathsf{lam}\,\mathsf{q}$ *is actually* $\mathsf{lam}\,\{\Gamma\}\,\{\iota\}\,\{\iota\}\,(\mathsf{q}\,\{\Gamma\}\,\{\iota\})$*.*

Applying the identity function to some argument gives $\mathsf{lam}\,\mathsf{q} \cdot u \overset{\beta}{=} \mathsf{q}[\mathsf{id}, u] \overset{\triangleright\beta_2}{=} u$*.*

Example 211 (C.f. Example 61)**.** *In any first-order model of STLC, the double function is defined as*

$$\mathsf{lam}\left(\mathsf{lam}\left((\mathsf{q}[\mathsf{p}]) \cdot ((\mathsf{q}[\mathsf{p}]) \cdot \mathsf{q})\right)\right) : \mathsf{Tm}\ \diamond\ \big((A \Rightarrow A) \Rightarrow A \Rightarrow A\big)$$

Exercise 212. *Derive* $\left(\mathsf{lam}\left(\mathsf{lam}\left((\mathsf{q}[\mathsf{p}]) \cdot ((\mathsf{q}[\mathsf{p}]) \cdot \mathsf{q})\right)\right)\right)[\epsilon] = \mathsf{lam}\left(\mathsf{lam}\left((\mathsf{q}[\mathsf{p}]) \cdot ((\mathsf{q}[\mathsf{p}]) \cdot \mathsf{q})\right)\right)$.

Exercise 213. *Derive* $\left(\mathsf{lam}\left(\mathsf{lam}\left((\mathsf{q}[\mathsf{p}]) \cdot ((\mathsf{q}[\mathsf{p}]) \cdot \mathsf{q})\right)\right)\right) \cdot t \cdot u = t \cdot (t \cdot u)$.

4.2 The general translation

The recipe for translating a SOGAT into a GAT is the following:

- the GAT starts with a category with a terminal object,
- closed sorts become presheaves on this category,
- open sorts become dependent presheaves,
- operations become natural transformations (that is, operations indexed by contexts together with substitution laws),
- equations become equations indexed by contexts,
- sorts which are in U^+ (which appear on the left hand side of an arrow in an operator argument) moreover have local representability structure (that is, come with a context extension operation)
- second-order arguments become context extensions,
- Π^+-applications become explicit instantiations,
- Π^+-abstractions become weakenings,
- variables bound by Π^+-functions become De Bruijn indices.

Here we show more examples. For the precise algorithm, see [KX24].

In Definition 205, we used some cleverness: we knew that there are no second-order Ty-operators (no binders in Ty, no types refer to variables), so we made Ty a closed sort. The generic algorithm does not know about this (we might have dependent types), so the first part of the GAT will be simply the result of translating the SOGAT signature $\Sigma \cup \lambda T y.\, T y \Rightarrow \mathsf{U}^+$ (types and terms indexed by types where we only have term-variables):

Definition 214 (Category with family, CwF)**.**

Con	$: \mathsf{Set}$	$[\mathsf{id}]$	$: A[\mathsf{id}] = A$
Sub	$: \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}$	Tm	$: (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}$
$- \circ -$	$: \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Sub}\,\Theta\,\Delta \to \mathsf{Sub}\,\Theta\,\Gamma$	$-[-]$	$: \mathsf{Tm}\,\Gamma\,A \to (\gamma : \mathsf{Sub}\,\Delta\,\Gamma) \to \mathsf{Tm}\,\Delta\,(A[\gamma])$
ass	$: (\gamma \circ \delta) \circ \theta = \gamma \circ (\delta \circ \theta)$	$[\circ]$	$: a[\gamma \circ \delta] = a[\gamma][\delta]$
id	$: \mathsf{Sub}\,\Gamma\,\Gamma$	$[\mathsf{id}]$	$: a[\mathsf{id}] = a$
idl	$: \mathsf{id} \circ \gamma = \gamma$	$- \triangleright -$	$: (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con}$
idr	$: \gamma \circ \mathsf{id} = \gamma$	$-, -$	$: (\gamma : \mathsf{Sub}\,\Delta\,\Gamma) \to \mathsf{Tm}\,\Delta\,(A[\gamma]) \to \mathsf{Sub}\,\Delta\,(\Gamma \triangleright A)$
$\diamond$	$: \mathsf{Con}$	p	$: \mathsf{Sub}\,(\Gamma \triangleright A)\,\Gamma$
ϵ	$: \mathsf{Sub}\,\Gamma\,\diamond$	q	$: \mathsf{Tm}\,(\Gamma \triangleright A)\,(A[\mathsf{p}])$
$\diamond\eta$	$: (\sigma : \mathsf{Sub}\,\Gamma\,\diamond) \to \sigma = \epsilon$	$\triangleright\beta_1$	$: \mathsf{p} \circ (\gamma, a) = \gamma$
Ty	$: \mathsf{Con} \to \mathsf{Set}$	$\triangleright\beta_2$	$: \mathsf{q}[\gamma, a] = a$
$-[-]$	$: \mathsf{Ty}\,\Gamma \to \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Ty}\,\Delta$	$\triangleright\eta$	$: \sigma = (\mathsf{p} \circ \sigma, \mathsf{q}[\sigma])$
$[\circ]$	$: A[\gamma \circ \delta] = A[\gamma][\delta]$		

Exercise 215. *Define the syntax of CwF without using quotients or QIITs. This involves proving its induction principle.*

The unoptimised first-order version of STLC is the following.

Definition 216 (First-order model of STLC (unoptimised version), see Definition 54 and Example 201)**.** *We extend CwF (Definition 214) with the following components.*

$$\begin{array}{ll}
\iota & : \mathsf{Ty}\,\Gamma \\
\iota[] & : \iota[\gamma] = \iota \\
- \Rightarrow - & : \mathsf{Ty}\,\Gamma \to \mathsf{Ty}\,\Gamma \to \mathsf{Ty}\,\Gamma \\
\Rightarrow[] & : (A \Rightarrow B)[\gamma] = (A[\gamma]) \Rightarrow (B[\gamma]) \\
\mathsf{lam} & : \mathsf{Tm}\,(\Gamma \triangleright A)\,(B[\mathsf{p}]) \to \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) \\
\mathsf{lam}[] & : (\mathsf{lam}\,b)[\gamma] = \mathsf{lam}\,(b[\gamma \circ \mathsf{p}, \mathsf{q}])
\end{array}$$

$$\begin{array}{ll}
- \cdot - & : \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,B \\
\cdot[] & : (f \cdot a)[\gamma] = (f[\gamma]) \cdot (a[\gamma]) \\
\beta & : \mathsf{lam}\,b \cdot a = b[\mathsf{id}, a] \\
\eta & : f = \mathsf{lam}\,(f[\mathsf{p}] \cdot \mathsf{q}) \\
[\mathsf{id}] & : a[\mathsf{id}] = a
\end{array}$$

The unoptimised version of STLC has more models than the optimised version, but the syntaxes are equivalent.

Actually, Definition 54 does not completely determine the shape of the application operator in the first-order version. However, the formal definition by a SOGAT signature completely determines the first-order version. Here are the three different versions, we used the first one in Example 201.

second-order signature version

$$\mathsf{el}^+\,(Tm \cdot (arr \cdot A \cdot B)) \Rightarrow \mathsf{el}^+\,(Tm \cdot A) \Rightarrow \mathsf{El}\,(\mathsf{el}^+\,(Tm \cdot B))$$

$$\mathsf{el}^+\,(Tm \cdot (arr \cdot A \cdot B)) \Rightarrow \mathsf{El}\,(Tm \cdot A \Rightarrow^+ \mathsf{el}^+\,(Tm \cdot B))$$

$$\mathsf{El}\,(Tm \cdot (arr \cdot A \cdot B) \Rightarrow^+ Tm \cdot A \Rightarrow^+ \mathsf{el}^+\,(Tm \cdot B))$$

first-order version

$$\begin{array}{ll}
- \cdot - & : \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,B \\
\cdot[] & : (f \cdot a)[\gamma] = (f[\gamma]) \cdot (a[\gamma]) \\
\mathsf{app} & : \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) \to \mathsf{Tm}\,(\Gamma \triangleright A)\,(B[\mathsf{p}]) \\
\mathsf{app}[] & : (\mathsf{app}\,t)[\gamma \circ \mathsf{p}, \mathsf{q}] = \mathsf{app}\,(t[\gamma]) \\
\mathsf{APP} & : \mathsf{Tm}\,(\Gamma \triangleright A \Rightarrow B \triangleright A[\mathsf{p}])\,(B[\mathsf{p}][\mathsf{p}]) \\
\mathsf{APP}[] & : \mathsf{APP}[(\gamma \circ \mathsf{p}, \mathsf{q}) \circ \mathsf{p}, \mathsf{q}] = \mathsf{APP}
\end{array}$$

TODO: show that substitution for app is equivalent...

Definition 217 (First-order model of mini-MLTT, see Definition 187)**.** *We extend CwF (Definition 214) with the following components.*

$$\begin{array}{ll}
\iota & : \mathsf{Ty}\,\Gamma \\
\iota[] & : \iota[\gamma] = \iota \\
\Pi & : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \triangleright A) \to \mathsf{Ty}\,\Gamma \\
\Pi[] & : (\Pi\,A\,B)[\gamma] = \Pi\,(A[\gamma])\,(B[\gamma \circ \mathsf{p}, \mathsf{q}]) \\
\mathsf{lam} & : \mathsf{Tm}\,(\Gamma \triangleright A)\,B \to \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \\
\mathsf{lam}[] & : (\mathsf{lam}\,b)[\gamma] = \mathsf{lam}\,(b[\gamma \circ \mathsf{p}, \mathsf{q}])
\end{array}$$

$$\begin{array}{ll}
- \cdot - & : \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \to (a : \mathsf{Tm}\,\Gamma\,A) \to \mathsf{Tm}\,\Gamma\,(B[\mathsf{id}, a]) \\
\cdot[] & : (f \cdot a)[\gamma] = (f[\gamma]) \cdot (a[\gamma]) \\
\beta & : \mathsf{lam}\,b \cdot a = b[\mathsf{id}, a] \\
\eta & : f = \mathsf{lam}\,(f[\mathsf{p}] \cdot \mathsf{q}) \\
[\mathsf{id}] & : a[\mathsf{id}] = a
\end{array}$$

Exercise 218. *Show that the equations $\cdot[]$, β, η make sense (the two sides are in the same set (meta type)).*

Exercise 219 (long)**.** *What are the extra equations that we need to add to a CwF with Π, Σ, $\top$ to obtain something equivalent to cartesian closed categories? (CCC)*

TODO: constant types.

Example 220 (Models of any GAT form a (co)complete CwF)**.** *For any GAT, we obtain a category of models where objects are models and morphisms are homomorphisms. This category has the structure of a CwF which supports constant types, $\top$, Σ, extensional equality, booleans, quotient types, and an initial object. We list what*

some of the components are in this CwF.

Con	*model*
$\mathsf{Sub}\,\Delta\,\Gamma$	*morphism from Δ to Γ*
$\mathsf{Ty}\,\Gamma$	*dependent model over Γ*
$\mathsf{Tm}\,\Gamma\,A$	*dependent morphism from Γ to A*
$-[-] : \mathsf{Ty}\,\Gamma \to \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Ty}\,\Delta$	*reindexing a dependent model over Γ along a morphism from Δ to Γ*
$A[\gamma \circ \delta] = A[\gamma][\delta]$	*reindexing is functorial*
$\diamond$	*trivial model where all sorts are $\mathbb{1}$*
$- \triangleright - : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con}$	*total model from a model and a dep. model over it, sorts are meta Σs*
$\mathsf{p} : \mathsf{Sub}\,(\Gamma \triangleright A)\,\Gamma$	*first projection morphism from a total model*
$\Sigma : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma \triangleright A) \to \mathsf{Ty}\,\Gamma$	*the meta Σ of a dependent model and another dependent over it*
$(\Sigma\,A\,B)[\gamma] = \Sigma\,(A[\gamma])\,(B[\gamma \circ \mathsf{p}, \mathsf{q}])$	*total dependent models are stable under reindexing*
$\mathsf{K} : \mathsf{Con} \to \mathsf{Ty}\,\Gamma$	*a dependent model which does not actually depend on the base model*
$\mathsf{Eq} : \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Ty}\,\Gamma$	*a dependent model expressing equality of two dependent morphisms*
$\mathsf{I} : \mathsf{Con}$	*the syntax (initial model)*
$\mathit{iterator} : (\Gamma : \mathsf{Con}) \to \mathsf{Sub}\,\mathsf{I}\,\Gamma$	*iterator from the syntax to any model Γ*
$\mathit{induction} : (A : \mathsf{Ty}\,\mathsf{I}) \to \mathsf{Tm}\,\mathsf{I}\,A$	*induction principle of the syntax: a dependent morphism from I to any A*

Exercise 221 (very very long)**.** *Show that any GAT (element of* Ty *in the syntax of Definition 197) gives rise to a (co)complete CwF as given in Example 220. Hint: define a first-order model of Definition 197 where* Con *is CwF. For checking your solution, see [KKA19, Kov22].*

First-order logic is interesting because it has two different kinds of variables: term and proof variables, so there are two context extensions corresponding to these (there are no formula-variables). The only optimisation that we do is to omit equations for Pf which all hold by irr.

Definition 222 (First-order model of minimal intuitionistic first-order logic, see Definition 192)**.**

Con	$: \mathsf{Set}$
Sub	$: \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}$
$- \circ -$	$: \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Sub}\,\Theta\,\Delta \to \mathsf{Sub}\,\Theta\,\Gamma$
ass	$: (\gamma \circ \delta) \circ \theta = \gamma \circ (\delta \circ \theta)$
id	$: \mathsf{Sub}\,\Gamma\,\Gamma$
idl	$: \mathsf{id} \circ \gamma = \gamma$
idr	$: \gamma \circ \mathsf{id} = \gamma$
$\diamond$	$: \mathsf{Con}$
ϵ	$: \mathsf{Sub}\,\Gamma\,\diamond$
$\diamond\eta$	$: (\sigma : \mathsf{Sub}\,\Gamma\,\diamond) \to \sigma = \epsilon$
For	$: \mathsf{Con} \to \mathsf{Set}$
$-[-]$	$: \mathsf{For}\,\Gamma \to \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{For}\,\Delta$
$[\circ]$	$: A[\gamma \circ \delta] = A[\gamma][\delta]$
$[\mathsf{id}]$	$: A[\mathsf{id}] = A$
Tm	$: \mathsf{Con} \to \mathsf{Set}$
$-[-]$	$: \mathsf{Tm}\,\Gamma \to \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Tm}\,\Delta$
$[\circ]$	$: t[\gamma \circ \delta] = t[\gamma][\delta]$
$[\mathsf{id}]$	$: t[\mathsf{id}] = t$
$-\triangleright_{\mathsf{Tm}}$	$: \mathsf{Con} \to \mathsf{Con}$
$-\,,_{\mathsf{Tm}}\,-$	$: \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Tm}\,\Delta \to \mathsf{Sub}\,\Delta\,(\Gamma \triangleright_{\mathsf{Tm}})$
p_{Tm}	$: \mathsf{Sub}\,(\Gamma \triangleright_{\mathsf{Tm}})\,\Gamma$
q_{Tm}	$: \mathsf{Tm}\,(\Gamma \triangleright_{\mathsf{Tm}})$
$\triangleright_{\mathsf{Tm}}\beta_1$	$: \mathsf{p}_{\mathsf{Tm}} \circ (\gamma\,,_{\mathsf{Tm}}\,t) = \gamma$
$\triangleright_{\mathsf{Tm}}\beta_2$	$: \mathsf{q}_{\mathsf{Tm}}[\gamma\,,_{\mathsf{Tm}}\,t] = t$
$\triangleright_{\mathsf{Tm}}\eta$	$: \sigma = (\mathsf{p} \circ \sigma, \mathsf{q}[\sigma])$
$- \supset -$	$: \mathsf{For}\,\Gamma \to \mathsf{For}\,\Gamma \to \mathsf{For}\,\Gamma$
$\supset[]$	$: (A \supset B)[\gamma] = (A[\gamma]) \supset (B[\gamma])$
$\forall$	$: \mathsf{For}\,(\Gamma \triangleright_{\mathsf{Tm}}) \to \mathsf{For}\,\Gamma$
$\forall[]$	$: (\forall\,A)[\gamma] = \forall\,(A[\gamma \circ \mathsf{p}_{\mathsf{Tm}}\,,_{\mathsf{Tm}}\,\mathsf{q}_{\mathsf{Tm}}])$
Eq	$: \mathsf{Tm}\,\Gamma \to \mathsf{Tm}\,\Gamma \to \mathsf{For}\,\Gamma$
$\mathsf{Eq}[]$	$: (\mathsf{Eq}\,t\,t')[\gamma] = \mathsf{Eq}\,(t[\gamma])\,(t'[\gamma])$
Pf	$: (\Gamma : \mathsf{Con}) \to \mathsf{For}\,\Gamma \to \mathsf{Set}$
$-[-]$	$: \mathsf{Pf}\,\Gamma\,A \to (\gamma : \mathsf{Sub}\,\Delta\,\Gamma) \to \mathsf{Pf}\,\Delta\,(A[\gamma])$
irr	$: (p\,q : \mathsf{Pf}\,\Gamma\,A) \to p = q$
$-\triangleright_{\mathsf{Pf}}-$	$: (\Gamma : \mathsf{Con}) \to \mathsf{For}\,\Gamma \to \mathsf{Con}$
$-\,,_{\mathsf{Pf}}\,-$	$: (\gamma : \mathsf{Sub}\,\Delta\,\Gamma) \to \mathsf{Pf}\,\Delta\,(A[\gamma]) \to \mathsf{Sub}\,\Delta\,(\Gamma \triangleright_{\mathsf{Pf}} A)$
p_{Pf}	$: \mathsf{Sub}\,(\Gamma \triangleright_{\mathsf{Pf}} A)\,\Gamma$
q_{Pf}	$: \mathsf{Pf}\,(\Gamma \triangleright_{\mathsf{Pf}} A)\,(A[\mathsf{p}_{\mathsf{Pf}}])$
$\triangleright_{\mathsf{Pf}}\beta_1$	$: \mathsf{p}_{\mathsf{Pf}} \circ (\gamma\,,_{\mathsf{Pf}}\,p) = \gamma$
$\triangleright_{\mathsf{Pf}}\eta$	$: \sigma = (\mathsf{p}_{\mathsf{Pf}} \circ \sigma\,,_{\mathsf{Pf}}\,\mathsf{q}_{\mathsf{Pf}}[\sigma])$
$\mathsf{intro}_{\supset}$	$: \mathsf{Pf}\,(\Gamma \triangleright_{\mathsf{Pf}} A)\,(B[\mathsf{p}_{\mathsf{Pf}}]) \to \mathsf{Pf}\,\Gamma\,(A \supset B)$
$\mathsf{elim}_{\supset}$	$: \mathsf{Pf}\,\Gamma\,(A \supset B) \to \mathsf{Pf}\,\Gamma\,A \to \mathsf{Pf}\,\Gamma\,B$
$\mathsf{intro}_{\forall}$	$: \mathsf{Pf}\,(\Gamma \triangleright_{\mathsf{Tm}})\,A \to \mathsf{Pf}\,\Gamma\,(\forall\,A)$
$\mathsf{elim}_{\forall}$	$: \mathsf{Pf}\,\Gamma\,(\forall\,A) \to (t : \mathsf{Tm}\,\Gamma) \to \mathsf{Pf}\,\Gamma\,(A[\mathsf{id}\,,_{\mathsf{Tm}}\,t])$
$\mathsf{intro}_{\mathsf{Eq}}$	$: (t : \mathsf{Tm}\,\Gamma) \to \mathsf{Pf}\,\Gamma\,(\mathsf{Eq}\,t\,t)$
$\mathsf{elim}_{\mathsf{Eq}}$	$: \mathsf{Pf}\,\Gamma\,(A[\mathsf{id}, t]) \to \mathsf{Pf}\,(\mathsf{Eq}\,t\,t') \to \mathsf{Pf}\,\Gamma\,(A[\mathsf{id}, t'])$

The syntax of monoids over A is definable without quotients, see Exercise 46. The syntax of STLC is definable without quotients using normal forms and proving normalisation via hereditary substitution [KA10]. Definable quotients were studied by Nuo Li [Li15].

Exercise 223 (long)**.** *Show that the syntax of Definition 222 can be defined in Agda/Coq without quotients (see [Avr23]); we need* Pf *to be in the universe of propositions to justify the only equation in (the second-order version of) this theory.*

Quotients not definable via normal forms are e.g. the syntax of untyped combinator calculus or lambda calculus. The simplest example is unordered pairs of a given type A. There are theories where it is open whether they are definable without quotients:

Food for thought 224. *First-order logic is a theory without equations, and its first-order syntax is definable without quotients. In objective type theory [vdBdB21] (also called weak type theory [BW19], axiomatic type theory) there are also no equations, the equations are expressed as elements of the* Id *type. Although the second-order version of this theory does not have any equations, it is open whether its first-order syntax is definable without quotients.*

System F is interesting because there are two different kind of variables: type and term variables.

Definition 225 (First-order model of System F, see Definition 152)**.** *We extend CwF (Definition 214) with the following components.*

$$
\begin{array}{ll}
-\triangleright_{\mathsf{Ty}} & : \mathsf{Con} \to \mathsf{Con} \\
- ,_{\mathsf{Ty}} - & : \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Ty}\,\Delta \to \mathsf{Sub}\,\Delta\,(\Gamma\triangleright_{\mathsf{Ty}}) \\
\mathsf{p}_{\mathsf{Ty}} & : \mathsf{Sub}\,(\Gamma\triangleright_{\mathsf{Ty}})\,\Gamma \\
\mathsf{q}_{\mathsf{Ty}} & : \mathsf{Tm}\,(\Gamma\triangleright_{\mathsf{Ty}})\,(A[\mathsf{p}_{\mathsf{Ty}}]) \\
\triangleright_{\mathsf{Ty}}\beta_1 & : \mathsf{p}_{\mathsf{Ty}} \circ (\gamma ,_{\mathsf{Ty}} A) = \gamma \\
\triangleright_{\mathsf{Ty}}\beta_2 & : \mathsf{q}_{\mathsf{Ty}}[\gamma ,_{\mathsf{Ty}} A] = A \\
\triangleright_{\mathsf{Ty}}\eta & : \sigma = (\mathsf{p}_{\mathsf{Ty}} \circ \sigma ,_{\mathsf{Ty}} \mathsf{q}_{\mathsf{Ty}}[\sigma])
\end{array}
\qquad
\begin{array}{ll}
\forall & : \mathsf{Ty}\,(\Gamma\triangleright_{\mathsf{Ty}}) \to \mathsf{Ty}\,\Gamma \\
\forall[] & : (\forall\,A)[\gamma] = \forall\,(A[\gamma \circ \mathsf{p}_{\mathsf{Ty}} ,_{\mathsf{Ty}} \mathsf{q}_{\mathsf{Ty}}]) \\
\mathsf{Lam} & : \mathsf{Tm}\,(\Gamma\triangleright_{\mathsf{Ty}})\,A \to \mathsf{Tm}\,\Gamma\,(\forall\,A) \\
\mathsf{Lam}[] & : (\mathsf{Lam}\,a)[\gamma] = \mathsf{Lam}\,(a[\gamma \circ \mathsf{p}_{\mathsf{Ty}} ,_{\mathsf{Ty}} \mathsf{q}_{\mathsf{Ty}}]) \\
- \bullet - & : \mathsf{Tm}\,\Gamma\,(\forall\,A) \to (B : \mathsf{Ty}\,\Gamma) \to \mathsf{Tm}\,\Gamma\,(A[\mathsf{id} ,_{\mathsf{Ty}} B]) \\
\bullet[] & : (t \bullet B)[\gamma] = (t[\gamma]) \bullet (B[\gamma]) \\
\forall\beta & : (\mathsf{Lam}\,a) \bullet B = a[\mathsf{id} ,_{\mathsf{Ty}} B] \\
\forall\eta & : t = \mathsf{Lam}\,(t[\mathsf{p}_{\mathsf{Ty}}] \bullet \mathsf{q}_{\mathsf{Ty}})
\end{array}
$$

In System F_ω, we still have two extensions, but type variables are now indexed with their kinds, so their context extension operation is more interesting. We could optimise the following definition such that Kind does not depend on Con. We write the universal properties in concise form.

Definition 226 (First-order model of System F_ω, see Definition 165)**.**

$\mathsf{Con} : \mathsf{Set}$
$\mathsf{Sub} : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}$
$- \circ - : \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Sub}\,\Theta\,\Delta \to \mathsf{Sub}\,\Theta\,\Gamma$
$\mathsf{ass} : (\gamma \circ \delta) \circ \theta = \gamma \circ (\delta \circ \theta)$
$\mathsf{id} : \mathsf{Sub}\,\Gamma\,\Gamma$
$\mathsf{idl} : \mathsf{id} \circ \gamma = \gamma$
$\mathsf{idr} : \gamma \circ \mathsf{id} = \gamma$
$\diamond : \mathsf{Con}$
$\epsilon : \mathsf{Sub}\,\Gamma\,\diamond$
$\diamond\eta : (\sigma : \mathsf{Sub}\,\Gamma\,\diamond) \to \sigma = \epsilon$
$\mathsf{Kind} : \mathsf{Con} \to \mathsf{Set}$
$-[-] : \mathsf{Kind}\,\Gamma \to \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Kind}\,\Delta$
$[\circ] : K[\gamma \circ \delta] = K[\gamma][\delta]$
$[\mathsf{id}] : K[\mathsf{id}] = K$
$\mathsf{Ty} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}$
$-[-] : \mathsf{Ty}\,\Gamma\,K \to (\gamma : \mathsf{Sub}\,\Delta\,\Gamma) \to \mathsf{Ty}\,\Delta\,(K[\gamma])$
$[\circ] : A[\gamma \circ \delta] = A[\gamma][\delta]$
$[\mathsf{id}] : A[\mathsf{id}] = A$
$-\triangleright_{\mathsf{Ty}} - : (\Gamma : \mathsf{Con}) \to \mathsf{Kind}\,\Gamma \to \mathsf{Con}$
$-,_{\mathsf{Ty}} - : (\gamma : \mathsf{Sub}\,\Delta\,\Gamma) \times \mathsf{Ty}\,\Delta\,(K[\gamma]) \to \mathsf{Sub}\,\Delta\,(\Gamma \triangleright_{\mathsf{Ty}} K)$
$\mathsf{p}_{\mathsf{Ty}} : \mathsf{Sub}\,(\Gamma \triangleright_{\mathsf{Ty}} K)\,\Gamma$
$\mathsf{q}_{\mathsf{Ty}} : \mathsf{Ty}\,(\Gamma \triangleright_{\mathsf{Ty}} K)\,(K[\mathsf{p}_{\mathsf{Ty}}])$
$\triangleright_{\mathsf{Ty}}\beta_1 : \mathsf{p}_{\mathsf{Ty}} \circ (\gamma ,_{\mathsf{Ty}} A) = \gamma$
$\triangleright_{\mathsf{Ty}}\beta_2 : \mathsf{q}_{\mathsf{Ty}}[\gamma ,_{\mathsf{Ty}} A] = A$
$\triangleright_{\mathsf{Ty}}\eta : \sigma = (\mathsf{p} \circ \sigma ,_{\mathsf{Ty}} \mathsf{q}_{\mathsf{Ty}}[\sigma])$
$- \Rrightarrow - : \mathsf{Kind}\,\Gamma \to \mathsf{Kind}\,\Gamma \to \mathsf{Kind}\,\Gamma$
$\Rrightarrow[] : (K \Rrightarrow L)[\gamma] = (K[\gamma]) \Rrightarrow (L[\gamma])$
$\mathsf{LAM} : \mathsf{Ty}\,(\Gamma \triangleright_{\mathsf{Ty}} K)\,L \to \mathsf{Ty}\,\Gamma\,(K \Rrightarrow L)$
$\mathsf{LAM}[] : (\mathsf{LAM}\,A)[\gamma] = \mathsf{LAM}\,(A[\gamma \circ \mathsf{p}_{\mathsf{Ty}} ,_{\mathsf{Ty}} \mathsf{q}_{\mathsf{Ty}}])$
$- \bullet - : \mathsf{Ty}\,\Gamma\,(K \Rrightarrow L) \to \mathsf{Ty}\,\Gamma\,K \to \mathsf{Ty}\,\Gamma\,L$
$\bullet[] : (F \bullet B)[\gamma] = (F[\gamma]) \bullet (B[\gamma])$
$\Rrightarrow\beta : \mathsf{LAM}\,A \bullet B = A[\mathsf{id} ,_{\mathsf{Ty}} B]$
$\Rrightarrow\eta : F = \mathsf{LAM}\,(F[\mathsf{p}_{\mathsf{Ty}}] \bullet \mathsf{q}_{\mathsf{Ty}})$
$* : \mathsf{Kind}\,\Gamma$
$*[] : *[\gamma] = *$
$\mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma * \to \mathsf{Set}$
$-[-] : \mathsf{Tm}\,\Gamma\,A \to (\gamma : \mathsf{Sub}\,\Delta\,\Gamma) \to \mathsf{Tm}\,\Delta\,(A[\gamma])$
$[\circ] : a[\gamma \circ \delta] = a[\gamma][\delta]$
$[\mathsf{id}] : a[\mathsf{id}] = a$
$-,_{\mathsf{Tm}} - : (\gamma : \mathsf{Sub}\,\Delta\,\Gamma) \times \mathsf{Tm}\,\Delta\,(A[\gamma]) \to \mathsf{Sub}\,\Delta\,(\Gamma \triangleright_{\mathsf{Tm}} A)$
$\mathsf{p}_{\mathsf{Tm}} : \mathsf{Sub}\,(\Gamma \triangleright_{\mathsf{Tm}} A)\,\Gamma$
$\mathsf{q}_{\mathsf{Tm}} : \mathsf{Tm}\,(\Gamma \triangleright_{\mathsf{Tm}} A)\,(A[\mathsf{p}_{\mathsf{Tm}}])$
$\triangleright_{\mathsf{Tm}}\beta_1 : \mathsf{p}_{\mathsf{Tm}} \circ (\gamma ,_{\mathsf{Tm}} a) = \gamma$
$\triangleright_{\mathsf{Tm}}\beta_2 : \mathsf{q}_{\mathsf{Tm}}[\gamma ,_{\mathsf{Tm}} a] = a$
$\triangleright_{\mathsf{Tm}}\eta : \sigma = (\mathsf{p} \circ \sigma ,_{\mathsf{Tm}} \mathsf{q}_{\mathsf{Tm}}[\sigma])$
$\forall : \mathsf{Ty}\,(\Gamma \triangleright_{\mathsf{Ty}} K) * \to \mathsf{Ty}\,\Gamma *$
$\forall[] : (\forall\,A)[\gamma] = \forall\,(A[\gamma \circ \mathsf{p}_{\mathsf{Ty}} \circ_{\mathsf{Ty}} \mathsf{q}_{\mathsf{Ty}}])$
$\mathsf{Lam} : \mathsf{Tm}\,(\Gamma \triangleright_{\mathsf{Ty}} K)\,A \to \mathsf{Tm}\,\Gamma\,(\forall\,A)$
$\mathsf{Lam}[] : (\mathsf{Lam}\,a)[\gamma] = \mathsf{Lam}\,(a[\gamma \circ \mathsf{p}_{\mathsf{Ty}} ,_{\mathsf{Ty}} \mathsf{q}_{\mathsf{Ty}}])$
$- \bullet - : \mathsf{Tm}\,\Gamma\,(\forall\,A) \to (B : \mathsf{Ty}\,\Gamma\,K) \to \mathsf{Tm}\,\Gamma\,(A[\mathsf{id} ,_{\mathsf{Ty}} B])$
$\bullet[] : (t \bullet A)[\gamma] = (t[\gamma]) \bullet (A[\gamma])$
$\forall\beta : (\mathsf{Lam}\,a) \bullet B = a[\mathsf{id} ,_{\mathsf{Ty}} B]$
$\forall\eta : t = \mathsf{Lam}\,(t[\mathsf{p}_{\mathsf{Ty}}] \bullet \mathsf{q}_{\mathsf{Ty}})$
$- \Rightarrow - : \mathsf{Ty}\,\Gamma * \to \mathsf{Ty}\,\Gamma * \to \mathsf{Ty}\,\Gamma *$
$\Rightarrow[] : (A \Rightarrow B)[\gamma] = (A[\gamma]) \Rightarrow (B[\gamma])$
$\mathsf{lam} : \mathsf{Tm}\,(\Gamma \triangleright_{\mathsf{Tm}} A)\,B \to \mathsf{Tm}\,\Gamma(A \Rightarrow B)$
$\mathsf{lam}[] : (\mathsf{lam}\,b)[\gamma] = \mathsf{lam}\,(b[\gamma \circ \mathsf{p}_{\mathsf{Tm}} ,_{\mathsf{Tm}} \mathsf{q}_{\mathsf{Tm}}])$
$- \cdot - : \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,B$
$\cdot[] : (t \cdot a)[\gamma] = (t[\gamma]) \cdot (a[\gamma])$
$\Rightarrow\beta : (\mathsf{lam}\,b) \cdot a = b[\mathsf{id}, a]$
$\Rightarrow\eta : t = \mathsf{lam}\,(t[\mathsf{p}_{\mathsf{Tm}}] \cdot \mathsf{q}_{\mathsf{Tm}})$

Exercise 227. *Check that instantiation for terms make sense (is well-typed in the metatheory).*

TODO: add MLTT with equality etc.

5 Admissibility in SOGATs

It is not a bad position to define languages abstractly as SOGATs, and then when proving properties of the language, we have to work with its more verbose translated GAT-version. It is nice that all the substitution rules are generated automatically, and we don't have to worry that we missed one or wrote down one incorrectly.

However, we can do better. Some proofs about first-order models can be factored through a generic sconing construction.

In this section, we fix the SOGAT to mini-MLTT: its second-order model is Definition 187, its first-order model is Definition 217.

5.1 Proofs about closed syntactic terms

I denotes the first-order model of mini-MLTT.

Definition 228 (Second-order dependent model)**.**

$$
\begin{aligned}
&\mathsf{Ty} && : \mathsf{Ty}_\mathsf{I}\,\diamond_\mathsf{I} \to \mathsf{Set}\\
&\mathsf{Tm} && : \{A_\mathsf{I} : \mathsf{Ty}_\mathsf{I}\,\diamond_\mathsf{I}\} \to \mathsf{Ty}\,A_\mathsf{I} \to \mathsf{Tm}_\mathsf{I}\,\diamond_\mathsf{I} A_\mathsf{I} \to \mathsf{Set}\\
&\iota && : \mathsf{Ty}\,(\iota_\mathsf{I}\,\{\diamond_\mathsf{I}\})\\
&\Pi && : \{A_\mathsf{I} : \mathsf{Ty}_\mathsf{I}\,\diamond_\mathsf{I}\}(A : \mathsf{Ty}\,A_\mathsf{I})\{B_\mathsf{I} : \mathsf{Ty}_\mathsf{I}\,(\diamond_\mathsf{I} \triangleright_\mathsf{I} A_\mathsf{I})\} \to (\{a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\,\diamond_\mathsf{I} A_\mathsf{I}\} \to \mathsf{Tm}\,A_\mathsf{I}\,a_\mathsf{I} \to \mathsf{Ty}\,(B_\mathsf{I}[\epsilon_\mathsf{I} ,_\mathsf{I} a_\mathsf{I}]_\mathsf{I})) \to \mathsf{Ty}\,(\Pi_\mathsf{I}\,A_\mathsf{I}\,B_\mathsf{I})\\
&\mathsf{lam} && : \big((a : \mathsf{Tm}\,A\,a_\mathsf{I}) \to \mathsf{Tm}\,(B\,a)\,(b_\mathsf{I}[\epsilon_\mathsf{I} ,_\mathsf{I} a_\mathsf{I}]_\mathsf{I})\big) \to \mathsf{Tm}\,(\Pi\,A\,B)\,(\mathsf{lam}_\mathsf{I}\,b_\mathsf{I})\\
&-\cdot- && : \mathsf{Tm}\,(\Pi\,A\,B)\,f_\mathsf{I} \to (a : \mathsf{Tm}\,A\,a_\mathsf{I}) \to \mathsf{Tm}\,(B\,a)\,(f_\mathsf{I} \cdot_\mathsf{I} a_\mathsf{I})\\
&\beta && : \mathsf{lam}\,b \cdot a = b\,a\\
&\eta && : f = \mathsf{lam}\,\lambda a.\, f \cdot a
\end{aligned}
$$

The left hand side of β has type $\mathsf{Tm}\,(B\,a)\,(\mathsf{lam}_\mathsf{I}\,b_\mathsf{I} \cdot_\mathsf{I} a_\mathsf{I})$, the right hand side has type $\mathsf{Tm}\,(B\,a)\,(b_\mathsf{I}[\epsilon_\mathsf{I} ,_\mathsf{I} a_\mathsf{I}]_\mathsf{I})$, which are equal by β_I.

Exercise 229. *Similarly check that the two sides of η are in the same set.*

The above dependent model only says something about closed types and closed terms. We have a general construction to make it work for any type and term:

Definition 230 (Scone-contextualisation)**.** *Assuming a second-order dependent model, we obtain the following first-order dependent model.*

$$
\begin{aligned}
&\mathsf{Con}\,\Gamma_\mathsf{I} && := \mathsf{Sub}_\mathsf{I}\,\diamond_\mathsf{I}\,\Gamma_\mathsf{I} \to \mathsf{Set}\\
&\mathsf{Sub}\,\Delta\,\Gamma\,\gamma_\mathsf{I} && := \Delta\,\delta_\mathsf{I} \to \Gamma\,(\gamma_\mathsf{I} \circ_\mathsf{I} \delta_\mathsf{I})\\
&\delta \circ \gamma && := \lambda\theta_*.\, \delta\,(\gamma\,\theta_*)\\
&\mathsf{id} && := \lambda\gamma_*.\, \gamma_*\\
&\diamond && := \lambda\_.\, \mathbb{1}\\
&\mathsf{Ty}\,\Gamma\,A_\mathsf{I} && := (\gamma_* : \Gamma\,\gamma_\mathsf{I}) \to \mathsf{Ty}\,(A_\mathsf{I}[\gamma_\mathsf{I}]_\mathsf{I})\\
&A[\gamma] && := \lambda\delta_*.\, A\,(\gamma\,\delta_*)\\
&\mathsf{Tm}\,\Gamma\,A\,a_\mathsf{I} && := (\gamma_* : \Gamma\,\gamma_\mathsf{I}) \to \mathsf{Tm}\,(A\,\gamma_*)\,(a_\mathsf{I}[\gamma_\mathsf{I}]_\mathsf{I})\\
&a[\gamma] && := \lambda\delta_*.\, a\,(\gamma\,\delta_*)\\
&\Gamma \triangleright A && := \lambda(\gamma_\mathsf{I} ,_\mathsf{I} a_\mathsf{I}).\, (\gamma_* : \Gamma\,\gamma_\mathsf{I}) \times \mathsf{Tm}\,(A\,\gamma_\mathsf{I})\,a_\mathsf{I}\\
&\gamma, a && := \lambda\delta_*.\, (\gamma\,\delta_*, a\,\delta_*)\\
&\mathsf{p} && := \lambda(\gamma_*, a_*).\, \gamma_*\\
&\mathsf{q} && := \lambda(\gamma_*, a_*).\, a_*\\
&\Pi\,A\,B && := \lambda\gamma_*.\, \Pi\,(A\,\gamma_*)\,(\lambda a_*.\, B\,(\gamma_*, a_*))\\
&\mathsf{lam}\,b && := \lambda\gamma_*.\, \mathsf{lam}\,(\lambda a_*.\, b\,(\gamma_*, a_*))\\
&f \cdot a && := \lambda\gamma_*.\, f\,\gamma_* \cdot a\,\gamma_*
\end{aligned}
$$

Example 231 (Canonicity for mini-MLTT)**.** *We define the following second-order dependent model:*

$$
\begin{aligned}
&\mathsf{Ty}\,A_\mathsf{I} && := \mathsf{Tm}_\mathsf{I}\,\diamond_\mathsf{I}\,A_\mathsf{I} \to \mathsf{Set}\\
&\mathsf{Tm}\,A\,a_\mathsf{I} && := A\,a_\mathsf{I}\\
&\iota && := \lambda t_\mathsf{I}.\, \mathbb{0}\\
&\Pi\,A\,B && := \lambda f_\mathsf{I}.\, \{a_\mathsf{I} : \mathsf{Tm}_\mathsf{I}\,\diamond_\mathsf{I} A_\mathsf{I}\}(a_* : A\,a_\mathsf{I}) \to B\,a_*\,(f_\mathsf{I} \cdot_\mathsf{I} a_\mathsf{I})\\
&\mathsf{lam}\,b && := b\\
&f \cdot a && := f\,a
\end{aligned}
$$

This is the canonicity proof without boilerplate. From this, we get e.g. that the set $\mathsf{Tm}_\mathsf{I}\,\diamond_\mathsf{I}\,\iota_\mathsf{I}$ is empty: applying (induction into the dependent scone-contextualisation of the canonicity second-order dependent model) on such a t_I we obtain an element of $(\sigma_* : \diamond\,\sigma_\mathsf{I}) \to \mathsf{Tm}\,(\iota\,\sigma_*)\,(t_\mathsf{I}[\sigma_\mathsf{I}]_\mathsf{I}) = \mathbb{1} \to \mathbb{0}$.

5.2 Internal languages of presheaf models of MLTT

If we say that a model of a SOGAT is a model of its first-orderification, then did the derived things in Section 3 made sense?

How do we know that the SOGAT $\longrightarrow$ GAT translation is the correct one? One requirement is that they can be used to build the same things: if we build something assuming a second-order model, then the corresponding first-order thing should be also buildable using De Bruijn combinators and explicit weakenings etc. How do we make this sure?

The derivable operations and equations of Section 3 are also derivable in the result of the translations.

5.3 Proofs about open syntactic terms

Renamings.

Internal to presheaves over renamings, we have a first-order model which we also denote I.

References

[Ack28] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133, 1928. URL: http://eudml.org/doc/159248.

[ACKS23] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. *Math. Struct. Comput. Sci.*, 33(8):688–743, 2023. URL: https://doi.org/10.1017/s0960129523000130, doi:10.1017/S0960129523000130.

[ACKS24] Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. Internal parametricity, without an interval. *Proc. ACM Program. Lang.*, 8(POPL):2340–2369, 2024. doi:10.1145/3632920.

[AKSV23] Thorsten Altenkirch, Ambrus Kaposi, Artjoms Sinkarovs, and Tamás Végh. Combinatory logic and lambda calculus are equal, algebraically. In Marco Gaboardi and Femke van Raamsdonk, editors, *8th International Conference on Formal Structures for Computation and Deduction, FSCD 2023, July 3-6, 2023, Rome, Italy*, volume 260 of *LIPIcs*, pages 24:1–24:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.FSCD.2023.24, doi:10.4230/LIPICS.FSCD.2023.24.

[Alt08] Thorsten Altenkirch. From high school to university algebra. https://people.cs.nott.ac.uk/psztxa/publ/unialg.pdf, 2008.

[Avr23] Samy Avrillon. Logic as a second-order generalized algebraic theory, 2023. Report on the 3-month research internship at the Faculty of Informatics of ELTE. URL: https://github.com/MysaaJava/m1-internship/releases/download/project-report/Avrillon-02.pdf.

[Awo10] S. Awodey. *Category Theory*. Oxford Logic Guides. OUP Oxford, 2010. URL: http://books.google.co.uk/books?id=-MCJ6x2lC7oC.

[Bar91] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991. doi:10.1017/S0956796800020025.

[BKS23] Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. For the metatheory of type theory, internal sconing is enough. In Marco Gaboardi and Femke van Raamsdonk, editors, *8th International Conference on Formal Structures for Computation and Deduction, FSCD 2023, July 3-6, 2023, Rome, Italy*, volume 260 of *LIPIcs*, pages 18:1–18:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.FSCD.2023.18.

[BL93] Stanley Burris and Simon Lee. Tarski's high school identities. *The American Mathematical Monthly*, 100(3):231–236, 1993. URL: http://www.jstor.org/stable/2324454.

[BSvB24] Ulrik Torben Buchholtz and Johannes Schipp von Branitz. Primitive recursive dependent type theory. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3661814.3662136.

[BW19] Simon Boulier and Théo Winterhalter. Weak type theory is rather strong. In Marc Bezem, editor, *25th International Conference on Types for Proofs and Programs, TYPES 2019*. Centre for Advanced Study at the Norwegian Academy of Science and Letters, 2019. URL: https://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_18.

[CD07] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007. doi:10.1007/978-3-540-73228-0_9.

[Csö12] Zoltán Csörnyei. *Bevezetés a típusrendszerek elméletébe*. ELTE Eötvös Kiadó, 2012.

[DL11] Gilles Dowek and Jean-Jacques Lévy. *Introduction to the Theory of Programming Languages*. Undergraduate Topics in Computer Science. Springer, 2011. doi:10.1007/978-0-85729-076-2.

[Gir72] Jean-Yves Girard. *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016.

[HHP93] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993. doi:10.1145/138027.138060.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. URL: http://www.jstor.org/stable/1995158.

[Hof99] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 204–213. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782616.

[Hut23] Graham Hutton. Programming Language Semantics: It's Easy As 1,2,3. *Journal of Functional Programming*, 33, October 2023.

[KA10] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In Venanzio Capretta and James Chapman, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, pages 3–10. ACM, 2010. doi:10.1145/1863597.1863601.

[Kap19] Ambrus Kaposi. Reduction of indexed w-types to w-types, an agda formalisation. https://akaposi.github.io/IW.agda, 2019.

[KKA19] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL):2:1–2:24, 2019. doi:10.1145/3290315.

[Kov22] András Kovács. *Type-Theoretic Signatures for Algebraic Theories and Inductive Types*. PhD thesis, Eötvös Loránd University, Hungary, 2022. URL: https://arxiv.org/pdf/2302.08837.pdf.

[KvR20] Ambrus Kaposi and Jakob von Raumer. A syntax for mutual inductive families. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. URL: https://doi.org/10.4230/LIPIcs.FSCD.2020.23, doi:10.4230/LIPICS.FSCD.2020.23.

[KX24] Ambrus Kaposi and Szumi Xie. Second-order generalised algebraic theories: Signatures and first-order semantics. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia*, volume 299 of *LIPIcs*, pages 10:1–10:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: https://doi.org/10.4230/LIPIcs.FSCD.2024.10, doi:10.4230/LIPICS.FSCD.2024.10.

[Li15] Nuo Li. *Quotient types in type theory*. Thesis. University of Nottingham, Department of Computer Science, 2015. URL: http://eprints.nottingham.ac.uk/28941.

[Lor25] Jem Lord. Easy parametricity. Workshop on Homotopy Type Theory / Univalent Foundations 2025, 2025. URL: https://hott-uf.github.io/2025/abstracts/HoTTUF_2025_paper_21.pdf.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

[Moe22] Hugo Moeneclaey. *Cubical models are cofreely parametric. (Les modèles cubiques sont colibrement paramétriques)*. PhD thesis, Paris Cité University, France, 2022. URL: https://tel.archives-ouvertes.fr/tel-04435596.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, July 1988. doi:10.1145/44501.45065.

[Par89] Michel Parigot. On the representation of data in lambda-calculus. In Egon Börger, Hans Kleine Büning, and Michael M. Richter, editors, *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, volume 440 of *Lecture Notes in Computer Science*, pages 309–321. Springer, 1989. doi:10.1007/3-540-52753-2_47.

[PdAC+23] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook, 2023. Version 6.5, http://softwarefoundations.cis.upenn.edu.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[Rey74] John C. Reynolds. Towards a theory of type structure. In Bernard J. Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974. doi:10.1007/3-540-06859-7_148.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[Sch24] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3):305–316, Sep 1924. doi:10.1007/BF01448013.

[Sch17] Gabriel Scherer. Deciding equivalence with sums and the empty type. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 374–386. ACM, 2017. doi:10.1145/3009837.3009901.

[Ste22] Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University, USA, 2022. URL: https://doi.org/10.1184/r1/19632681.v1, doi:10.1184/R1/19632681.V1.

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi:10.2307/2271658.

[vdBdB21] Benno van den Berg and Martijn den Besten. Quadratic type checking for objective type theory. *CoRR*, abs/2102.00905, 2021. URL: https://arxiv.org/abs/2102.00905, arXiv:2102.00905.

[Wad89] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989. doi:10.1145/99370.99404.

[Wad90] Philip Wadler. Recursive types for free! https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt, 1990.

[Wil01] Alex Wilkie. On exponentiation - a solution to tarski's high school algebra problem. *Quaderni di Matematica*, 6, 02 2001.

[WKS22] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.