

1 **Implementation of Type Theory based on**
2 **dependent Inductive and Coinductive**
3 **Types**

4 Florian Engel

5 November 17, 2020



9 Masterarbeit

10 **Implementation of Type Theory based on**
11 **dependent inductive and coinductive types**

12 Eberhard Karls Universität Tübingen

13 Mathematisch-Naturwissenschaftliche Fakultät

14 Wilhelm-Schickard-Institut für Informatik

15 Programiersprachen

16 Florian Engel, florian.engel@student.uni-tuebingen.de, 2020

Bearbeitungszeitraum: von-bis

17

Betreuer/Gutachter: Prof. Dr. Klaus Ostermann, Universität Tübingen

Zweitgutachter: Prof. Dr. Max Mustermann, Universität Tübingen

18 **Selbstständigkeitserklärung**

19 Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur
20 mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem
21 Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von
22 Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde
23 in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung
24 vorgelegt.

25 Florian Engel (Matrikelnummer 3860700), November 17, 2020

26 **Abstract**

27 Dependent types are an useful tool to restrict types even further then types of
28 strongly typed languages like Haskell. This gives us further type safety. With them
29 we can also proof theorems. Coinductive types allow us to define types by their
30 observations rather then by their constructors. This is useful for infinite types like
31 streams. In many common dependently typed languages , like coq and agda, we
32 can define inductive types which depend on values and coinductive types but not
33 coinductive types, which depend on values.

34 In this work I will first give a survey of coinductive types in these languages and
35 then implement the type theory from [BG16]. This type theory has both dependent
36 inductive types and dependent coinductive types. In this type theory the dependent
37 function space becomes definable. This leads to a more symmetrical approach of
38 coinduction in dependently typed languages.

Contents

40	1. Introduction	11
41	2. Coinductive Types	15
42	3. Coinductive Types in dependent languages	17
43	3.1. Coinductive Types in Coq	19
44	3.1.1. Postive Coinductive Types	19
45	3.1.2. Negative Coinductive Types	21
46	3.2. Coinductive Types in Agda	23
47	3.2.1. Positive Coinductive Types in Agda	23
48	3.2.2. Negative Coinductive Types in Agda	24
49	3.2.3. Termination Checking with Sized Types	26
50	4. Type Theory based on dependent Inductive and Coinductive Types	29
51	5. Implementation	31
52	5.1. Abstract Syntax	31
53	5.1.1. Declarations	31
54	5.1.2. Expressions	33
55	5.2. Substitution	35
56	5.3. Typing rules	36
57	5.3.1. Context rules	37
58	5.3.2. Beta-equivalence	38
59	5.3.3. Unit type and expression introduction	39
60	5.3.4. Variable lookup	39
61	5.3.5. Type and expression instantiation	40
62	5.3.6. Parameter abstraction	41
63	5.3.7. (co)inductive types	41
64	5.3.8. Constructor and Destructor	42
65	5.3.9. Recursion and Corecursion	43
66	5.4. Evaluation	44
67	6. Examples	47
68	6.1. Terminal and Initial Object	47
69	6.2. Natural Numbers and Extended Naturals	47
70	6.3. Binary Product and Coproduct	49

Contents

71	6.4. Sigma and Pi Type	50
72	6.5. Vectors and Streams	51
73	7. Conclusion	55
74	A. Type action proof	57
75	B. Bigger proofs	63

1. Introduction

In functional programming we have functions which get an input and produce an output. These functions don't depend on mutable values i.e. if they are no IO involved they produce for the same input always the same output. For example if we call a function `or` on the values `true` and `false` we always get `true`. This makes the code more predictable.

The `or` function should only be working on booleans. To call it on strings `'foo'` and `'bar'`, wouldn't make sense i.e. there is no defined output for these inputs. To prevent calls like these some functional programming languages introduced types. Types contain only certain values. For example the type for truth values contains only the values for true and false. In Haskell we can define it like the following.

```
data Bool = True | False
```

This says we can construct values of type `Bool` with the constructors `True` and `False`. These types which have constructors are called inductive types. We can then define `or` like this.

```
or :: Bool -> Bool -> Bool
or True _   = True
or _   True = True
or _   _   = False
```

Here we just list equations which say what the output for a given input is. For example in the first equation, we say if the first value is constructed with the constructor `True` we give back `True`. We don't care about the second value, therefore we write `_`. We are matching on the construction of the input values. Therefore we call this method pattern matching. If we call this function somewhere in the code on values which aren't of type `Bool`, Haskell won't compile our code. Instead it gives back a type error.

If we now want to change `Bool` to a three-valued logic, we have to add an third constructor to `Bool`. After that we have to change every function which pattern matches on `Bool`. If we have many that would be a lot of repetitive work. If Haskell would have coinductive type, this could be a lot less work. Coinductive types are types which are, in contrary to inductive types defined over their destruction. So we could define `Bool` over its destructors. These would be `or`, `and`, etc.

Through this work we will explain coinductive types at the examples of streams and functions. They will be generalized to partial streams and the `Pi` type in dependently typed languages. Streams are lists which are infinitely long. They are useful to

modeling many IO interaction. For example a chat of a text messenger might be infinitely long. We can never know if the chat is finished. This is of course limited by the hardware, but we are interested in abstract models. Functions are used everywhere in functional programming. In most of these languages they are first-class objects. But in languages with coinductive types we can define them. If we only have inductive and coinductive types, we get a symmetrical language. This is useful, because then we can change an inductive type to a coinductive one and vice versa. It is straight forward to add function which destruct an inductive type by pattern matching on the constructor. But it is hard to add a new constructor. We then have to add this constructor to every pattern matching on that type. For coinductive types its the other way around. For more on this see [BJSO19]. In the implemented syntax we can define streams like the following.

```

118 codata Stream(A : Set) : Set where Hd : Stream (succ @ k)
119     Hd : Stream → A
120     Tl : Stream → Stream

```

And functions like follows.

```

122 codata Fun(A : Set, B : Set) : Set where
123     Inst : (x : A) → Fun → B

```

We can generalize streams to partial streams as the following.

```

125 codata PStr(A : Set) : (n : Conat) → Set where
126     Hd : (k : Conat) → PStr (succ @ k) → A
127     Tl : (k : Conat) → PStr (succ @ k) → PStr @ k

```

And functions to the Pi type.

```

129 codata Pi(A : Set, B : (x : A) → Set) : Set where
130     Inst : (x : A) → Pi → B @ x

```

The rest of this thesis will be structured as follows.

- In chapter 2 we will see how coinductive types can be defined. Here we will be defining the stream and function type. We will also define some functions on the stream.
- We will see in chapter 3 how coinductive types are defined in the dependently typed languages coq and agda. We will see that we can define them positive or negative. We will show why defining them positive leads to problems.
- In chapter 4 we see how they are defined in [BG16]. With this theory we can then define coinductive types which depend on values. But we can not define types which depend on types.
- We will then in chapter 5 explain how this theory is implemented. Therefore we need to rewrite the typing rules. It will also be possible to define type schemata.

144 • At last we look at the examples from this paper in the implemented syntax.
145 Here we will see the reduction steps for recursion and corecursion. We will con-
146 clude this section with the example of partial streams, which is a coinductive
147 type which depends on a value.

148 2. Coinductive Types

149 Inductive types are defined via their constructors. Coinductive types on the other
 150 hand are defined via their destructors. In the paper [APTS13] functions, which have
 151 coinductive types as their output, are implemented via copattern matching. In this
 152 paper streams are defined like the following.

```
153 record Stream A = { head : A,  
154                   tail : Stream A }
```

155 The **A** in the definition should be a concrete type. The type system in the paper
 156 don't has dependent types. What differentiate this from regular record types (for
 157 example in Haskell), is the recursive field **tail**. So they call it a recursive record. In
 158 a strict language without coinductive types we could never instantiate such a type,
 159 because to do this we already need something of type **Stream A** to fill in the field **tail**.
 160 To remedy this the paper defines copattern matching. With the help of copattern
 161 matching we can define functions which outputs expressions of type **Stream A**. As an
 162 example we look at the definition of **repeat**. This function takes in a value of type
 163 **Nat** and generates a stream which just infinitely repeats it.

```
164 repeat : Nat → Stream Nat  
165 head (repeat x) = x  
166 tail (repeat x) = repeat x
```

167 As you can see copattern matching works via observations i.e. we define what should
 168 be the output of the fields applied to the result of the function. These fields are
 169 also called observers, because we observe parts of the type. Because inhabitants
 170 of **Stream** are infinitely long we can't print out a stream. Because of this we also
 171 consider each expression with has a type, which is coinductive, as a value. To get
 172 a subpart of this value we have to use observers. For example we can look at the
 173 third value of **repeat 2** via **head (tail (tail (repeat 2)))** which should evaluate to
 174 **2**. We can also implement a function which looks at the *n*th. value. Here it is.

```
175 nth : Nat → Stream A → A  
176 nth 0 x = head x  
177 nth (S n) x = nth n (tail x)
```

178 As you can see we use ordinary pattern matching on the left hand side and observers
 179 on the right hand side. **nth 3 (repeat 2)** will output **2** as expected. Functions can also
 180 be defined via a recursive record. It is defined like the following.

```
181 record A → B = { apply : A → B }
```

182 Here we differentiate between our defined function $\mathbf{A} \rightarrow \mathbf{B}$ and \leadsto in the destructor.
 183 Constructor application or, as is the case here, destructor application is not the same
 184 as function application, like in Haskell. In the paper $\mathbf{f} \mathbf{x}$ means `apply f x`. We will
 185 also use this convention in the following. In fact we already used it in the definitions
 186 of the functions `repeat` and `nth`. `nth 0 x = head x` is just a nested copattern. We
 187 can also write it with ‘`apply`’ like so: `apply (apply nth 0) x = head x`. Here we use
 188 currying. So the first `apply` is the sole observer of type $\mathbf{Stream} \mathbf{A} \rightarrow \mathbf{A}$ and the second
 189 of type $\mathbf{Nat} \rightarrow (\mathbf{Stream} \mathbf{A} \rightarrow \mathbf{A})$.

3. Coinductive Types in dependent languages

In this section we will look how coinductive types are implemented in dependently typed languages. In dependently typed languages types can depend on values. The classical example for such a type is the vector. Vectors are like list, except their length is contained in their type. For example a vector of natural numbers of length 2 has type `Vec Nat 2`. This type depends on two things. Namely the type `Nat` and the value 2, which is itself of type `Nat`. We can define vectors in coq like follows.

```
Inductive Vec (A : Set) : nat -> Set :=
| Nil : Vec A 0
| Cons : forall {k : nat}, A -> Vec A k -> Vec A (S k).
```

Contrary to a list the type constructor `Vec` has a second argument `nat`. This is the already mentioned length of the vector. A Vector has two constructors. One for the empty vector called `Nil` and one to append a element at the front of a vector called `Cons`. `Nil` just returns a vector of length 0. And `Cons` gets an `A` and a vector of length `k`. It returns a vector of length `S k` (`S` is just the successor of `k`). This type can also be defined in agda like follows.

```
data Vec (A : Set) : ℕ → Set where
  Nil : Vec A 0
  Cons : {k : ℕ} → A → Vec A k → Vec A (suc k)
```

One advantage of vectors over list is that we can define a total function (a function which is defined for every input) which takes the head of a vector. This function can't be total for lists, because we can't know if the input list is empty. an empty list has no head. For vectors we can enforce this in coq like follow.

```
Definition hd {A : Set} {k : nat} (v : Vec A (S k)) : A :=
  match v with
  | Cons _ x _ => x
end.
```

We just pattern match on `v`. The only patter is for the `Cons` constructor. The `Nil` constructor is a vector of length 0. But `v` has type `Vec A (S k)`. So it can't be a vector of length 0. In agda the function looks like follow.

```
hd : {A : Set} {k : ℕ} → Vec A (suc k) → A
hd (cons x _) = x
```

That terms can occur in types makes it necessary to ensure that function terminate. Otherwise type checking wouldn't be decidable. If we have a function `f : Nat → Nat` and we want to check a value `a` against a type `Vec (f 1)` we have to

214 know what `f 1` evaluates to. So `f` has to terminate. We check termination in `coq`
 215 via a structural decreasing argument. An argument is structural decreasing, if it
 216 is structural smaller in a recursive call. Structural smaller means it is a recursive
 217 occurrence in a constructor. As an example we look at the definition of the natural
 218 numbers and the function for addition on them. We define the natural numbers in
 219 `coq` like follows.

```
Inductive nat : Set :=
| 0 : nat
| S : nat -> nat.
```

220 `0` is the constructor for 0 and `S` is the successor of its argument. Here the recursive
 221 argument to `S` is structural smaller than `S` applied to it i.e. `n` is structural smaller
 222 than `S n`. Then we can define addition like follows.

```
Fixpoint add (n m : nat) : nat :=
match n with
| 0 => m
| S p => S (add p m)
end.
```

223 In the recursive call the first argument is structural decreasing. `p` is smaller than
 224 `S p`. So `coq` accepts this definition. The classical example for a function where an
 225 argument is decreasing, but not structural decreasing is quicksort. A naive imple-
 226 mentation would be the following.

```
Fixpoint quicksort (l : list nat) : list nat :=
match l with
| nil => nil
| cons x xs => match split x xs with
| (lower, upper) => app (quicksort lower) (cons x (quicksort upper))
end
end.
```

227 Here `split` is just a function which gets a number and a list of numbers. It gives back
 228 a pair of two lists where the left list are all elements of the input list which are smaller
 229 than the input number and the right these which are bigger. It is clear that these
 230 lists can't be longer than the input list. So `lower` and `upper` can't be longer than
 231 `xs`. Here `xs` is structural smaller than the input `cons x xs`. So `lower` and `upper` are
 232 smaller than the input. Therefore we know that `quicksort` is terminating. But `coq`
 233 won't accept our code, because no argument is structural decreasing.

234 For coinductive types termination means that functions which produce them should
 235 be productive. If a function is productive it produces in each step a new part of the
 236 infinitely large coinductive type.

237 In section 3.1 we will look at the implementation in `coq`. There are two ways to
 238 define them. The older way uses positive coinductive types. This is known to violate
 239 subject reduction. Therefore it is highly discouraged to use them. To fix this the new
 240 way uses negative coinductive types. In section 3.2 we look at the implementation in
 241 `agda`. `Agda` also has the two ways of defining such types. One special thing about it,
 242 is that it implements copattern matching. To help `agda` with termination checking

we can use sized types. We will explain them in section 3.2.3.

3.1. Coinductive Types in Coq

There are two approaches to define coinductive types in coq. The older one is described in 3.1.1. It works over constructors. Therefore they are called positive coinductive types. The newer and recommended one is described in section 3.1.2. They are defined over primitive records (a relatively new feature of coq). Therefore they are called negative coinductive Types.

3.1.1. Postive Coinductive Types

Positive coinductive types are defined over constructors in coq. The keyword **CoInductive** is used to indicate that we about to define a coinductive type. This is the only syntactical difference from the definition of inductive types. For example streams are defined like the following.

```
CoInductive Stream (A : Set) : Set :=
  Cons : A -> Stream A -> Stream A.
```

If this was an inductive type we couldn't generate a value of this type. To generate values of coinductive types coq uses guarded recursion. This checks if the recursive call to the function occurs as an argument to a coinductive constructor. In addition to the guard condition the constructor can only be nested in other constructors, fun or match expressions. With all of this in mind we can define **repeat** like the following.

```
CoFixpoint repeat (A : Set) (x : A) : Stream A := Cons A x (repeat A x).
```

Then we can produce the constant zero stream with **repeat nat 0**. If we used a normal coq function i.e. write **Fixpoint** instead of **CoFixpoint** coq wouldn't except our code. It rejects it, because there is no argument which is structural decreasing. **x** stays always the same. **CoFixpoint** on the other hand only checks the previously mentioned conditions. It sees the recursive call **repeat A x** occurs as an argument to constructor **Cons** of the coinductive type **Stream**. This constructor is also not nested. So our definition is accepted.

We can use the normal pattern matching of coq to destruct a coinductive type. We define **nth** like the following.

```
Fixpoint nth (A : Set) (n : nat) (s : Stream A) {struct n} : A :=
  match s with
    Cons _ a s' =>
      match n with 0 => a | S p => nth A p s' end
  end.
```

The guard condition is necessary to ensure every expression is terminating. If we didn't have the guard condition we could define the following.

CoFixpoint loop (A : Set) : Stream A = loop A.

272 Here the recursive call doesn't occur in a constructor. So the guard condition is vio-
 273 lated. With this definition the expression **nth 0 loop** wouldn't terminate. **nth** would
 274 try to pattern match on **loop**. But to succeed in that **loop** has to unfold to some-
 275 thing of the form **Cons a ?** which it never does. So **nth 0 loop** will never evaluate to
 276 a value. This would lead to undecidable type checking.

277 We illustrate the purpose of the other conditions on an example taken from [Ch13].
 278 First we implement the function **tl** like so.

Definition tl A (s : Stream A) : Stream A :=
 match s with
 | Cons _ s' => s'
 end.

279 This is just one normal pattern match on **Stream**. If we didn't had the other condition
 280 we could define the following.

CoFixpoint bad : Stream nat := tl nat (Cons nat 0 bad).

281 This doesn't violate the guard condition. The recursive call **bad** is a argument to
 282 the constructor **Cons**. But the constructor is nested in a function. If we would allow
 283 this, **nth 0 bad** would loop forever. To understand why, we first unfold **tl** in **bad**. So
 284 we get

nth 0 (cofix bad : Stream nat :=
 match (Cons 0 bad) with
 | Cons _ s' => s'
 end)

285 We can now simplify this to just

nth 0 (cofix bad : Stream nat := bad)

286 After that **bad** isn't anymore an argument to a constructor. Here we can also see
 287 easily that the expression **cofix bad : Stream nat := bad** loops for ever. So we never
 288 get the value at position 0.

289 An important property of typed languages is subject reduction. Subject reduction
 290 says if we evaluate an expression e_1 of type t to an expression e_2 , e_2 should also be
 291 of type t . With positive coinductive types subject reduction is no longer valid. We
 292 illustrate this by Oury's counterexample [Our08]. First we define the codata type
 293 **U** as follows

CoInductive U : Set := In : U -> U.

294 We can now define a value of **u** with the following **Cofixpoint** like so

CoFixpoint u : U := In u.

295 This generates an infinite succession of **In**. We use the function **force** to force **U** to
 296 evaluate one step i.e. **x** becomes **In y**.

Definition force (x: U) : U :=
 match x with

```

    In y => In y
end.

```

297 The same trick will be used to define **eq** which states that **x** is definitional equal to
 298 **force x**.

```

Definition eq (x : U) : x = force x :=
  match x with
    In y => eq_refl
  end.

```

299 This first matches on **x** to force it, to reduce to **In y**. Then the new goal becomes
 300 **In y = force (In y)**. **force (In y)** evaluates to just **In y**, as it is just pattern match-
 301 ing on **In y**. So the final goal is **In y = In y** which can be shown by **eq_refl**.
 302 **eq_refl** is a constructor for **=**, where both sides of **=** are exactly the same. If we
 303 now instantiate **eq** with **u** we become **eq u**.

```

Definition eq_u : u = In u := eq u

```

304 But **u** is not definitional equal to **In u**. As mentioned above expression with a coin-
 305 ductive type are always values to prevent infinite evaluation. So **In u** is a value and
 306 **u** is also a value. But values are only definitional equal, if they are exactly the same.
 307 The next section will solve this problem through negative coinductive types.

308 3.1.2. Negative Coinductive Types

309 In coq 8.5. primitive records were introduced. With this it is now possible to define
 310 types over there destructors. So we can have negative , especially negative coinduc-
 311 tive, types in coq. With primitive records we can define streams like the following.

```

CoInductive Stream (A : Set) : Set :=
  Seq { hd : A; tl : Stream A }.

```

312 Now we can define **repeat** over the fields of **Stream**.

```

CoFixpoint repeat (A : Set) (x : A) : Stream A :=
  {| hd := x; tl := repeat A x|}.

```

313 To define **repeat** we must define what is the head of the constructed stream and
 314 what it is tail. The guard condition says now that corecursive occurrences must be
 315 guarded by a record field. We can see that the corecursive call **repeat** is a direct
 316 argument to the field **tl** of the corecursive type **Stream A**. This means coq accepts
 317 the above definition. If we want to access parts of a stream we use the destructors
 318 **hd** and **tl**. With them we can define **nth** again for the negative stream.

```

Fixpoint nth (A : Set) (n : nat) (s : Stream A) : list A :=
  match n with
    | 0 => s.(hd A)
    | S n' => nth A n' s.(tl A)
  end.

```

319 With negative coinductive types we can't form the above mentioned counterexample
 320 to subject reduction anymore, because we can't pattern match on negative types.

321 Oury's example becomes.

```
CoInductive U := { out : U }.
```

322 U is now defined over its destructor **out**, instead of its constructor **in**. Then **in**
323 becomes just a function. In Fact its just a definition, because we don't recurse or
324 corecure on it.

```
Definition In (y : U) : U := { | out := y | }.
```

325 We define it over the only field **out**. When we put a **y** in then we get the same **y** out.
326 We can also again define **u**.

```
CoFixpoint u : U := { | out := u | }.
```

327 With coinductive types it is know possible to define the pi type (the depend funcion
328 type).

```
CoInductive Pi (A : Set) (B : A -> Set) := { Apply (x : A) : B x }.
```

329 The pi type is defined over its destructor **Apply**. If we evaluate **Apply** on a value of
330 **Pi** (which is a function) and an argument, we get the result i.e. we apply the value to
331 the function. It looks like the pi type becomes definable in coq. But we are cheating.
332 The type of **Apply** is already a pi type. This is because we identify constructors
333 and destructors with functions. We will see that the theory of the paper avoids this
334 identification. To define a function we use **Cofixpoint**. As a simple non recursive,
335 non dependent example we use the function **plus2**.

```
Cofixpoint plus2 : Pi nat (fun _ => nat) :=  
  { | Apply x := S (S x) | }.
```

336 If we apply (i.e. call the destructor **Apply**) a **x** to **plus2** it gives back **S (S x)**. Which
337 is twice the successor on **x**. So we add 2 to **x**. We use **_** here because **plus2** is not
338 a dependent function i.e. the result type **nat** doesn't depend on the input value. To
339 define functions with more than one argument we just use currying i.e. we use the
340 type **Pi** as the second argument to **Pi**. For example a 2-ary non-dependent function
341 from **A** and **B** to **C** would have type **Pi A (fun _ => Pi B (fun _ => C))**. It would be
342 fortunate if we could define **plus** like the following.

```
Cofixpoint plus : Pi nat (fun _ => Pi nat (fun _ => nat)) :=  
  { | Apply := fun (n : nat) =>  
    match n with  
    | 0 => { | Apply (m : nat) := m | }  
    | S n' => { | Apply m := S (Apply _ _ (Apply _ _ plus n') m) | }  
    end  
  | }.
```

343 But coq doesn't accept this definition. The guard condition is violated. **plus n'** is
344 not a direct argument of the field **Apply**. The definition should terminate because
345 we are decreasing **n** and the case for **0** is accepted. In the case for **0**, there is no
346 recursive call.

347 We can also define a dependent function. We define **append2Units** like follows

```

CoFixpoint append2Units : Pi nat
  (fun n => Pi (Vec unit n)
    (fun _ => Vec unit (S (S n)))) :=
  {| Apply n := {| Apply v := Cons _ tt (Cons _ tt v) |} |}.

```

348 This just appends 2 units at a vector of length n . Here the second argument and the
 349 result depend on the first argument i.e. the first argument is the length of the input
 350 vector and the output vector is this length plus two.

351 3.2. Coinductive Types in Agda

352 In agda coinductive types were first also introduced as positive types. In the section
 353 3.2.1 we will look at them in detail. In section 3.2.2 we describe the correct way to
 354 implement coinductive types in agda. There are functions which terminate but are
 355 rejected by the type checker. In fact in any total language there have to be such
 356 functions. We can show that by trying to list all total functions. The following table
 357 lists functions per row. The columns say what the output of the functions for the
 358 given input is.

	1	2	3	4	...
f_1	2	7	8	6	...
f_2	4	4	6	19	...
359 f_3	6	257	1	2	...
f_4	7	121	23188	2313	...
\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

360 We can now define a function $g(n) = f_n(n) + 1$ this function is total and not in the list,
 361 because it is different to any function in the list for at least one input. To allow more
 362 functions we can use an unique feature of agda, sized types. They are described in
 363 section 3.2.3.

364 3.2.1. Positive Coinductive Types in Agda

365 Agda doesn't have a special keyword to define coinductive types like `coq`. It uses
 366 the symbol ∞ to mark arguments to constructors as coinductive. This symbol says
 367 that the computation of arguments of this type are suspended. ∞ is just a type
 368 constructor. So agda ensures productivity over type checking. We define streams
 369 like so.

```

data Stream (A : Set) : Set where
  cons : A → ∞ (Stream A) → Stream A

```

370 Here the second argument to `cons` is marked with ∞ . This is the tail of the stream.
 371 Because it is infinitely long (we don't have a constructor of an empty stream) we can't

372 compute it completely, so we suspend the computation. We can delay a computation
 373 with the constructor \sharp and force it with the function \flat . Their types are given below.

```
 $\sharp$  _ :  $\forall \{a\} \{A : \mathbf{Set} \ a\} \rightarrow A \rightarrow \infty \ A$   

 $\flat$  _ :  $\forall \{a\} \{A : \mathbf{Set} \ a\} \rightarrow \infty \ A \rightarrow A$ 
```

374 We can now again define our usual functions. We begin with **repeat**.

```
repeat :  $\{A : \mathbf{Set}\} \rightarrow A \rightarrow \mathbf{Stream} \ A$   

repeat x = cons x ( $\sharp$  (repeat x))
```

375 We first apply **cons** to **x**. So the head of the stream is **x**. We then apply it to the
 376 corecursive call **repeat**. So the tail will be a repetition of **xs**. We have to call the
 377 **repeat** with \sharp to suspend the computation. Otherwise the code doesn't type check.
 378 If we would write this function without \sharp on a stream which has no ∞ on the second
 379 argument of **cons**, the function would run forever. In fact the termination checker
 380 won't allow us to write such a function. We can also write **nth** again, which consumes
 381 a stream.

```
nth :  $\{A : \mathbf{Set}\} \rightarrow \mathbb{N} \rightarrow \mathbf{Stream} \ A \rightarrow A$   

nth 0 (cons x _) = x  

nth (suc n) (cons _ xs) = nth n ( $\flat$  xs)
```

382 Here we have to use \flat on the right hand side of the second case, to force the computa-
 383 tion of the tail of the input stream. We have to do that because **nth** wants a stream.
 384 It doesn't want a suspended stream. Productivity on coinductive types like stream
 385 is checked by only allowing non decreasing recursive calls behind the \sharp constructor.

386 3.2.2. Negative Coinductive Types in Agda

387 In agda we can also define negative coinductive types. This is the recommended
 388 way. Agda implements the previously mentioned copattern matching. We can define
 389 a record with the keyword **record**. We use the keyword **coinductive** to make it
 390 possible to define recursive fields. Stream is defined like the following.

```
record Stream (A : Set) : Set where  

  coinductive  

  field  

  hd : A  

  tl : Stream A
```

391 A Stream has 2 fields. **hd** is the head of the stream. It has type **A**. **tl** is the tail
 392 of the stream. It is another stream, so it has type **Stream A**. **tl** is a recursive field.
 393 So agda wouldn't accept the definition without **coinductive**. Stream can never be
 394 empty. Every stream has a head (a field **hd**) and an empty stream wouldn't have a
 395 head. So the tail of a stream can never be empty. Therefor every stream is infinitely
 396 long. We can now define **repeat** with copattern matching.

```
repeat :  $\forall \{A : \mathbf{Set}\} \rightarrow A \rightarrow \mathbf{Stream} \ A$   

hd (repeat x) = x  

tl (repeat x) = repeat x
```


We have to copattern match on every field of **Stream**, namely **hd** and **tl**. Because agda is total it won't accept non-exhaustive (co)pattern matches like Haskell. First we define what the head of **repeat x** is. We just repeat **x** infinitely often. So every element of the stream is **x**, including the head. Therefore we just write **x**. In the second and last copattern we define what the tail of the stream is. The tail is just **repeat x**. Infinitely often repeated **x** is the same as **x** and then infinitely repeated **x**. We can use normal pattern matchings and the destructors for functions which consume streams. We define **nth** like the following.

```
nth : ∀ {A : Set} → ℕ → Stream A → A
nth zero s = hd s
nth (suc n) s = nth n (tl s)
```

Here we just pattern match on the first argument (excluding the implicit argument of the type). If it is zero the result is just the head of the stream. If it is $n + 1$ the result is the recursive call of **nth** on **n** and **tl s**. Agda accepts this code, because it is structural decreasing on the first (or second if we count the implicit) argument.

We can also define the pi type. We use **__\$** as the apply operator. This operator is taken from Haskell.

```
record Pi (A : Set) (B : A → Set) : Set where
  field _$_ : (x : A) → B x
  infixl 20 _$_
  open Pi
```

Like in coq we are using the first-class pi type to define the pi type. We can also define a function which adds 2 to a number **plus2** in agda.

```
plus2 : ℕ → 'ℕ
plus2 $ x = suc (suc x)
```

We just use copattern matching to define it. If we apply a **x** to **plus2** we get **suc (suc x)**. **→'** is just the non-dependent function it is defined using our pi type. Here it is.

```
→' : Set → Set → Set
A →' B = Pi A (λ _ → B)
infixr 20 →'_
```

In agda it becomes possible to define plus. We just use nested copattern matching.

```
plus : ℕ → 'ℕ →' ℕ
plus $ 0 $ m = m
plus $ (suc n) $ m = suc (plus $ n $ m)
```

If we change **→'** to **→** and remove **\$** we get the standard definition for plus in agda.

We can also define a dependent function **repeatUnit** like follow

```
repeatUnit : Pi ℕ (λ n → Vec τ n)
repeatUnit $ 0 = nil
repeatUnit $ suc n = tt :: (repeatUnit $ n)
```

This function gives back a vector with the length of the input, where every element is unit.

421 3.2.3. Termination Checking with Sized Types

422 They are many functions, which are total but are not accepted by agda's termination
 423 checker. For example we could try to define division with rest on natural numbers
 424 like the following.

```

_/_ : ℕ → ℕ → ℕ
zero / y = zero
suc x / y = suc ( (x - y) / y)

```

425 The problem with this definition is that agda doesn't know that $x - y$ is smaller
 426 than $x + 1$, which is clearly the case (x and y are positive). This definition would
 427 work perfectly fine in a language without termination checking (like Haskell). Agda
 428 only checks if an argument is structurally decreasing. Here it is neither the case for
 429 x nor for y .

430 To remedy this problem sized types were introduced first to mini-agda (a language
 431 specifically developed to explore them) by [Abe10]. Later they got introduced to
 432 agda itself. Sized types allow us to annotate data with their size. Functions can use
 433 this sizes to check termination and productivity.

434 We can now define the natural numbers depending on a size argument.

```

data ℕ (i : Size) : Set where
  zero : ℕ i
  suc : ∀ {j : Size < i} → ℕ j → ℕ i

```

435 The natural number now depends on a size i . The constructor **zero** is of arbitrary
 436 size i . **suc** gets a size j which is smaller than i , a natural number of size j and gives
 437 back a natural number of size i . This means the size of the input is smaller than the
 438 size of the output. For inductive types, a size is an upper bound on the number of
 439 constructors. With **suc** we add a constructor so the size has to increase i . We can
 440 now define subtraction on these sized nats.

```

_/_ : {i : Size} → ℕ i → ℕ ∞ → ℕ i
zero - _ = zero
n - zero = n
(suc n) - (suc m) = n - m

```

441 Through the sized annotations, we know now that the result isn't larger than the
 442 first input. ∞ means that the size isn't bound. If the first argument is zero the result
 443 is also zero, which has the same type. If the second argument is zero we return
 444 just the first. In the last case both arguments are non-zero. We call subtraction
 445 recursively on the predecessors of the inputs. Here the size and both arguments are
 446 smaller. So the function terminates. Though the type is smaller than i , the result type
 447 checks because sizes are upper bounds. We can now define division.

```

_/_ : {i : Size} → ℕ i → ℕ ∞ → ℕ i
zero / _ = zero
suc x / y = suc ( (x - y) / y)

```

448 From the definition of **suc** we know that the size of x is smaller than i . Because the

449 result of $-$ has the same size as its first input (here x), we also know that $(x - y)$ has
 450 the same size as x . Therefore $(x - y)$ is smaller than $\text{succ } x$ and the function is decreasing
 451 on the first argument. Also, agda accepts this definition.

452 We can also use sized types for coinductive types. To show this we will define the
 453 hamming function. This produces a stream of all composites of two and three in
 454 order. First we will define the sized stream type.

```
record Stream (i : Size) (A : Set) : Set where
  coinductive
  field
    hd : A
    tl :  $\forall \{j : \text{Size} < i\} \rightarrow \text{Stream } j \text{ } A$ 
  open Stream
```

455 This stream has a new parameter of type **Size**. This size gives the minimal definition
 456 depth of the stream. The definition depth says how often we can destruct the stream
 457 without diverging. If we take the tail of an stream, the output streams depth would
 458 be one smaller. Because in agda coinductive types can't have indexes, we can only say
 459 that its depth is smaller. We will now define some helper functions for the hamming
 460 function. First we need a cons function.

```
cons : {i : Size} {A : Set}  $\rightarrow A \rightarrow \text{Stream } i \text{ } A \rightarrow \text{Stream } i \text{ } A$ 
hd (cons x _) = x
tl (cons _ xs) = xs
```

461 This just appends an element at the front of the stream. Because the output streams
 462 depth is larger than the input and the size is a minimum, we can give the output
 463 the same size parameter as the input. Now we will define map over stream.

```
map : {A B : Set} {i : Size}  $\rightarrow (A \rightarrow B) \rightarrow \text{Stream } i \text{ } A \rightarrow \text{Stream } i \text{ } B$ 
hd (map f xs) = f (hd xs)
tl (map f xs) = map f (tl xs)
```

464 This function just changes the content of the stream so the size stays the same. The
 465 last helper function we need is the merge function.

```
merge : {i : Size}  $\rightarrow \text{Stream } i \text{ } \mathbb{N} \rightarrow \text{Stream } i \text{ } \mathbb{N} \rightarrow \text{Stream } i \text{ } \mathbb{N}$ 
hd (merge xs ys) = hd xs  $\sqcap$  hd ys
tl (merge xs ys) = if [ hd xs  $\leq?$  hd ys ]
  then cons (hd ys) (merge (tl xs) (tl ys))
  else cons (hd xs) (merge (tl xs) (tl ys))
```

466 This functions just merges two streams. It always compares one element of each
 467 stream with each other and puts the bigger after the smaller. This is clear in the
 468 case for **hd** (\sqcap is just the binary minimum function in agda). in the **tl** case we just
 469 compare the heads of the stream and construct the tail with **cons** accordingly. Both
 470 input streams have a minimal definition depth of **i**. Because **cons** isn't destructing
 471 the stream (the minimal depth doesn't get smaller) we can say that the minimal
 472 depth of the output also won't get smaller. With all this function we can now define
 473 the ham function. Here it is.

```
ham : {i : Size}  $\rightarrow \text{Stream } i \text{ } \mathbb{N}$ 
hd ham = 1
tl ham = (merge (map (*_ 2) ham) (map (*_ 3) ham))
```

⁴⁷⁴ None of the used function is destructing the stream, so this definition gets accepted.

4. Type Theory based on dependent Inductive and Coinductive Types

In the paper [BG16] a type theory, where inductive types and coinductive types can depend on values, is developed. For example we can, in contrast to the coinductive types of coq and agda, define streams which depend on their definition length. The theory differentiates types from terms. We don't have infinite universes, where a term in universe n has a type in universe $n+1$ (This is how it is done in coq [ST14] and agda [agd]). Therefore types can only depend on values, not on other types. We only have functions on the type level. These functions abstract over terms. For example $\lambda x.A$ is a type where all occurrences of the term variable x in A are bound. We will see that functions are definable on the term level. We can apply types to terms. For example $A@t$ means we apply the term A to x . Every type has a kind. A kind is either $*$ or $\Gamma \rightarrow *$. Here Γ is a context, which states to what terms we can apply the type. For example we can apply A of kind $(x : B) \rightarrow *$ only to a term of type B . If we apply it to t of type B , we get a type of kind $*$. We write \rightarrow instead of \rightarrow to indicate, that these are not functions. We can also apply a term to another term. For example $t@s$ means we apply the term t to the term s . Terms also can depend on contexts. For example if we have a term t of type $(x : A) \rightarrow B$ and apply it to a term s of type A we get a term of type B . We can also define our own types. $\mu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})$ is an inductive type and $\nu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})$ is a coinductive type. X is a variable which stands for the recursive occurrence of the type. It has the same kind $\Gamma \rightarrow *$ as the defined type. The \vec{A} can contain this variable. There are also contexts $\vec{\Gamma}$, which are implicit in the paper. σ_k and A_k can contain variables from Γ_k . σ_k is a context morphism from Γ_k to Γ . A context morphism is a sequence of terms, which depend on Γ_k and instantiate Γ . $\vec{\sigma}$, \vec{A} and $\vec{\Gamma}$ are of the same length.

In this theory we can define partial streams on some type A like the following.

$$\begin{aligned} \text{PStr } A &:= \nu(X : (n : \text{Conat}) \rightarrow *; (\text{succ}@n, \text{succ}@n); (A, X@n)) \\ &\text{with } \Gamma_1 = (n : \text{Conat}) \text{ and } \Gamma_2 = (n : \text{Conat}) \end{aligned}$$

Here **succ** is the successor on conats. Conats are natural numbers with one additional element, infinity. See 6.2 for their definition. Here the first destructor is the head. It becomes a stream with length $\text{succ}@N$ and returns an A . The second destructor is the tail. It becomes also a stream of length $\text{succ}@N$. It gives back an $X@n$, which is a stream of length n . We can also define the Pi type from A to B , where B can

depend on A .

$$\begin{aligned} \Pi x : A. B &:= \nu(_ : *; \epsilon_1; B) \\ \text{with } \Gamma_1 &= (x : A) \end{aligned}$$

500 By $_$ we mean, we are ignoring this variable. ϵ_1 is one empty context morphism.
 501 So the only destructor gives back a B which can depend on x of type A . It is the
 502 function application.

503 To construct an inductive types we use constructors (written $\alpha_k^{\mu(X:\Gamma \rightarrow *; \vec{\sigma}; \vec{A})}$ in the pa-
 504 per, which is the k 'st constructor of the given type). We can destruct it with recursion
 505 (written $\text{rec } \overrightarrow{(\Gamma_k \cdot y_k) \cdot g_k}$). Coinductive type work the other way around. We destruct
 506 them with destructors (written $\xi_k^{\nu(X:\Gamma \rightarrow *; \vec{\sigma}; \vec{A})}$) and construct them with corecursion
 507 (written $\text{corec } \overrightarrow{(\Gamma_k \cdot y_k) \cdot g_k}$).

508 We will give the rules for the theory in section 5.3 and a detailed explanation of the
 509 reduction in 5.4.

510 **5. Implementation**

511 In this section we look at the implementation details. We use the functional pro-
512 gramming language Haskell for implementing the theory. Haskell is a pure language.
513 This means functions which aren't in the IO monad have no side effects. The only
514 IO we are doing is reading a file and as the last step printing it. Because everything
515 between is pure, we can test it without bordering on side effects. Another feature
516 of Haskell, which will be get useful in our implementation is pattern matching. We
517 will see its usefulness in section 5.3.

518 In section 5.1 we will develop the abstract syntax of our language from the raw
519 syntax in the paper. Then we rewrite the typing rules in 5.3. At last we look at the
520 implementation of the reduction in 5.4

521 **5.1. Abstract Syntax**

522 In the following we will scratch out the abstract syntax. In contrast to [BG16] we
523 can't write anonymous inductive and coinductive types. We will give every inductive
524 and coinductive type a name. They will be defined via declarations. In these declara-
525 tions we will give, their constructors/destructors. They will also be given names. In
526 [BG16] they are anonymous. We can then refer to the previously defined types. We
527 will described declarations in section 5.1.1. We will also be able to bind expressions
528 to names. In section 5.1.2 we will define the syntax of expressions. This will mostly
529 be in one to one correspondence with the syntax of [BG16]. Note however that we
530 use the names of the constructors instead of anonymous constructors together with
531 their type and number. Also the order of the matches in rec and corec is irrelevant.
532 We use the names of the Con/Destructors to identify them. In the following section
533 6 we will see how the examples from the paper look in our concrete syntax.

534 **5.1.1. Declarations**

535 The abstract syntax is given in figure 5.1. With the keywords data and codata
536 we define inductive and coinductive types respectively. After that we will write
537 the name. We can only use names which aren't used already. Behind that we can
538 give a parameter context. This is a type context. These types are not polymorphic.
539 They are merely macros to make the code more readable and allow the definition of

N	$:= [A - Z][a - zA - Z0 - 9]^*$	Names for types, constructors and destructors
n	$:= [a - z][a - zA - Z0 - 9]^*$	Names for expressions
EV	$:= x, y, z, \dots$	Expression variables
TV	$:= X, Y, Z, \dots$	Type expression variables
PV	$:= A, B, C, \dots$	Parameter variables
EC	$:= \diamond$ $ (EV : TV, EV : TV)^*$	Expression Context
PC	$:= \langle \rangle$ $ \langle (PV : EC \rightarrow \text{Set})^* \rangle$	Parameter Context
$Decl$	$:= \text{data } N \text{ } PC : (EC \rightarrow)? \text{ Set where}$ $(N : (EC \rightarrow)? \text{TypeExpr} \rightarrow N \text{Expr})^*$ $ \text{codata } N \text{ } PC : (EC \rightarrow)? \text{ Set where}$ $(N : (EC \rightarrow)? N \text{Expr}^* \rightarrow \text{TypeExpr})^*$ $ n \text{ } PC \text{ } EC = \text{Expr}$	Declarations

Figure 5.1.: Syntax for declarations

540 nested types. If we want to use these types we have to fully instantiate this context.
 541 These types can occur everywhere in the definition where a type is expected. A
 542 (co)inductive type can have a context, which is written before an arrow. **Set** stands
 543 for type (or $*$ in the paper). If a type don't has a context we omit the arrow. We
 544 will also give names to every constructor and destructor. These names have to be
 545 unique. Constructors and destructors also have contexts. Additionally they have
 546 one argument which can has a recursive occurrence of the type we are defining. A
 547 constructor gives back a value of the type, where its context is instantiated. This
 548 instantiation corresponds to the sigmas in the paper. If we write a name before an
 549 equal sign we can bind the following expression to the name. Every such defined
 550 name can depend on a parameter context and an argument context. We write the
 551 parameter context like in the case for data types behind the name. After that we
 552 can give a term context between round parenthesis.

553 The declarations in Figure 5.1 correspond to $\rho(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A}) : \Gamma \rightarrow *$ as follows.

- 554 • The first N is X
- 555 • The other N will be used later for $\alpha_1^{\mu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})}, \alpha_2^{\mu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})}, \dots$ in the case of
 556 inductive types and $\xi_1^{\nu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})}, \xi_2^{\nu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})}, \dots$ in the coinductive case
- 557 • The TypeExpr are the \vec{A}

- The $Expr^*$ are the $\vec{\sigma}$
- The first EC is Γ
- The other EC stand for $\Gamma_1, \dots, \Gamma_m$

To parse the abstract syntax we use megaparsec. The parser generates an abstract syntax tree, which is given for declarations in Listing 1. The field **ty** in **ExprDef** is used later in type checking. The parser just fills them in with **Nothing**. data and co-data definitions are both saved in **TypeDef**. The Haskell type **OpenDuctive** contains all the information for inductive and coinductive types. It corresponds to μ and ν in the paper. We use an **OpenDuctive** where the field **inOrCoin** is **IsIn** for μ and an **OpenDuctive** where the field **inOrCoin** is **IsCoin** for ν . The Haskell type **StrDef** ensures that the sigmas, as and gammals have the same length. We omit the implementation details for the parser, because we are manly focused on type checking.

```

data Decl = ExprDef { name :: Text
                     , tyParameterCtx :: TyCtx
                     , exprParameterCtx :: Ctx
                     , expr :: Expr
                     , ty :: Maybe Type
                     }
  | TypeDef OpenDuctive
  | Expression Expr

data OpenDuctive = OpenDuctive { nameDuc :: Text
                                , inOrCoin :: InOrCoin
                                , parameterCtx :: TyCtx
                                , gamma :: Ctx
                                , strDefs :: [StrDef]
                                }

data StrDef = StrDef { sigma :: [Expr]
                     , a :: TypeExpr
                     , gammal :: Ctx
                     , strName :: Text
                     }

```

Listing 1: Implementation of the abstract syntax of fig. 5.1

5.1.2. Expressions

The abstract syntax for expression is given in figure 5.2. We will separate expression in expressions for terms and expressions for types. There are given as regular expressions in **Expr** and **TypeExpr** respectively.

An **Expr** is either a **rec**, a **corec**, a con/destructor, an application @, the only primitive unit expression \diamond or a variable. With the keyword **rec** we can destruct an inductive type. We write **N ParInst?** to **TypeExpr**, where **N** is a previously defined inductive type and **ParInst?** the instantiation of its parameter context, after **rec** to facilitate type checking. It says we want to destruct an inductive type to some other

$ParInst$	$:= \langle TypeExpr, TypeExpr \rangle^*$	Instantiations for parameter contexts
$ExprInst$	$:= (Expr, Expr)^*$	Instantiations for expression contexts
$Expr$	$:= \text{rec } N \text{ } ParInst? \text{ to } TypeExpr \text{ where}$ $\quad Match^*$ $ \text{ corec } TypeExpr \text{ to } N \text{ } ParInst? \text{ where}$ $\quad Match^*$ $ Expr @ Expr$ $ \Diamond$ $ EV$ $ n \text{ } ParInst \text{ } ExprInst$	expression
$Match$	$:= N \text{ } EV^* = Expr$	match
$TypeExpr$	$:= (EV : TypeExpr).TypeExpr$ $ TypeExpr @ Expr$ $ Unit$ $ TV$ $ N \text{ } ParInst?$	Type expressions

Figure 5.2.: Syntax for expressions

type . We have to list all the constructors above one another. For each constructor we write an expression behind the equal sign, which should be of type **TypeExpr** which we have given above. In this expression we can use variables given in the match expression. The last one is the recursive occurrence. With the keyword **corec** we can do the same thing to construct a coinductive type. Here we have to swap the **NParInst?** and the **TypeExpr** and list the destructors. All con/destructors have to be instantiate with all variables in the parameter contexts of their types. This is done by giving types of the expected kinds separated by ',' enclosed in \langle and \rangle . The variables are separated in local variables and global variables. Global variables refer to previously defined expressions. We have to fully instantiate they parameter contexts and their expression contexts. We can also apply an expression to another with **@**. This application is left associative. So if we write **t @ s @ v** we mean **(t @ s) @ v**.

The **typeExpr** is either the unit type **Unit**, a lambda abstraction on types, an application or a variable. In the lambda expression we have to give the type of the variable. We apply a type to a term (types can only depend on terms) with **@**. As in the case of term application this is also left associative. The unit type is the only primitive type expression.

The generated abstract syntax tree is given in listing 2. The variables for expressions are separated in **LocalExprVar** and **GlobalExprVar**. **LocalExprVar** should refer to variables which are only locally defined i.e. in **Rec** and **Corec**. We use de-Brujin indexes for them. This facilitates substitution which we will describe in section 5.2. **GlobalExprVar** refers to variables from definitions. Here we just use names. We do the same thing for **LocalTypeVar** and **GlobalTypeVar**. In the abstract syntax tree we use anonymous constructors like in the paper. We combine them to the Haskell constructor **Structor**. We know from the field **ductive** if it is a constructor or a destructor. The types in field **parameters** are to fill in the parameter context of the field **ductive**. The field **nameStr** in **Constructor** and **Destructor** are just for printing. We combine rec and corec to **Iter**.

5.2. Substitution

In the following we will write $t[s/x]$ for "substitute every free occurrences of x in t by s ". Substitution is done in the module **Subst.hs**. We use de-Brujin indexes [DB72] for bound variables to facilitate substitution. With this method every bound variable is a number instead of a string. The number says where the variable is bound. To find the binder of a variable we go outwards from it and count every binder until we reach the number of the variable. For example $\lambda.\lambda.\lambda.1$ says that the variable is bound by the second binder (we start counting at zero). This would be the same as $\lambda x.\lambda y.\lambda z.y$. This means we never have to generate fresh names. We just shift the free variables in the term with which we substitute by one, every time we encounter a binder. This shifting is done in the module **ShiftFreeVars.hs**. We

```

data TypeExpr = UnitType
  | TypeExpr :@ Expr
  | LocalTypeVar Int Bool Text
  | Parameter Int Bool Text
  | GlobalTypeVar Text [TypeExpr]
  | Abstr Text TypeExpr TypeExpr
  | Ductive { openDuctive :: OpenDuctive
             , parametersTyExpr :: [TypeExpr] }

data Expr = UnitExpr
  | LocalExprVar Int Bool Text
  | GlobalExprVar Text [TypeExpr] [Expr]
  | Expr :@: Expr
  | Structor { ductive :: OpenDuctive
              , parameters :: [TypeExpr]
              , num :: Int
              }
  | Iter { ductive :: OpenDuctive
         , parameters :: [TypeExpr]
         , motive :: TypeExpr
         , matches :: [(Text, Expr)]
         }

```

Listing 2: Implementation of the abstract syntax of fig. 5.2

619 also want to be able to substitute multiple variables simultaneously. If we would
 620 just substitute one term after another we could substitute into a previous term. For
 621 example the substitution $x[y/x][z/y]$ would yield z if we substitute sequential and y if
 622 we substitute simultaneously. To make simultaneous substitution possible every local
 623 variable has a boolean flag. If this flag is set to true substitution won't substitute
 624 for that variable. So for simultaneous substitution we just set this flag to true for all
 625 terms with which we want to substitute. Then we substitute with them. In the last
 626 step we just have to set the flags to false in the result. This setting (marking of the
 627 variables) is done in the module **Mark.hs**.

628 5.3. Typing rules

629 A typing rule says that some expression or declaration is of some type, given some
 630 premises. If we can for every declaration or expression form a tree of such rules
 631 with no open premises, our program type checks. We have to rewrite the typing
 632 rules of the paper, to get rules which are syntax directed. Syntax directed means
 633 we can infer from the syntax alone what we have to check next i. e. which rule with
 634 which premises we have to apply. In the paper there are rules containing variables
 635 in the premises where their type isn't in the conclusion. So if we want to type-check
 636 something which is the conclusion of such a rule we have no way of knowing what
 637 these variables are.

638 We don't need the weakening rules because we can lookup a variable in a context.
 639 So we ignore them in our implementation.

The order in **TyCtx** isn't relevant so we can use a map for it. In the code we use a list, because the names of the variables are the index of their type in the context. The order of **Ctx** is relevant because types of later variables can refer to former variables and application instantiate the first variable in **Ctx**. We add a new context for data types. We also need a context for the parameters. **Ctx** can contain variables from this context, but not from **TyCtx**.

We also rewrite the rules which are already syntax-directed to rules which work on our syntax. We will mark semantic differences in the rewritten rules gray. We use variables $\Phi, \Phi', \Phi_1, \Phi_2, \dots$ for parameter contexts, $\Theta, \Theta', \Theta_1, \Theta_2, \dots$ for type variable contexts and $\Gamma, \Gamma', \Gamma_1, \Gamma_2, \dots$ for term variable contexts. The judgements in our rules are of one of the following form.

- $\Phi | \Theta | \Gamma \vdash \Theta'$ - The type variable context Θ' is well formed in the combined context $\Phi | \Theta | \Gamma$.
- $\Phi | \Theta | \Gamma \vdash \Gamma'$ - The term variable context Γ' is well formed in the combined context $\Phi | \Theta | \Gamma$.
- $\Phi | \Theta | \Gamma \vdash \Phi'$ - The parameter variable context Φ' is well formed in the combined context $\Phi | \Theta | \Gamma$.
- $A \longrightarrow_T^* B$ - The type A fully evaluates to type B .
- $A \equiv_\beta B$ - The type A is computational equivalent to type B .
- $\Phi | \Theta | \Gamma \vdash A : \Gamma_2 \rightarrow *$ - The type A is well formed in the combined context $\Phi | \Theta | \Gamma$ and can be instantiated with arguments according to context Γ_2 .
- $\Phi | \Theta | \Gamma \vdash t : \Gamma_2 \rightarrow A$ - The term t is well formed in the combined context $\Phi | \Theta | \Gamma$ and can be instantiated with arguments according to context Γ_2 . After this instantiation it is of type A , where the arguments are substituted in A .
- $\Phi \vdash \sigma : \Gamma_1 \triangleright \Gamma_2$ - The context morphism σ is a well-formed substitution for Γ_2 with terms in context Γ_1 in parameter context Φ .

We will write \vdash for $\Phi | \Theta | \Gamma \vdash$ where Φ, Θ and Γ are arbitrary and aren't referred to by the right hand side.

In the module **TypeChecker** we will implement the following rules. It defines a monad **TI** which can throw errors and has a reader on the contexts in which we are type checking. To add something to a context we use the function **local**. This function gets a function to change the current content of the reader monad and executes a reader on this changed context in the current monad.

5.3.1. Context rules

The rules for valid contexts are already syntax directed so we take just them.

$$\frac{}{\vdash \emptyset \text{ TyCtx}} \quad \frac{\vdash \Theta \text{ TyCtx} \quad \vdash \Gamma \text{ Ctx}}{\vdash \Theta, X : \Gamma \rightarrow * \text{ TyCtx}}$$

$$\frac{}{\vdash \emptyset \text{ Ctx}} \quad \frac{|\emptyset| \Gamma \vdash A : *}{\vdash \Gamma, x : A \text{ Ctx}}$$

In the rules for valid contexts we ensure that the types in the context can not depend on **TyCtx**. Note however that they can depend on **ParCtx**. This ensures that only strictly positive types are possible.

We also need new rules for checking if a parameter context is valid.

$$\frac{}{\vdash \emptyset \text{ ParCtx}} \quad \frac{\vdash \Phi \text{ ParCtx} \quad \vdash \Gamma \text{ Ctx}}{\vdash \Phi, X : \Gamma \rightarrow * \text{ ParCtx}}$$

This are structural the same rule as this for **TyCtx**. The difference is that **ParCtx** and **TyCtx** are used differently in the other rules, as we have already seen in the rule for **Ctx**.

We use the notation $\Theta(X) \rightsquigarrow \Gamma \rightarrow *$ for looking up the type variable X in type context Θ yields type $\Gamma \rightarrow *$. We add 2 rules for looking up something in a type context. They are:

$$\frac{\vdash \Theta \text{ TyCtx} \quad \vdash \Gamma \text{ Ctx}}{\Theta, X : \Gamma \rightarrow *(X) \rightsquigarrow \Gamma \rightarrow *} \quad \frac{\vdash \Gamma_1 \text{ Ctx} \quad \Theta(X) \rightsquigarrow \Gamma_2 \rightarrow *}{\Theta, Y : \Gamma_1 \rightarrow *(X) \rightsquigarrow \Gamma_2 \rightarrow *}$$

Here Y and X are different variables.

The rules for looking up something in a parameter context are principally the same.

$$\frac{\vdash \Phi \text{ ParCtx} \quad \vdash \Gamma \text{ Ctx}}{\Phi, X : \Gamma \rightarrow *(X) \rightsquigarrow \Gamma \rightarrow *} \quad \frac{\vdash \Gamma_1 \text{ Ctx} \quad \Phi(X) \rightsquigarrow \Gamma_2 \rightarrow *}{\Phi, Y : \Gamma_1 \rightarrow *(X) \rightsquigarrow \Gamma_2 \rightarrow *}$$

Respectively the notation $\Gamma(x) \rightsquigarrow A$ means looking up the term variable x in term context Γ yields type A . The rules for term contexts are:

$$\frac{\vdash \Gamma \text{ Ctx} \quad \Gamma \vdash A : *}{\Gamma, x : A(x) \rightsquigarrow A} \quad \frac{\Gamma(x) \rightsquigarrow A \quad \Gamma \vdash B : *}{\Gamma, y : B(x) \rightsquigarrow A}$$

5.3.2. Beta-equivalence

Two types are beta equivalent if they evaluate to the same type. Because our language is deterministic this just means if we fully evaluate both of them they are alpha equivalent. Alpha equivalence means we can substitute some variables in both of them and get the same type. So we first need to define rules which say what full evaluation means. We write $A \longrightarrow_T^* B$ for evaluating A as long as it is possible yields B .

The rules are:

$$\frac{\neg \exists B : A \longrightarrow_T B}{A \longrightarrow_T^* A} \quad \frac{A \longrightarrow_T B \quad B \longrightarrow_T^* C}{A \longrightarrow_T^* C}$$

704 \longrightarrow_T is defined in section 5.4.

705 We can then introduce a new rule for beta-equivalence.

$$706 \frac{A \longrightarrow_T^* A' \quad B \longrightarrow_T^* B' \quad A' \equiv_\alpha B'}{A \equiv_\beta B}$$

707 This rule says if A evaluates to A' , B to B' and A' and B' are alpha equivalent, then
 708 A and B are beta equivalent. In the implementation \equiv_α is trivial, because we use *de*
 709 *Bruijn indices*.

710 We also add some rules to check if two contexts are the same.

$$711 \frac{}{\emptyset \equiv_\beta \emptyset} \quad \frac{\Gamma_1 \equiv_\beta \Gamma_2 \quad A \equiv_\beta B}{\Gamma_1, x : A \equiv_\beta \Gamma_2, y : B}$$

712 5.3.3. Unit type and expression introduction

713 The paper defines one rule for the unit type and one for the unit value. These are.

$$714 \frac{}{\vdash \top : *} \text{ (}\top\text{-I)} \quad \frac{}{\vdash \diamond : \top} \text{ (}\top\text{-I)}$$

715 The first rule says that the type \top has always an empty context. The second rule
 716 says its value \diamond is always of type \top . These rules get rewritten to.

$$717 \frac{}{\Phi \mid \Theta \mid \Gamma \vdash \text{Unit} : *} \text{ (Unit-I)} \quad \frac{}{\Phi \mid \Theta \mid \Gamma \vdash \diamond : \text{Unit}} \text{ (}\top\text{-I)}$$

718 We change the syntax " \top " to "Unit" and add the contexts Φ, Θ, Γ . We will do this for
 719 every rule which has empty contexts to subsume the weakening rules of the paper.
 720 The unit term always has the unit type as its type.

721 5.3.4. Variable lookup

722 We have three kinds of variables we can lookup. They are type variables, term vari-
 723 ables and parameters. The paper already has rules for the type and term variables.
 724 We need to rewrite them. We add a new rule for looking up a parameter.

725 The rule

$$726 \frac{\vdash \Theta \quad \text{TyCtx} \quad \vdash \Gamma \quad \text{Ctx}}{\Theta, X : \Gamma \rightarrow * \mid \emptyset \vdash X : \Gamma \rightarrow *} \text{ TyVar-I}$$

727 gets rewritten to

$$728 \frac{\Theta(X) \rightsquigarrow \Gamma \rightarrow * \quad \vdash \Gamma_1 \quad \text{Ctx}}{\Phi \mid \Theta \mid \Gamma_1 \vdash X : \Gamma \rightarrow *} \text{ TyVar-I}$$

729 The rule

$$730 \frac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A} \text{ (Proj)}$$

731 gets rewritten to

$$732 \quad \frac{\Gamma(x) \rightsquigarrow A}{\Phi \mid \Theta \mid \Gamma \vdash x : A} \text{ (Proj)}$$

733 The rule for looking something up in the parameter context is.

$$734 \quad \frac{\Phi(X) \rightsquigarrow \Gamma \rightarrow * \quad \vdash \Gamma_1 \text{ Ctx}}{\Phi \mid \Theta \mid \Gamma_1 \vdash X : \Gamma \rightarrow *} \text{ TyVar-I}$$

735 In the rule from the paper we can only infer the type or kind of the last variable
 736 in the context. In our rules we just look up the variable in the context. These rules
 737 can check the same thing if we take the weakening rules into account. With them
 738 we can just weaken the context until we get to the desired variable.

739 5.3.5. Type and expression instantiation

740 We can instantiate types and terms. The rule

$$741 \quad \frac{\Theta \mid \Gamma_1 \vdash A : (x : B, \Gamma_2) \rightarrow * \quad \Gamma_1 \vdash t : B}{\Theta \mid \Gamma_1 \vdash A@t : \Gamma_2[t/x] \rightarrow *} \text{ (Ty-Inst)}$$

742 for instantiating types gets rewritten to

$$743 \quad \frac{\Phi \mid \Theta \mid \Gamma_1 \vdash A : (x : B, \Gamma_2) \rightarrow * \quad \Phi \mid \Theta \mid \Gamma_1 \vdash t : B' \quad B \equiv_\beta B'}{\Phi \mid \Theta \mid \Gamma_1 \vdash A@t : \Gamma_2[t/x] \rightarrow *} \text{ (Ty-Inst)}$$

744 For this rule we have to check if t has the expected type for the first variable in the
 745 context of A . In our version we just infer the type for A and t . Then we check if the
 746 first variable in the context is beta-equal to the type of t . If that isn't the case type
 747 checking fails. Otherwise we just substitute in the remaining context.

748 We also have a rule to instantiate terms. This rule

$$749 \quad \frac{\Gamma_1 \vdash t : (x : A, \Gamma_2) \rightarrow B \quad \Gamma_1 \vdash s : A}{\Gamma_1 \vdash t@s : \Gamma_2[s/x] \rightarrow B[s/x]} \text{ (Inst)}$$

750 gets rewritten to

$$751 \quad \frac{\Phi \mid \Theta \mid \Gamma_1 \vdash t : (x : A, \Gamma_2) \rightarrow B \quad \Phi \mid \Theta \mid \Gamma_1 \vdash s : A' \quad A \equiv_\beta A'}{\Phi \mid \Theta \mid \Gamma_1 \vdash t@s : \Gamma_2[s/x] \rightarrow B[s/x]} \text{ (Inst)}$$

752 These rules are similar to the rule for type instantiation. Here we have to check(or
 753 infer) a term instead of a type. We also have to substitute s in the result type of t (in
 754 the case of types its always $*$, which obviously has no free variables).

5.3.6. Parameter abstraction

The rule

$$\frac{\Theta \mid \Gamma_1, x : A \vdash B : \Gamma_2 \rightarrow *}{\Theta \mid \Gamma_1 \vdash (x).B : (x : A, \Gamma_2) \rightarrow *} \text{ (Param-Abstr)}$$

gets rewritten to

$$\frac{\Phi \mid \Theta \mid \Gamma_1, x : A \vdash B : \Gamma_2 \rightarrow *}{\Phi \mid \Theta \mid \Gamma_1 \vdash (x:\mathbf{A}).B : (x : A, \Gamma_2) \rightarrow *} \text{ (Param-Abstr)}$$

Here we just add the argument of the lambda to the expression context. Then we check the body of the lambda. In the syntax directed version we have to annotate the variable with its type, so we know which type we have to add to the context.

5.3.7. (co)inductive types

We have to separate the rule

$$\frac{\sigma_k : \Gamma_k \triangleright \Gamma \quad \Theta, X : \Gamma \rightarrow * \mid \Gamma_k \vdash A_k : *}{\Theta \mid \emptyset \vdash \rho(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A}) : \Gamma \rightarrow *} \text{ (FP-Ty)}$$

into multiple rules. First we need rules to check the definitions of (co)inductive types. These are

$$\frac{\sigma_k : \Gamma_k \triangleright \Gamma \quad \Phi \mid X : \Gamma \rightarrow * \mid \Gamma_k \vdash A_k : * \quad \vdash \phi \text{ ParCtx}}{\vdash \text{data } X\langle\Phi\rangle : \Gamma \rightarrow \text{Set where; } \overline{\text{Constr}_k : \Gamma_k \rightarrow A_k \rightarrow X\sigma_k}} \text{ (FP-Ty)}$$

and

$$\frac{\sigma_k : \Gamma_k \triangleright \Gamma \quad \Phi \mid X : \Gamma \rightarrow * \mid \Gamma_k \vdash A_k : * \quad \vdash \phi \text{ ParCtx}}{\vdash \text{codata } X\langle\Phi\rangle : \Gamma \rightarrow \text{Set where; } \overline{\text{Destr}_k : \Gamma_k \rightarrow X\sigma_k \rightarrow A_k}} \text{ (FP-Ty)}$$

Because we only allow top level definitions of (co)inductive types our rules have empty contexts. We first have to check if σ_k is a context morphism from Γ_k to Γ . This basically means that the terms in σ_k are of the types in Γ , if we check them in Γ_k . After that we have to check if the \vec{A} (the arguments where we can have a recursive occurrence) are of kind $*$. Because this is a top level definition the context ϕ is provided by the code. So we have to check if it is valid. We will now have to rewrite the rules for context morphism. Here we just add the parameter context to the rules of the paper.

$$\frac{}{\Phi \vdash () : \Gamma_1 \triangleright \emptyset} \quad \frac{\Phi \vdash \sigma : \Gamma_1 \triangleright \Gamma_2 \quad \Phi \mid \Gamma_1 \vdash t : A[\sigma]}{\Phi \vdash (\sigma, t) : \Gamma_1 \triangleright (\Gamma_2, x : A)}$$

We also need a rule for the cases in which we are using these defined variables. This is.

$$\frac{\Phi \mid \Theta \mid \Gamma' \vdash \vec{A} : \Gamma_i \rightarrow *}{\Phi \mid \Theta \mid \Gamma' \vdash X(\vec{A}) : \Gamma[\vec{A}] \rightarrow *} \quad 782$$

783 Here X is a data or codata definition. The parser can decide if a variable is a such a
 784 definition or a local definition. Because we are type checking on the abstract syntax
 785 tree we also know Γ and Φ' . Γ is just the context from the definition and Φ is the
 786 parameter context. Because we already typed checked this definition we just have to
 787 check if the types given for the parameters have the right kind. Then we substitute
 788 these parameters in its type. We will now give the rules for checking if a list of
 789 parameters matches a parameter context.

$$\frac{}{\Phi \mid \Theta \mid \Gamma \vdash () : ()} \quad \frac{\Phi \mid \Theta \mid \Gamma \vdash A : \Gamma' \rightarrow * \quad \Phi \mid \Theta \mid \Gamma \vdash \vec{A} : \Phi'[A/X]}{\Phi \mid \Theta \mid \Gamma \vdash A, \vec{A} : (X : \Gamma' \rightarrow *, \Phi')} \quad 790$$

791 We just check every variable for the kinds in Φ' one after the other. We also have
 792 to substitute the type into the context. Because kinds in a parameter context can
 793 depend on variables previously defined in this context.

794 5.3.8. Constructor and Destructor

795 The rule for constructors

$$\frac{\mu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A}) : \Gamma \rightarrow * \quad 1 \leq k \leq |\vec{A}|}{\vdash a_k^{\mu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})} : (\Gamma_k, y : A_k[\mu/X]) \rightarrow \mu @_{\sigma_k}} \text{ (Ind-I)} \quad 796$$

797 gets rewritten to

$$\frac{\Phi \mid \Theta \mid \Gamma \vdash \vec{B} : \Phi'}{\Phi \mid \Theta \mid \Gamma \vdash \text{Constr}(\vec{B}) : (\Gamma_k[\vec{B}], y : A_k[\mu/X][\vec{B}]) \rightarrow \mu @_{\sigma_k}[\vec{B}]} \text{ (Ind-I)} \quad 798$$

799 The rule for destructors

$$\frac{\nu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A}) : \Gamma \rightarrow * \quad 1 \leq k \leq |\vec{A}|}{\vdash \xi_k^{\nu(X : \Gamma \rightarrow *; \vec{\sigma}; \vec{A})} : (\Gamma_k, y : \nu @_{\sigma_k}) \rightarrow A_k[\nu/X]} \text{ (Coind-E)} \quad 800$$

801 gets rewritten to

$$\frac{\Phi \mid \Theta \mid \Gamma \vdash \vec{B} : \Phi'}{\Phi \mid \Theta \mid \Gamma \vdash \text{Destr}(\vec{B}) : (\Gamma_k[\vec{B}], y : \nu @_{\sigma_k}[\vec{B}]) \rightarrow A_k[\nu/X][\vec{B}]} \text{ (Ind-I)} \quad 802$$

803 In the paper de/constructors are anonymous. They come together with their type.
 804 Therefor we have to check if this type is valid. Constructors construct their type. So
 805 their output value is their type μ applied to the context morphism σ_k , where k is the
 806 number of the constructor. They become as input the context Γ_k , which is implicit
 807 in the paper, and a value of type $A_k[\mu/X]$, which is the type, which can contain the
 808 recursive occurrence. Destructors are destructing their type so we get their type ν
 809 applied to σ_k as input and $A_k[\nu/X]$ as output.

In our rules, in contrast to the paper, the de/constructors refer to some type which we have already type checked. We just have to check the parameters. Every term we need is in the Haskell representation of the de/constructor. The de/constructor has the type which we have defined in the data definition. We just substitute the type itself for the free variable. At last we need to substitute the parameters for the respective variables.

5.3.9. Recursion and Corecursion

The rule

$$\frac{\vdash C : \Gamma \rightarrow * \quad \Delta, \Gamma_k, y_k : A_k[C/X] \vdash g_k : (C@_{\sigma_k}) \quad \forall k = 1, \dots, n}{\Delta \vdash \text{rec } (\overrightarrow{\Gamma_k, y_k}).g_k : (\Gamma, y : \mu@id_\Gamma) \rightarrow C@id_\Gamma} \text{ (Ind-E)}$$

gets rewritten to

$$\frac{\vdash C : \Gamma \rightarrow * \quad \frac{\vdash \Gamma \equiv_\beta \Gamma'[\vec{D}]}{\vdash B_k \equiv_\beta (C@_{\sigma_k}[\vec{D}])} \quad \frac{\Phi \mid \Theta \mid \Delta \vdash \vec{D} : \Phi'}{\Phi \parallel \Delta, \Gamma_k[\vec{D}], y_k : A_k[\vec{D}][C/X] \vdash g_k : B_k}}{\Phi \mid \Theta \mid \Delta \vdash \text{rec } \mu(\vec{D}) \text{ to } C; \text{Constr}_k \vec{x}_k y_k = g_k : (\Gamma, y : \mu[\vec{D}]@id_\Gamma) \rightarrow C@id_\Gamma} \text{ (Ind-E)}$$

We are recursing over some previously inductively defined type μ to some type C . This types must have the same context. Recursing is done by listing each constructor with the result, which the whole expression should have if we apply it to this constructor. This result can refer to the arguments of the constructor via the variables \vec{x}_k, y_k . The type must be the result type C applied to the σ_k of this constructor. In the syntax directed version we also have to check the parameters. We check if the types match by inferring them and compare them on beta equality.

We have a similar rule for corecursion. It

$$\frac{\vdash C : \Gamma \rightarrow * \quad \Delta, \Gamma_k, y_k : (C@_{\sigma_k}) \vdash g_k : A_k[C/X] \quad \forall k = 1, \dots, n}{\Delta \vdash \text{corec } (\overrightarrow{\Gamma_k, y_k}).g_k : (\Gamma, y : C@id_\Gamma) \rightarrow v@id_\Gamma} \text{ (Coind-I)}$$

gets rewritten to

$$\frac{\vdash C : \Gamma \rightarrow * \quad \frac{\vdash \Gamma \equiv_\beta \Gamma'[\vec{D}]}{\vdash B_k \equiv_\beta A_k[\vec{D}][C/X]} \quad \frac{\Phi \mid \Theta \mid \Delta \vdash \vec{D} : \Phi'}{\Phi \parallel \Delta, \Gamma_k[\vec{D}], y_k : (C@_{\sigma_k}[\vec{D}]) \vdash g_k : B_k}}{\Phi \mid \Theta \mid \Delta \vdash \text{corec } C \text{ to } v(\vec{D}); \text{Destr}_k \vec{x}_k y_k = g_k : (\Gamma, y : C@id_\Gamma) \rightarrow v[\vec{D}]@id_\Gamma} \text{ (Coind-I)}$$

A corecursion produces a coinductive type v . We have to give it a type C and list the destructors together with the expression they should be destructed to. We get the syntax directed rule analog as in the case of recursion.

836 5.4. Evaluation

837 There are two kinds of reduction steps in this system. The implementation of this is
 838 in `Eval.hs`. Will give the formal definition in the following.

The first is a reduction on the type level, \longrightarrow . It is defined like follows.

$$((x).A)@t \longrightarrow_p A[t/x]$$

839 It is standard beta reduction. If we apply a lambda $(x).A$ to a term t we substitute
 840 this term for the binding variable x in the body. This body is then the result of the
 841 reduction.

842 The other is the reduction on the term level, \succ . To define this reduction we need a
 843 action on types (written $\widehat{C}(A)$) and terms (written $\widehat{C}(t)$), where the following holds.

$$844 \frac{X : \Gamma_1 \rightarrow * \mid \Gamma'_2 \vdash C : \Gamma_2 \rightarrow * \quad \Gamma_1, x : A \vdash t : B}{\Gamma'_2, \Gamma_2, x : \widehat{C}(A) \vdash \widehat{C}(t) : \widehat{C}(B)}$$

845 Here we have a type C with a free type variable X and a term t of type B with a free
 846 term variable x of type A . If we use the action of this type on t we get a term with a
 847 type of this action on B . This term contains a free term variable x of type, the action
 848 applied to A . The type action is implemented in the module `TypeAction.hs`. Both
 849 the type action and the evaluation are done in the `Eval` monad. This monad has
 850 access to the previously defined declarations. We will now define the type action.

Definition 1. *Let $n \in \mathbb{N}$ and $1 \leq i \leq n$. Let:*

$$\begin{aligned} X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow * \\ \Gamma_i \vdash A_i : * \\ \Gamma_i \vdash B_i : * \\ \Gamma_i, x : A_i \vdash t_i : B_i \end{aligned}$$

Then we define the type action on terms inductively over C

$$\begin{aligned}
\widehat{C}(\vec{t}, t_{n+1}) &= \widehat{C}(\vec{t}) && \text{for } (\mathbf{TyVarWeak}) \\
\widehat{X}_i(\vec{t}) &= t_i \\
\widehat{C'@s}(\vec{t}) &= \widehat{C'}(\vec{t})[s/y], && \text{for } \Theta \mid \Gamma' \vdash C' : (y, \Gamma) \rightarrow * \\
(\widehat{y}).\widehat{C'}(\vec{t}) &= \widehat{C'}(\vec{t}), && \text{for } \Theta \mid (\Gamma', y) \vdash C' : \Gamma \rightarrow * \\
\mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}) &= \overrightarrow{\text{rec}^{R_A}(\Delta_k, x).g_k@\text{id}_\Gamma@x} && \text{for } \Theta, Y : \Gamma \rightarrow * \mid \Delta_k \vdash D_k : * \\
&\text{with } g_k = \alpha_k^{R_B}@\text{id}_{\Delta_k}@\widehat{D_k}(\vec{t}, x) \\
&\text{and } R_A = \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}[(\Gamma_i).\vec{A}/\vec{X}]) \\
&\text{and } R_B = \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}[(\Gamma_i).\vec{B}/\vec{X}]) \\
\nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}) &= \overrightarrow{\text{corec}^{R_B}(\Delta_k, x).g_k@\text{id}_\Gamma@x} && \text{for } \Theta, Y : \Gamma \rightarrow * \mid \Delta_k \vdash D_k : * \\
&\text{with } g_k = \widehat{D_k}(\vec{t}, x)[(\xi_k^{R_A}@\text{id}_{\Delta_k}@x)/x] \\
&\text{and } R_A = \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}[(\Gamma_i).\vec{A}/\vec{X}]) \\
&\text{and } R_B = \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}[(\Gamma_i).\vec{B}/\vec{X}])
\end{aligned}$$

And the type action on types as follow

$$\widehat{C}(\vec{A}) = C[(\Gamma_i).\vec{A}/\vec{X}]@\text{id}_\Gamma$$

851 The type action generates a term with a free variable x . In the type of this term
852 we have changed all the free variables to the types of \vec{t} . We will show the proof in
853 appendix A.

854 The reduction on terms is subdivided into an reduction on recursion and one on
855 corecursion. Here $\sigma_k \bullet \tau$ is a context morphism, where we first substitute with τ and
856 then with σ_k .

The reduction on recursion is defined as follows

$$\overrightarrow{\text{rec}(\Gamma_k, y_k).g_k}@\text{id}_\Gamma@x > g_k[\widehat{A_k}(\overrightarrow{\text{rec}(\Gamma_k, y_k).g_k}@\text{id}_\Gamma@x)/y_k][\tau, u]$$

857 If we apply a recursion $\overrightarrow{\text{rec}(\Gamma_k, y_k).g_k}$ to this context morphism and a constructor
858 $\alpha_k@\tau@u$, which is fully applied, we lookup the case for this constructor. In this case
859 we substitute τ for the variables from Γ_k and u , where we apply the recursion to
860 all recursive occurrences, for y_k . For this application we need the type action. So a
861 recursion is destructing an inductive type and all its recursive occurrences to another
862 type, while we use different cases for the different constructors of the type.

On the contrary corecursion is constructing a coinductive type. It is defined like follows.

$$\xi_k@\tau@(\overrightarrow{\text{corec}(\Gamma_k, y_k).g_k}@\text{id}_\Gamma@x) > \widehat{A_k}(\overrightarrow{\text{corec}(\Gamma_k, y_k).g_k}@\text{id}_\Gamma@x)[g_k/x][\tau, u]$$

863 If we apply a destructor together with its arguments for it context $\xi_k @ \tau$, on such a
 864 construction $(\text{corec}(\overline{\Gamma_k}, y_k). \overrightarrow{g_k} @ (\sigma_k \bullet \tau) @ u)$, we are taking the case of this destructor.
 865 In this case we are applying the corecursion to all recursive occurrences. τ and u are
 866 substituted as in recursion.

867 6. Examples

868 In this section we reiterate the example types from the paper. We use our syntax,
869 which is defined in 5.1. We will also show some functions on these types. On some
870 of them we will show the reduction steps in detail.

871 6.1. Terminal and Initial Object

872 The terminal object is a type which has exactly one value. In category theory every
873 object in the category has an unique morphism to it. We define it as a coinductive
874 type **Terminal** with no destructors . It gets a terminal and returns a terminal. To
875 get a terminal value we use corecursion on the unit type, which is the first class
876 terminal object.

```
877 codata Terminal : Set where  
878 terminal = corec Unit to Terminal where @ ◇
```

879 Contrary to the definition in the paper there is no destructor **Terminal**. In the
880 paper definitions of coinductive or inductive types need at least one de/constructor.
881 Therefore our definition wouldn't work.

882 The initial object is a type which has no values. In category theory it is the object
883 which has an unique morphism to every other object in the category. We define
884 it inductively as **Intial** with no constructor. In the paper it is defined with one
885 constructor. This constructor want's one value of the same type. We can't have a
886 value of this type, because to get one we already need one. Our way of defining it
887 is shorter and more clear. We can't construct an value of this type because we have
888 no constructors. If we could get something of type **Intial**, we could generate with
889 **exfalsum** a value of arbitrary type **C**.

```
890 data Initial : Set where  
891 exfalsum(C : Set) = rec Initial to C where
```

892 6.2. Natural Numbers and Extended Naturals

893 We use the classical peano numbers to define natural numbers. Therefor we use
894 the inductive type **Nat** with the constructors **Zero** and **Suc**. **Zero** is just the number
895 zero. Every constructor has to have an argument, which can contain a recursive

Chapter 6. Examples

occurrence. Every Type **A** is isomorphic to the function type **Terminal** \rightarrow **A**. So we use **Terminal** for this occurrence. **Suc** is the successor. So the meaning of **Suc n** is $n + 1$.

```

899 data Nat : Set where
900     Zero : Terminal  $\rightarrow$  Nat
901     Suc : Nat  $\rightarrow$  Nat
902 zero = Zero @  $\diamond$ 
903 one = Suc @ zero

```

We can then define a identity recursion on it to see how reduction works. It's a recursion which goes from a **Nat** to **Nat** and gives back in every case its input.

```

906 id = rec Nat to Nat where
907     Zero u = Zero @ u
908     Succ n = Succ @ n

```

We use it on one to see all cases.

```

910 id @ one = id @ (Succ @ zero)
911           > Succ @ n[ $\widehat{X}(\text{id @ x})/n$ ] [zero]
912           = Succ @  $\widehat{X}(\text{id @ x})$  [zero]
913           = Succ @ (id @ x)[zero]
914           = Succ @ (id @ zero)
915           = Succ @ (id @ (Zero @  $\diamond$ ))
916           > Succ @ (Zero @ u[ $\widehat{\text{Unit}}(\text{id @ x})/u$ ][ $\diamond$ ])
917           = Succ @ (Zero @ u[ $\widehat{\text{Unit}}(\text{id @ x})/u$ ][ $\diamond$ ])
918           = Succ @ (Zero @  $\widehat{\text{Unit}}(\text{id @ x})$ )[ $\diamond$ ]
919           = Succ @ (Zero @ x)[ $\diamond$ ]
920           = Succ @ (Zero @ x) = Succ @ zero = one

```

As expected the identity recursion applied to one gives back one.

We will now define extended naturals. There are also called conat. There are natural numbers with an additional value, infinity. We define it coinductively with the predecessor as its only destructor. The predecessor is either not defined or another natural number. We use the type **Maybe** to describe something which is either present (the constructor **Just**) or absent (the constructor **Nothing**). We can define the successor as a corecursion. The predecessor of the successor of **x** is just **x**. So the only case of corec returns a **Just x** (remember **Prec** returns a **Maybe(Conat)** not a **Conat**).

```

929 data Maybe(A : Set) : Set where
930     Nothing : Unit  $\rightarrow$  Maybe
931     Just : A  $\rightarrow$  Maybe
932 nothing(A) = Nothing(A) @  $\diamond$ 
933 codata Conat : Set where
934     Prec : Conat  $\rightarrow$  Maybe(Conat)
935 succ = corec Conat to Conat where
936     Prec x = Just(Conat) @ x

```

We now define the values zero and infinity

```

938 zero = (corec Unit to Conat where
939     {Prev x = nothing(Unit)}) @  $\diamond$ 
940 infinity = (corec Unit to Conat where
941     {Prev x = Just(Conat) @ x}) @  $\diamond$ 

```


For **zero** the predecessor is absent, there is no predecessor of 0 in the natural numbers, so we give pack **Nothing**. We then have to apply the **corec** to \diamond to get the value. The predecessor of **infinity** should also be **infinity**. We apply the **corec** to another **Conat**, so the **x** is also a **Conat**. We will know see that the predecessor on this values give back the right value.

$$\begin{aligned}
\text{Prev @ zero} &> \widehat{\text{Maybe}(X)} \left(\underbrace{\text{corec Unit to Conat where } \{\text{Prev } x = \text{nothing}(\text{Unit})\}}_{t_1} @ x \right) [\text{nothing}(\text{Unit})/x][\diamond] \\
&= \text{rec Maybe}(\text{Unit}) \text{ to Maybe}(\text{Conat}) \text{ where} \\
&\quad \{\text{Nothing } u = \text{Nothing}(\text{Conat}) @ \widehat{\text{Unit}}(t_1, u) \\
&\quad \text{Just } c = \text{Just}(\text{Conat}) @ \widehat{X}(t_1, c)\} @ x[\text{nothing}(\text{Unit})/x][\diamond] \\
&= \underbrace{\text{rec Maybe}(\text{Unit}) \text{ to Maybe}(\text{Conat}) \text{ where} \{\text{Nothing } u = \text{Nothing}(\text{Conat}) @ u, \text{Just } c = \text{Just}(\text{Conat}) @ t_1\}}_{t_2} @ \text{nothing}(\text{Unit}) \\
&> \text{Nothing}(\text{Conat}) @ u[\widehat{\text{Unit}}(t_2 @ x)/u][\diamond] \\
&= \text{Nothing}(\text{Conat}) @ u[x/u][\diamond] \\
&= \text{Nothing}(\text{Conat}) @ \diamond
\end{aligned}$$

$$\begin{aligned}
\text{Prev @ infinity} &> \widehat{\text{Maybe}(X)} \left(\underbrace{\text{corec Unit to Conat where } \{\text{Prev } x = \text{Just}(\text{Unit}) @ x\}}_{t_1} @ x \right) [\text{Just}(\text{Unit}) @ /x][\diamond] \\
&= \text{rec Maybe}(\text{Unit}) \text{ to Maybe}(\text{Conat}) \text{ where} \\
&\quad \{\text{Nothing } u = \text{Nothing}(\text{Conat}) @ \widehat{\text{Unit}}(t_1, u) \\
&\quad \text{Just } c = \text{Just}(\text{Conat}) @ \widehat{X}(t_1, c)\} @ x[\text{Just}(\text{Unit}) @ /x][\diamond] \\
&= \underbrace{\text{rec Maybe}(\text{Unit}) \text{ to Maybe}(\text{Conat}) \text{ where} \{\text{Nothing } u = \text{Nothing}(\text{Conat}) @ u, \text{Just } c = \text{Just}(\text{Conat}) @ t_1\}}_{t_2} @ \text{Just}(\text{Unit}) @ \\
&> \text{Just}(\text{Conat}) @ t_1[\widehat{\text{Unit}}(t_2 @ x)/x][\diamond] \\
&= \text{Just}(\text{Conat}) @ t_1[x/x][\diamond] \\
&= \text{Just}(\text{Conat}) @ \text{infinity}
\end{aligned}$$

942

943 6.3. Binary Product and Coproduct

944 The product is defined as a coinductive type. It has two destructors. The first gives
 945 back the first element. And the second the second. To use this type, the types A and
 946 B have to be instantiated to concrete types. We don't have type polymorphism in our
 947 language. We also define a pair expression which generates a pair over corecursion.

```

948 codata Product(A : Set , B : Set) : Set where
949     Fst : Product → A
950     Snd : Product → B

```

```

951 pair⟨A : Set , B : Set⟩ (x:A, y:B) = corec Unit where
952                                     { Fst u → x
953                                     ; Snd u → y } @ ◇

```

For types with other contexts we have to define different Products. For example if **B** depends on **Nat**, we define product like the following.

```

956 codata Pair⟨A : Set , B : (n : Nat) → Set⟩ : (n : Nat) → Set where
957   First : (n : Nat) → Pair n → A
958   Second : (n : Nat) → Pair n → B @ n

```

Here the product also depends on **Nat**. If **A** or **B** depends on values the product must also depend on these values. This is the product, which is used for the definition of vectors in [BG16].

On **Product** we can define the swap function.

```

963 swap⟨A : Set , B : Set⟩ =
964   corec Product⟨A,B⟩ to Product⟨B,A⟩ where
965     Fst x → Snd x
966     Snd x → Fst x

```

This is a well typed function as shown by the following proof

$$\begin{array}{c}
 (A : *, B : *) \parallel (x : A) \vdash \text{Snd} @ x : \text{Product}\langle A, B \rangle \text{ (a)} \\
 \hline
 (A : *, B : *) \parallel \vdash \text{Product}\langle A, B \rangle : * \quad (A : *, B : *) \parallel (y : B) \vdash \text{Fst} @ y : \text{Product}\langle A, B \rangle \text{ (b)} \\
 \hline
 (A : *, B : *) \parallel \vdash \text{swap} : (p : \text{Product}\langle A, B \rangle) \rightarrow \text{Product}\langle B, A \rangle
 \end{array}$$

We show (a) in the following proof. (b) works analog.

$$\begin{array}{c}
 (A : *, B : *) \parallel (x : A) \vdash \text{Snd} : (x : A) \rightarrow \text{Product}\langle A, B \rangle \quad \frac{(x : A)(x) \rightsquigarrow A}{(x : A) \vdash x : A} \\
 \hline
 (A : *, B : *) \parallel (x : A) \vdash \text{Snd} @ x : \text{Product}\langle A, B \rangle
 \end{array}$$

For brevity we omitted the beta equality premises and the checking for of the parameters. The beta equality premises wouldn't be interesting because they all already syntactically identical.

The Binary Coproduct corresponds to the Either type in Haskell. It is defined as an inductive type. It is either **A** or **B**. We have one constructor **Left** for **A** and one constructor **Right** for **B**.

```

977 data Coproduct⟨A,B⟩ : Set where
978   Left  : A → Coproduct
979   Right : B → Coproduct

```

6.4. Sigma and Pi Type

The sigma type is a dependent pair of two types. The second type can depend on the value of the first type. It corresponds to exists in logic. We define it as an inductive type and call the constructor **Exists**.

```

984 data Sigma(A : Set , B : (x : A) → Set) : Set where
985     Exists : (x:A) → B x → Sigma

```

986 The pi type is a generalization of the function type to dependent types. The type
 987 of the codomain or result of a function can depend on the value We define it as a
 988 coinductive type. To destruct a function we just apply it to a value. So the destructor
 989 is **Apply**.

```

990 codata Pi(A : Set , B : (x : A) → Set) : Set where
991     Apply : (x : A) → Pi x → B

```

992 To construct a function we use corecursion on **Unit**. The identity function is defined
 993 like this

```

994 id(A : Set) = corec Unit to Pi(A,(v:A).A) where
995     { Apply v p = v } @ ϕ

```

996 Evaluation on one goes as follows.

```

997 apply = Apply(Nat,(v : Nat).Nat)
998 one = S @ (Z @ )
999 apply @ id(Nat) @ one
1000 = apply @ one @ ((corec Unit to Pi(Nat,(x:Nat).Nat) where
1001     Apply v p = v ) @ ϕ)
1002 > Nat  $\left( \underbrace{\text{corec Unit to Pi where } \{ \text{Apply } v \_ = v \} @ x}_t \right) [v/x] [one, \phi]$ 
1003 = (rec Nat to Nat where
1004     Zero x = Zero @ (Unit(t,x))
1005     Succ x = Suc @ (Y(t,x)) @ x[v/x] [one, ϕ]
1006 = (rec Nat to Nat where
1007     Zero x = Zero @ (Unit(t))
1008     Succ x = Suc @ x @ x[v/x] [one, ]
1009 = (rec Nat to Nat where
1010     Zero x = Zero @ (Unit())
1011     Succ x = Suc @ x @ x[v/x] [one, ϕ]
1012 = (rec Nat to Nat where
1013     Zero x = Zero @ x
1014     Succ x = Suc @ x @ x[v/x] [one, ϕ]
1015 = (rec Nat to Nat where
1016     Zero x = Zero @ x
1017     Succ x = Suc @ x @ v [one, ϕ]
1018 = (rec Nat to Nat where
1019     Zero x = Zero @ x
1020     Succ x = Suc @ x @ one
1021 = one

```

1022 6.5. Vectors and Streams

1023 Vectors are a standard example for dependent types. They are like lists, except their
 1024 type depends on their length. For example a vector [1;2] has type **Vector**(Nat) 2,
 1025 because its length is 2. It has 2 constructors **Nil** and **Cons** like lists. **Nil** gives back

Chapter 6. Examples

the empty vector. Because the length of the empty vector is zero its return type is **Vector 0**. The second constructor **Cons** takes a natural number **k**, a value of type **A** and a vector of length **k**, a **Vector k**. It returns a new vector. Its head is the first argument and its tail the second. So the results length is one more then the second argument . Therefore it is **Vector (Suc k)**. In [BG16] the head and tail are encoded in a pair.

```

1032 data Vector(A : Set) : (n:Nat) → Set where
1033   Nil : Unit → Vector zero
1034   Cons : (k:Nat, v:A) → Vector @ k → Vector (Suc @ k)
1035   nil(A : Set) = Nil(A : Set) @ ∅

```

The function **extend** takes a value **x** and extends it to a vector.

```

1037 extend(A : Set) =
1038   rec Vec(A) to ((x).Vec(A) @ (Suc x) where
1039     Nil u = Cons(A) @ x @ nil(A)
1040     Cons k v = Cons(A) @ (Suc @ k) @ v

```

The type checking of this function goes as follows.

$$\begin{array}{c}
 (A : \text{Set}) \Vdash (x).(\text{Vec}(A) @ (\text{Suc } @ x)) : (k : \text{Nat}) \rightarrow * \\
 (A : \text{Set}) \Vdash (u : A) \vdash \text{Cons}(A) @ 0 @ (\text{Nil}(A @)) : (x).(\text{Vec}(A) @ (\text{Suc } @ x)) @ 0 \\
 \hline
 (k : \text{Nat}, v : (x).(\text{Vec } @ (\text{Suc } @ x)) @ k) \vdash \text{Cons}(A) @ (\text{Suc } @ k) @ v : (x).(\text{Vec } @ (\text{Suc } @ x)) @ (\text{Suc } @ k) \\
 \hline
 \vdash \text{extend}(A) : (k : \text{Nat}, y : \text{Vec}(A) @ k) \rightarrow (x).(\text{Vec}(A) @ (\text{Suc } x)) @ k
 \end{array}$$

As an example we evaluate a vector of length 1 with this function. We choose length one to see all rec cases.

```

extend(Nat) @ 1 @ (Cons(Nat) @ 0 @ 0 @ nil(Nat))
= extend(Nat) @ (Suc @ k • 0) @ (Cons(Nat) @ 0 @ 0 @ nil(Nat))
> Cons(Nat) @ (Suc @ k) @ v [X @ k (extend(Nat) @ n @ x) / v] [0, nil(Nat)]
= Cons(Nat) @ (Suc @ k) @ v [X (extend(Nat) @ n @ x) [k/n] / v] [0, nil(Nat)]
= Cons(Nat) @ (Suc @ k) @ v [extend @ n @ x [k/n] / v] [0, nil(Nat)]
= Cons(Nat) @ (Suc @ k) @ v [extend @ k @ x / v] [0, nil(Nat)]
= Cons(Nat) @ (Suc @ k) @ (extend @ k @ x) [0, nil(Nat)]
= Cons(Nat) @ (Suc @ 0) @ (extend @ 0 @ (nil(Nat)))
= Cons(Nat) @ 1 @ (extend @ 0 @ (Nil(Nat) @ ))
> Cons(Nat) @ 1 @ (Cons(Nat) @ 0 @ (Nil(Nat) @ )) [Unit(extend @ k @ x) / u] [∅]
= Cons(Nat) @ 1 @ (Cons(Nat) @ 0 @ (Nil(Nat) @ x)) [∅]
= Cons(Nat) @ 1 @ (Cons(Nat) @ 0 @ (Nil(Nat) @ ))

```

Here we write 1 for **Suc @ (Zero @)** and 0 for **Zero @ ∅**.

With the help of extended naturals, we can define partial streams. These are streams which depend on there definition depth. Like non-dependent streams they are coinductive and have 2 destructors for head and tail.

```

1047 codata PStr(A : Set): (n: ExNat) → Set where
1048   hd : (k : ExNat) → PStr(A) (succE k) → A
1049   tl : (k : ExNat) → PStr(A) (succE k) → PStr(A) @ k

```

1050 These streams are like vectors except they also can be infinite long. This is in contrary
 1051 to non dependent streams. A non dependent stream could not be of length zero.
 1052 Because then a call of **hd** and **tl** on it wouldn't be defined. In the dependent case
 1053 the type checker wouldn't allow such a call because **hd** and **tl** expect streams which
 1054 are at least of length one. We can then define **repeat**.

```
1055 repeat<A : Set>(x : A, n : Conat) =
1056   corec (n : Conat).Unit to PStr<A> where
1057     { Hd k s = x
1058       ; Tl k s = () } @ n @ ()
```

1059 This function gets a value and an extended natural number. It generate an constant
 1060 partial stream of that value with the number as its length.

1061 7. Conclusion

1062 We have implemented a depend type theory with inductive and coinductive types. In
1063 this theory, contrary to coq and agda, coinductive types can also depend on values.
1064 Contrary to the theory of the paper we can define schemata like **Maybe** $\langle \mathbf{A} : \mathbf{Set} \rangle$
1065 where **A** can be an arbitrary type of kind **Set**.

1066 One downside is that we don't have universes. This prevents type polymorphism.
1067 Further work needs to be done to solve this. Another problem is, that each con-
1068 structor or destructor has at least one argument. The argument with the recursive
1069 occurrence. For example we have to apply an unit to the constructors of a boolean
1070 type. We could allow recursive occurrences in the contexts of the constructors and
1071 destructors. This makes it possible to remove the argument with the recursive oc-
1072 currence. We then have to change the evaluation rules.

1073 Our system allowed us to define the (depenend) function type. Therefor we don't
1074 have it as primitive expression. We are hopeful, that in the future we get an more
1075 mainstream language, like Coq or Agda, where the depended function is definable.
1076 As already mentioned in the introduction this would lead to a symmetrical language.

1077 A. Type action proof

1078 **Theorem 1.** $(\Gamma).A@id_\Gamma \leftrightarrow_T A$

1079 *Proof.* We show this by induction on the length of Γ

- $\Gamma = \epsilon$:

$$A \longleftrightarrow_T A$$

- $\Gamma = x : B, \Gamma'$:

$$(x : B, \Gamma').A@x@id_{\Gamma'} \longrightarrow_p (\Gamma').A@id_{\Gamma'}[x/x] = (\Gamma').A@id_{\Gamma'} \xleftrightarrow{IdH}_T A$$

1080

□

1081 **Theorem 2.** *The following rule holds*

$$\frac{x : A \vdash t : B \quad A \longleftrightarrow_T A'}{x : A' \vdash t : B}$$

1083 *Proof.* We show this by induction on t

□

1084 **Theorem 3.** *The typing rule (5) in the paper holds*

$$\frac{X : \Gamma_1 \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow * \quad \Gamma_1, x : A \vdash t : B}{\Gamma', \Gamma, x : \widehat{C}(A) \vdash \widehat{C}(t) : \widehat{C}(B)}$$

1086 *Proof.* First we will generalize the rule to

$$\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow * \quad \Gamma_i, x : A_i \vdash t_i : B_i}{\Gamma', \Gamma, x : \widehat{C}(\vec{A}) \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})}$$

1088 Then we gonna show it by Induction on the derivation \mathcal{D} of C

- $\mathcal{D} = \frac{}{\vdash \top : *} \text{ (}\tau\text{-I)}$

Then the type actions got calculated as follows

$$\begin{aligned} \widehat{\top}(\vec{A}) &= \widehat{\top}() = \top \\ \widehat{\top}(\vec{t}) &= \widehat{\top}() = x \\ \widehat{\top}(\vec{B}) &= \widehat{\top}() = \top \end{aligned}$$

1090 We than got the following prooftree

$$1091 \quad \frac{\vdash \top : *}{x : \top \vdash x : \top} \text{ (Proj)}$$

$$1092 \quad \bullet \mathcal{D} = \frac{\frac{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrow *, \dots, X_{n-1} : \Gamma_{n-1}} \text{ TyCtx} \quad \frac{\mathcal{D}_2}{\Gamma_n \text{ Ctx}} \text{ TyVar-I}}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \emptyset \vdash X_n : \Gamma_n \rightarrow *}$$

Again we calculate the type actions

$$\begin{aligned} \widehat{X_n}(\vec{A}) &= X_n[(\Gamma_i).\vec{A}/\vec{X}]@id_{\Gamma_n} = X_n[(\Gamma_n).A_n/X_n]@id_{\Gamma_n} = (\Gamma_n).A_n@id_{\Gamma_n} \\ \widehat{X_n}(\vec{t}) &= t_n \\ \widehat{X_n}(\vec{B}) &= X_n[(\Gamma_i).\vec{B}/\vec{X}]@id_{\Gamma_n} = X_n[(\Gamma_n).B_n/X_n]@id_{\Gamma_n} = (\Gamma_n).B_n@id_{\Gamma_n} \end{aligned}$$

1093 We know from the first premise that $\Gamma = \Gamma_n$ and $\Gamma' = \emptyset$

1094 Here we got the prooftree

$$1095 \quad \frac{\frac{\Gamma_n, x : A \vdash t : B}{\Gamma_n, x : (\Gamma_n).A@id_{\Gamma_n} \vdash t : B} \text{Thrm. 1} \quad \frac{A \longleftrightarrow_T (\Gamma_n).A@id_{\Gamma_n}}{\Gamma_n, x : (\Gamma_n).A@id_{\Gamma_n} \vdash t_n : (\Gamma_n).B@id_{\Gamma_1}} \text{Thrm. 2}}{\Gamma_n, x : (\Gamma_n).A@id_{\Gamma_n} \vdash t_n : (\Gamma_n).B@id_{\Gamma_1}} \text{Thrm. 1 Conv}$$

$$1096 \quad \bullet \mathcal{D} = \frac{\frac{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C : \Gamma \rightarrow *} \quad \frac{\mathcal{D}_2}{\Gamma_n \text{ Ctx}}}{X_1 : \Gamma_1 \rightarrow *, \dots, X_{n+1} : \Gamma_{n+1} \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow *} \text{ (TyVar-Weak)}$$

1097 Here we got the prooftree

$$1098 \quad \frac{\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_{n+1} : \Gamma_{n+1} \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow *} (*)}{\Gamma', \Gamma, x : \underbrace{\widehat{C}(\vec{A})}_{\equiv \widehat{C}(\vec{A}, A_{n+1})} \vdash \underbrace{\widehat{C}(\vec{t})}_{\equiv \widehat{C}(\vec{t}, t_{n+1})} : \underbrace{\widehat{C}(\vec{B})}_{\equiv \widehat{C}(\vec{B}, B_{n+1})}}_{\Gamma_i, x : A_i \vdash t_i : B_i} \text{ IdH.}$$

1099 (*) Here we undo **(TyVar-Weak)**

1100 (**) X_{n+1} doesn't occur free in C, otherwise \mathcal{D}_1 wouldn't be possible

1101 (***) Case for **(TyVar-Weak)** of type actions on terms

1102 $\bullet \mathcal{D} =$

$$1103 \quad \frac{\frac{\mathcal{D}_1}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C : \Gamma \rightarrow *} \quad \frac{\mathcal{D}_2}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \mid \Gamma' \vdash D : *}}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma', y : D \vdash C : \Gamma \rightarrow *} \text{ (Ty-Weak)}$$

1104 Here we got the prooftree

$$1105 \quad \frac{\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma', y : D \vdash C : \Gamma \rightarrow *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C : \Gamma \rightarrow *} (*)}{\Gamma', \Gamma, x : \widehat{C}(\vec{A}) \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})} \text{ IdH.} \quad \frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \mid \Gamma' \vdash D : *}{\Gamma', \Gamma, x : \widehat{C}(\vec{A})y \vdash \widehat{C}(\vec{t}) : \widehat{C}(\vec{B})} \text{ (Term-Weak)}$$

1107

(*) Here we undo **(Ty-Weak)**

1108

$$\bullet \mathcal{D} = \frac{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C' : (y : D, \Gamma) \rightarrow * \quad \Gamma' \vdash s : D}{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash C' @s : \Gamma \rightarrow *} \text{ (Ty-Inst)}$$

1109

Then we got the following induction hypothesis

1110

$$\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash C' : (y : D, \Gamma) \rightarrow * \quad \Gamma_i, x : A_i \vdash t_i : B_i}{\Gamma', y : D, \Gamma, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C'}(\vec{B})}$$

Calculated type actions:

$$\widehat{C' @s}(\vec{A}) = C' @s[(\Gamma_i). \vec{A} / \vec{X}] @id_\Gamma = C'[(\Gamma_i). \vec{A} / \vec{X}] @s @id_\Gamma = \widehat{C'}(\vec{A})[s/y]$$

$$\widehat{C' @s}(\vec{t}) = \widehat{C'}(\vec{t})[s/y]$$

$$\widehat{C' @s}(\vec{B}) = C' @s[(\Gamma_i). \vec{B} / \vec{X}] @id_\Gamma = C'[(\Gamma_i). \vec{B} / \vec{X}] @s @id_\Gamma = \widehat{C'}(\vec{B})[s/y]$$

1111

We then got the following proof tree

1112

$$\frac{\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma'_2 \vdash C' @s : \Gamma_2[s/y] \rightarrow *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma'_2 \vdash C' : (y : D, \Gamma_2) \rightarrow *} (*) \quad \Gamma_i, x : A_i \vdash t_i : B_i}{\frac{\Gamma'_2, y : D, \Gamma_2, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C'}(\vec{B})}{\Gamma'_2, \Gamma_2[s/y], x : \widehat{C'}(\vec{A})[s/y] \vdash \widehat{C'}(\vec{t})[s/y] : \widehat{C'}(\vec{B})[s/y]}} \text{ IdH.}$$

1113

(*) This is the reverse of **(Ty-Inst)**.

1114

$$\bullet \mathcal{D} = \frac{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma', y : D \vdash C' : \Gamma \rightarrow *}{X_1 : \Gamma_1, \dots, X_n : \Gamma_n \mid \Gamma' \vdash (y). C' : (y : D, \Gamma) \rightarrow *} \text{ (Param-Abstr)}$$

Calculated type actions:

$$\begin{aligned} (\widehat{y}). \widehat{C'}(\vec{A}) &= (y). C'[(\Gamma_i). \vec{A} / \vec{X}] @id_\Gamma \\ &= (y). (C'[(\Gamma_i). \vec{A} / \vec{X}]) @y @id_\Gamma \\ &\longleftrightarrow_T (C'[(\Gamma_i). \vec{A} / \vec{X}]) @id_\Gamma \\ &= \widehat{C'}(\vec{A}) \\ (\widehat{y}). \widehat{C'}(\vec{t}) &= \widehat{C'}(\vec{t}) \\ (\widehat{y}). \widehat{C'}(\vec{B}) &= (y). C'[(\Gamma_i). \vec{B} / \vec{X}] @id_\Gamma \\ &= (y). (C'[(\Gamma_i). \vec{B} / \vec{X}]) @y @id_\Gamma \\ &\longleftrightarrow_T (C'[(\Gamma_i). \vec{B} / \vec{X}]) @id_\Gamma \\ &= \widehat{C'}(\vec{B}) \end{aligned}$$

1115

The proof tree then becomes the following

Appendix A. Type action proof

$$\begin{array}{l}
1116 \quad \frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \Gamma' \vdash (y).C' : (y : D, \Gamma) \rightarrow *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid y : D, \Gamma' \vdash C' : \Gamma \rightarrow *} (*) \\
1117 \quad \frac{\Gamma_i, x : A_i \vdash t_i : B_i}{y : D, \Gamma', \Gamma, x : \widehat{C'}(\vec{A}) \vdash \widehat{C'}(\vec{t}) : \widehat{C'}(\vec{B})} \text{IdH.}
\end{array}$$

1118 (*) This is the reverse of **(Param-Abstr)**.

1119 • $\mathcal{D} =$

$$\begin{array}{l}
1120 \quad \frac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \sigma_k : \Delta_k \triangleright \Gamma \quad X_1 : \Gamma_1 \rightarrow *, \dots, X_n \rightarrow *, X : \Gamma \rightarrow * \mid \Delta_k \vdash D_k : *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n \rightarrow * \mid \emptyset \vdash \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrow *} \text{(FP-Ty)}
\end{array}$$

1121 From this we know $\Gamma' = \emptyset$

Calculated type actions:

$$\begin{aligned}
& \mu(Y : \widehat{\Gamma \rightarrow *}; \vec{\sigma}; \vec{D})(\vec{A}) \\
&= \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\overline{\Gamma_i}).\vec{A} / \vec{X}]@_{\text{id}_\Gamma} \\
&= \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\overline{\Gamma_i}).\vec{A} / \vec{X}]@_{\text{id}_\Gamma} \\
& \mu(Y : \widehat{\Gamma \rightarrow *}; \vec{\sigma}; \vec{D})(\vec{t}) \\
&= \text{rec}^{\mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\overline{\Gamma_i}).\vec{A} / \vec{X}]}(\Delta_k, x). \alpha_k @_{\text{id}_{\Delta_k}} @_{\widehat{D_k}}(\vec{t}, x) @_{\text{id}_\Gamma} @_x \\
& \mu(Y : \widehat{\Gamma \rightarrow *}; \vec{\sigma}; \vec{D})(\vec{B}) \\
&= \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\overline{\Gamma_i}).\vec{B} / \vec{X}]@_{\text{id}_\Gamma} \\
&= \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\overline{\Gamma_i}).\vec{B} / \vec{X}]@_{\text{id}_\Gamma}
\end{aligned}$$

From the assumptions

$$\begin{array}{l}
X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \emptyset \vdash \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrow * \\
\Gamma_i, x : A_i \vdash t_i : B_i
\end{array}$$

We have to proof that in **Ctx**

$$\Gamma, x : \mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\overline{\Gamma_i}).\vec{A} / \vec{B}]@_{\text{id}_\Gamma}$$

the expression

$$\text{rec}^{\mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\overline{\Gamma_i}).\vec{A} / \vec{X}]}(\Delta_k, y). \alpha_k @_{\text{id}_{\Delta_k}} @_{\widehat{D_k}}(t, y) @_{\text{id}_\Gamma} @_x$$

has type

$$\mu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\overline{\Gamma_i}).\vec{B} / \vec{X}]@_{\text{id}_\Gamma}$$

1122 We can use the induction hypothesis

$$\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow *, Y : \Gamma_{n+1} \rightarrow * \mid \Delta_k \vdash D_k : * \quad \Gamma_i, x : A_i \vdash t_i : B_i}{\Delta_k, x : \widehat{D}_k(\vec{A}, A_{n+1}) \vdash \widehat{D}_k(\vec{t}, y) : \widehat{D}_k(\vec{B}, B_{n+1})} \quad 1123$$

See appendix B for a proof of it. 1124

• $\mathcal{D} =$ 1125

$$\frac{\mathcal{D}_1 \quad \mathcal{D}_2 \quad \sigma_k : \Delta_k \triangleright \Gamma \quad X_1 : \Gamma_1 \rightarrow *, \dots, X_n \rightarrow *, X : \Gamma \rightarrow * \mid \Delta_k \vdash D_k : *}{X_1 : \Gamma_1 \rightarrow *, \dots, X_n \rightarrow * \mid \emptyset \vdash \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrow *} \text{ (FP-Ty)} \quad 1126$$

From this we know $\Gamma' = \emptyset$. 1127

Calculated type actions:

$$\begin{aligned} & \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})(\vec{A}) \\ &= \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{A} / \vec{X}]@id_\Gamma \\ &= \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{A} / \vec{X}]@id_\Gamma \\ & \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})(\vec{t}) \\ &= \text{corec}^{\nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]}(\Delta_k, x) \widehat{D}_k(\vec{t}, x)[(\xi_k @ id_{\Delta_k} @ x) / x]@id_\Gamma @ x \\ & \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})(\vec{B}) \\ &= \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]@id_\Gamma \\ &= \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]@id_\Gamma \end{aligned}$$

From the assumptions

$$\begin{aligned} & X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow * \mid \emptyset \vdash \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D}) : \Gamma \rightarrow * \\ & \Gamma_i, x : A_i \vdash t_i : B_i \end{aligned}$$

We have to proof that in **Ctx**

$$\Gamma, x : \nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_1). A / X]@id_\Gamma$$

the expression

$$\text{corec}^{\nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]}(\Delta_k, x) \widehat{D}_k(\vec{t}, x)[(\xi_k @ id_{\Delta_k} @ x) / x]@id_\Gamma @ x$$

has type

$$\nu(Y : \Gamma \rightarrow *; \vec{\sigma}; \vec{D})[(\Gamma_i). \vec{B} / \vec{X}]@id_\Gamma$$

We can use the induction hypothesis 1128

$$\frac{X_1 : \Gamma_1 \rightarrow *, \dots, X_n : \Gamma_n \rightarrow *, Y : \Gamma_{n+1} \rightarrow * \mid \Delta_k \vdash D_k : * \quad \Gamma_i, y_k : A_i \vdash t_i : B_i}{\Delta_k, y_k : \widehat{D}_k(\vec{A}, A_{n+1}) \vdash \widehat{D}_k(\vec{t}, y) : \widehat{D}_k(\vec{B}, B_{n+1})} \quad 1129$$

See appendix B for this proof. 1130

1131

□

1132

B. Bigger proofs

1133

$$\frac{\Gamma_1 \vdash \sigma : \Gamma_2 \quad \Gamma_3 \vdash \tau : \Gamma_1}{\Gamma_3 \vdash \sigma \circ \tau : \Gamma_2} (*)$$

[illegible]

1138 Bibliography

- 1139 [Abe10] Andreas Abel. Miniagda: Integrating sized and dependent types. *arXiv*
1140 *preprint arXiv:1012.4896*, 2010.
- 1141 [agd]
- 1142 [APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Co-
1143 patterns: programming infinite structures by observations. *ACM SIG-*
1144 *PLAN Notices*, 48(1):27–38, 2013.
- 1145 [BG16] Henning Basold and Herman Geuvers. Type theory based on depen-
1146 dent inductive and coinductive types. In *Proceedings of the 31st Annual*
1147 *ACM/IEEE Symposium on Logic in Computer Science*, pages 327–336,
1148 2016.
- 1149 [BJSO19] David Binder, Julian Jabs, Ingo Skupin, and Klaus Ostermann. Decom-
1150 position diversity with symmetric data and codata. *Proceedings of the*
1151 *ACM on Programming Languages*, 4(POPL):1–28, 2019.
- 1152 [Chl13] Adam Chlipala. *Certified programming with dependent types: a pragmatic*
1153 *introduction to the Coq proof assistant*. MIT Press, 2013.
- 1154 [DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless
1155 dummies, a tool for automatic formula manipulation, with application to
1156 the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*,
1157 volume 75, pages 381–392. North-Holland, 1972.
- 1158 [Our08] Nicolas Oury, 06 2008. Message on the coq-clup mailing list.
- 1159 [ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq.
1160 In *International Conference on Interactive Theorem Proving*, pages 499–
1161 514. Springer, 2014.