1 PROGRAMMING WITH Lan TYPES

We can use Lan types in several different ways in our calculus. In this note we will explore how to use Lan types to encode GADTs in our calculus. These Lan types / GADTs have several useful applications in our calculus. In addition to allowing for more general return types in data type constructors, Lan can be combined with singleton types to mimick some dependent types.

We can also define existential types and an equality type using Lan, but the latter is not really useful in the absence of type refinement(?).

The usefulness of existential types is also somewhat limited depending on what is allowed in the context of G in the ∂ rule.

1.1 Using Lan in a constructor

Our calculus does not include "GADT-syntax" (like Agda, Haskell + GADTs) in which one can make the "return type" explicit. Instead, we write a data type as the fixed point of the sum of its constructors. To produce a term of some data type with n constructors $(\mu\phi.\lambda\overline{\alpha}.F_1 + \ldots + F_n)\overline{G}$, we first construct a term

$$t : F_i[\phi :=_\beta (\mu\phi.\lambda\overline{\alpha}.F_1 + \ldots + F_n)\overline{\beta}][\overline{\alpha := G}]$$

and invoke in and some combination of inl/inr to get

$$(\mathsf{in}_{F_1+\ldots+F_n})_{\overline{G}}\,(..\mathsf{inl/inr}..\,t) : (\mu\phi.\lambda\overline{\alpha}.F_1 + \ldots + F_n)\overline{G}$$

So how are these restrictions on the return types enforced in our calculus? Although $\mathsf{in}_{F_1+\ldots+F_n}$ is a natural transformation from

$$(F_1 + \ldots + F_n)[\phi :=_\beta (\mu\phi.\lambda\overline{\alpha}.F_1 + \ldots + F_n)\overline{\beta}][\overline{\alpha := \beta}]$$

to

$$(\mu\phi.\overline{\alpha}.F_1 + \ldots + F_n)\overline{\beta}$$

we can only use the component $(\mathsf{in}_{F_1+\ldots+F_n})_{\overline{G}}$ when $F_i[\phi :=_\beta (\mu\phi.\lambda\overline{\alpha}.F_1 + \ldots + F_n)\overline{\beta}][\overline{\alpha := G}]$ is inhabited for some $1 \leqq i \leq n$.

So when $F_i = (\mathsf{Lan}^{\overline{\gamma}}_{\overline{K}} H)\overline{\alpha}$, we know from the $\int$ rule that we can only introduce terms of type $(\mathsf{Lan}^{\overline{\gamma}}_{\overline{K}} H)\overline{K[\overline{\gamma := A}]}$ for some types $\overline{A}$, so we can never use a component of $\mathsf{in}_{F_1+\ldots F_n}$ unless it has the form $(\mathsf{in}_{F_1+\ldots F_n})_{\overline{K[\overline{\gamma:=A}]}}$

For example, consider the third constructor in the type of simply-typed lambda terms:

```
data Term (A : Set) : Set where
      lit : ℕ → Term ℕ
      inc : Term ℕ → Term ℕ
      isZ : Term ℕ → Term Bool
      pair : Term A → Term B → Term (A × B)
```

$$\mathit{Term}\,\beta = (\mu\phi.\lambda\alpha.(\mathsf{Lan}^{\emptyset}_{\mathbb{N}}\,\mathbb{N})\alpha + (\mathsf{Lan}^{\emptyset}_{\mathbb{N}}\,\phi\mathbb{N})\alpha + (\mathsf{Lan}^{\emptyset}_{\mathit{Bool}}\,\phi\mathbb{N})\alpha + (\mathsf{Lan}^{\gamma_1,\gamma_2}_{\gamma_1\times\gamma_2}\,\phi\gamma_1 \times \phi\gamma_2)\alpha)\beta$$

In this case, $F_i = (\mathsf{Lan}^{\emptyset}_{\mathit{Bool}}\,\phi\,\mathbb{N})\alpha$, which is only inhabited when $\alpha = \mathit{Bool}$. The only component of $\mathsf{in}_{F_1+\ldots F_n}$ we can hope to use with this constructor is $(\mathsf{in}_{F_1+\ldots F_n})_{\mathit{Bool}}$, and this restriction corresponds exactly with the return type of `isZ` in the Agda definition.

2 SINGLETON TYPES AND FAKE DEPENDENT TYPES

Since the return types of constructors in a GADT can vary, the type of a term can now potentially give us information about its value.

We have already seen the *Term* datatype, which ensures that the lambda terms which inhabit the datatype are well-typed. This additional structure can also make programs more efficient in a practical setting by eliminating some pattern matching cases.

It turns out that we can mimick dependent types by using GADTs with "singleton types", which are type-level representations of terms. For instance, we can represent type-level natural numbers using $Z = \mathbb{1}$ and $S\,\beta = (\mu\phi.\lambda\alpha.\alpha)\beta$.

Using Z and S we can define a type of length-indexed lists:

```
data Vec (A : Set) (N : Set) : Set where
    gnil : Vec A Z
    gcons : A → Vec A N → Vec A (S N)
```

$$Vec\,\beta_1\beta_2 = (\mu\phi.\lambda\alpha_1,\alpha_2.(\mathsf{Lan}^{\gamma}_{\gamma,Z}\,1)(\alpha_1,\alpha_2) + (\mathsf{Lan}^{\gamma_1,\gamma_2}_{\gamma_1,S\,\gamma_2}\,\gamma_1 \times \phi(\gamma_1,\gamma_2))(\alpha_1,\alpha_2))(\beta_1,\beta_2)$$

and for convenience we can define some constructors
(abusing notation to write in_{Vec} for $\mathsf{in}_{(\mathsf{Lan}^{\gamma}_{\gamma,Z}\,1)(\alpha_1,\alpha_2)+(\mathsf{Lan}^{\gamma_1,\gamma_2}_{\gamma_1,S\,\gamma_2}\,\gamma_1\times\phi(\gamma_1,\gamma_2))(\alpha_1,\alpha_2)}$ and similar below):

$$\begin{aligned}
&nil : \mathsf{Nat}^{\alpha}\,\mathbb{1}\,(List\,\alpha)\\
&nil = L_{\alpha}\,*\,.(\mathsf{in}_{\mathbb{1}+\alpha\times\phi\alpha})_{\alpha}(\mathsf{inl}\,*)\\
&cons : \mathsf{Nat}^{\alpha}\,(\alpha \times List\,\alpha)\,(List\,\alpha)\\
&cons = L_{\alpha}\,t.(\mathsf{in}_{\mathbb{1}+\alpha\times\phi\alpha})_{\alpha}(\mathsf{inr}\,t)\\
&vnil : \mathsf{Nat}^{\alpha}\,\mathbb{1}\,(Vec\,\alpha\,Z)\\
&vnil = L_{\alpha}\,*\,.(\mathsf{in}_{Vec})_{\alpha,Z}(\mathsf{inl}\,(\textstyle\int_{(\gamma,Z),\mathbb{1}}\,*))\\
&vcons : \mathsf{Nat}^{\alpha,n}\,(\alpha \times Vec\,\alpha\,n)\,(Vec\,\alpha\,(S\,n))\\
&vcons = L_{\alpha,n}\,t.(\mathsf{in}_{Vec})_{\alpha,(S\,n)}(\mathsf{inr}\,(\textstyle\int_{(\gamma_1,S\,\gamma_2),\gamma_1\times Vec\,\gamma_1\,\gamma_2}\,t))
\end{aligned}$$

We can write a function to convert a vector into a list:

$$\begin{aligned}
&vecToList : \mathsf{Nat}^{\alpha,n}\,(Vec\,\alpha\,n)\,(List\,\alpha)\\
&vecToList = (\mathsf{fold}_{Vec})_{\emptyset}\,alg\\
&alg : \mathsf{Nat}^{\alpha,n}\,((\mathsf{Lan}^{\gamma}_{\gamma,Z}\,\mathbb{1})(\alpha,n) + (\mathsf{Lan}^{\gamma_1,\gamma_2}_{\gamma_1,S\,\gamma_2}\,\gamma_1 \times List\,\gamma_1)(\alpha,n))\,(List\,\alpha)\\
&alg = L_{\alpha,n}\,t.\mathsf{case}\ t\ \mathsf{of}\ \{\\
&\qquad vn \mapsto nil_{\alpha}*;\\
&\qquad vc \mapsto cons_{\alpha}(\partial^{\beta\times List\,\beta,(\gamma_1,S\,\gamma_2)}_{\gamma_1\times List\,\gamma_1}\,(id_{\gamma_1\times List\,\gamma_1})\,vc)\\
&\}
\end{aligned}$$

In *alg*, we use the following instance of the Lan-elimination rule:

$$\frac{\vdash id_{\gamma_1\times List\,\gamma_1} : \mathsf{Nat}^{\alpha,n,\gamma_1,\gamma_2}\,(\gamma_1 \times List\,\gamma_1)\,(\gamma_1 \times List\,\gamma_1) \qquad \emptyset;\alpha,n \mid \emptyset \vdash vc : (\mathsf{Lan}^{\gamma_1,\gamma_2}_{\gamma_1,S\,\gamma_2}\,\gamma_1 \times List\,\gamma_1)\,(\alpha,n)}{\emptyset;\alpha,n \mid \emptyset \vdash \partial^{\beta_1\times List\,\beta_1,(\gamma_1,S\,\gamma_2)}_{\gamma_1\times List\,\gamma_1}\eta\,vc : (\beta_1 \times List\,\beta_1)[\beta_1 := \alpha][\beta_2 := n]}$$

to produce a term of type $\alpha \times List\,\alpha$. We can also convert lists to vectors, but this is tackled in the next section with existential types.

2.1 Safe(?) head

Given a the type of length-indexed vectors, one might ask whether we can write a *head* function of type $\mathsf{Nat}^{\alpha,n}(Vec\,\alpha\,(S\,n))\,(\alpha)$ in which we only need to give the case for the *vcons* constructor.

$$
\begin{aligned}
&\mathit{head} : \mathsf{Nat}^{\alpha,n}(\mathit{Vec}\,\alpha\,(S\,n))\;(\alpha)\\
&\mathit{head} = L_{\alpha,n}\,t.\mathsf{case}\,((\mathsf{in}^{-1}_{\mathit{Vec}})_{\alpha,S\,n}\,t)\,\mathsf{of}\{\\
&\qquad \upsilon n : (\mathsf{Lan}^{\gamma}_{\gamma,Z}\,1)(\alpha,S\,n) \mapsto ???;\\
&\qquad \upsilon c : (\mathsf{Lan}^{\gamma_1,\gamma_2}_{\gamma_1,S\,\gamma_2}\,\gamma_1 \times \mathit{Vec}\,\gamma_1\,\gamma_2)(\alpha,S\,n) \mapsto \partial^{\beta_1,(\gamma_1,\gamma_2)}_{\gamma_1\times\mathit{Vec}\,\gamma_1\,\gamma_2}\,(L_{\alpha,n,\gamma_1,\gamma_2}\,x.\pi_1 x)\,\upsilon c\\
&\}
\end{aligned}
$$

In the second case above, we can use the following instance of the Lan-elimination rule to extract a term of type α from a term of type $(\mathsf{Lan}^{\gamma_1,\gamma_2}_{\gamma_1,S\,\gamma_2}\,\gamma_1 \times \mathit{Vec}\,\gamma_1\,\gamma_2)(\alpha,n)$:

$$
\frac{\vdash \eta : \mathsf{Nat}^{\alpha,n,\gamma_1,\gamma_2}\,(\gamma_1 \times \mathit{Vec}\,\gamma_1\,\gamma_2)\;(\gamma_1) \qquad \emptyset;\alpha,n \mid \upsilon c \vdash \upsilon c : (\mathsf{Lan}^{\gamma_1,\gamma_2}_{\gamma_1,S\,\gamma_2}\gamma_1 \times \mathit{Vec}\,\gamma_1\,\gamma_2)\,(\alpha,S\,n)}{\emptyset;\alpha,n \mid \upsilon c \vdash \partial^{\beta_1,(\gamma_1,S\,\gamma_2)}_{\gamma_1\times\mathit{Vec}\,\gamma_1\,\gamma_2}\,\eta\,\upsilon c : (\beta_1)[\beta_1 := \alpha][\beta_2 := S\,n]}
$$

Unfortunately, we have no way of producing a term of type α in the first case. In a practical setting, the type unification algorithm would determine at this point that the first constructor pattern need not be matched, because Z cannot be unified with $S\,n$ for any n. It is also true in our calculus that, due to the Lan-introduction rule, it is impossible to construct a term of type $(\mathsf{Lan}^{\overline{\gamma}}_{\overline{K}}\,F)\overline{B}$ unless $B_i = K_i[\overline{\gamma := A}]$ for $i = 1, \ldots, |\overline{K}|$ and some types $\overline{A}$. In particular, it is impossible to construct a term of type $(\mathsf{Lan}^{\gamma}_{\gamma,Z}\,1)(\alpha,S\,n)$, so assuming its existence should allow us to do something akin to $\bot$-eliimination in the first case above. I don't know how we could do this because it seems we would need type-equality at the meta-level in order to have a rule that can check the arguments of a Lan type in this way. This concern is probably orthogonal to the goals of this project, but without it, the "fake dependent types" are not incredibly useful.

Note about S and Z

Note that we cannot write a function from the type-level natural numbers to the natural numbers data type. We can write each instance, e.g. $Zto0 : \mathsf{Nat}^{\emptyset}(Z)\,(\mu\beta.\mathbb{1} + \beta)$, $SZto1 : \mathsf{Nat}^{\emptyset}(S\,Z)\,(\mu\beta.\mathbb{1} + \beta)$, but we cannot write them all in a single function because S and Z are separate types. (What would the type be? $\mathsf{Nat}^{\alpha}(Z + S\,\alpha)(\mu\beta.\mathbb{1} + \beta)$? This is too generic because it must be defined for all α.) We address this issue with the following GADT:

```
data GNat (A : Set) : Set where
     gzero : GNat Z
     gsucc : GNat A → GNat (S A)
```

Note that using *GNat* in the definition of *Vec* would required having a Lan type (*GNat*) in the subscript of another Lan (a constructor for *Vec*).

3 EXISTENTIAL TYPES

The following defines an existential type in Agda:

```
data Exists (F : Set → Set) : Set₁ where
     e : F A → Exists F
```

The aspect that makes it existential is that the type `A` used in the constructor argument is hidden in the return type. To provide a witness to the fact that there exists a type `A` such that `List A` is inhabited, we just need to provide an inhabitant of `List A` for some type `A`.

Unfortunately, we cannot define an exact analogue (with closed type) of the `Exists` datatype in our calculus because it is parameterized over a 1-ary variable (`F`), and we can only bind 0-ary variables in our types (we can bind other variables with μ, but that isn't what we want here).

However, we can define an existential type for a variable ψ as $\emptyset; \psi \vdash (\mathsf{Lan}^{\overline{\alpha}}_{\emptyset}\ \psi\ \overline{\alpha})$, or given some type $\Gamma; \Phi, \overline{\gamma} \vdash F$ we can existentially quantify $\overline{\gamma}$ in F with $\Gamma; \Phi \vdash (\mathsf{Lan}^{\overline{\alpha}}_{\emptyset}\ F[\overline{\gamma := \alpha}])$.

The introduction and elimination rules for Lan give these types their "existential" character:

$$\frac{\Gamma; \Phi, \overline{\alpha} \vdash F \qquad \overline{\Gamma; \Phi \vdash A} \qquad \Gamma; \Phi \mid \Delta \vdash t : F[\overline{\alpha := A}]}{\Gamma; \Phi \mid \Delta \vdash \int_{\emptyset, F} t : (\mathsf{Lan}^{\overline{\alpha}}_{\emptyset}\ F)}$$

$$\frac{\Gamma; \emptyset \mid \Delta \vdash \eta : \mathsf{Nat}^{\Phi, \overline{\alpha}}\ F\ G \qquad \Gamma; \Phi \mid \Delta \vdash t : (\mathsf{Lan}^{\overline{\alpha}}_{\emptyset} F)}{\Gamma; \Phi \mid \Delta \vdash \partial^{G, \emptyset}_{F} \eta\ t : G}$$

The introduction rule is precisely as we expect: to introduce an existential type, we just need a term of type $F[\overline{\alpha := A}]$ for some types $\overline{A}$. The elimination rule also behaves as expected: if we have a term t of existential type and a natural transformation from F to G, we don't need to know which instance of F produced t in order to reach G, because the natural transformation is defined for all instances of F. For example, if we have a list whose elements are of an unknown type, we can still compute its length given a natural transformation of type $\mathsf{Nat}^{\alpha}(\mathit{List}\,\alpha)(\mathbb{N})$.

3.1 From List to Vec (PTree to GTree)

In order to write a natural transformation from lists to length-indexed vectors, we must existentially quantify the length variable in the return type. Otherwise, we must define the natural transformation for all possible values of n. The function is defined as follows:

I am having issues defining the list to vector function at the moment, but I will define an analogous embedding function for *PTree* and *GTree*:

```
data PTree (A : Set) : Set where
      pleaf : A → PTree A
      pnode : PTree (A × A) → PTree A
```

$$\mathit{PTree}\,\beta = (\mu\phi.\lambda\alpha.\alpha + \phi\,(\alpha \times \alpha))\beta$$

```
data GTree (A : Set) (N : Set) : Set where
      gleaf : A → GTree A Z
      gnode : GTree A N → GTree A N → GTree A (S N)
```

$$\mathit{GTree}\,\beta_1\,\beta_2 = (\mu\phi.\lambda\alpha_1, \alpha_2.(\mathsf{Lan}^{\gamma}_{\gamma, Z}\ \gamma)(\alpha_1, \alpha_2) + (\mathsf{Lan}^{\gamma_1, \gamma_2}_{\gamma, S\,\gamma_2}\ \phi\,(\gamma_1, \gamma_2) \times \phi\,(\gamma_1, \gamma_2))(\alpha_1, \alpha_2))(\beta_1, \beta_2)$$

We first define some constructors for *GTree* and some helper functions for *ptreeToGTree*. There is some trickiness with variable names in the Lan types, and so we define *alphaRename* to deal with this.

The origin of this issue is in the typing of the Lan-elimination rule used in the second case of *alg* below. In particular, it does not seem possible to let the codomain of η be typed as $\emptyset; \alpha, n \vdash (\mathsf{Lan}^{n}_{\emptyset}\ \mathit{GTree}\,\alpha\,n)$, even though it is legal to form $\emptyset; \alpha \vdash (\mathsf{Lan}^{n}_{\emptyset}\ \mathit{GTree}\,\alpha\,n)$ and then weaken to get $\emptyset; \alpha, n \vdash (\mathsf{Lan}^{n}_{\emptyset}\ \mathit{GTree}\,\alpha\,n)$. The issue is that a natural transformation of type $\mathsf{Nat}^{\alpha, n}\ (\mathit{GTree}\,(\alpha \times \alpha)\,n)\ (\mathsf{Lan}^{n}_{\emptyset}\ \mathit{GTree}\,\alpha\,n)$ apparently cannot be defined directly. To define such a natural transformation, we would need to change *helper*'s type and definition:

$$\mathit{helper} : \mathsf{Nat}^{\alpha, n}\ (\mathit{GTree}\,(\alpha \times \alpha)\,n)\ (\mathsf{Lan}^{n}{}_{\emptyset}\ \mathit{GTree}\,\alpha\,n)$$

$$helper = L_{\alpha,n}\, t.\ \int_{\emptyset, GTree\ \alpha\ n} (gnode_{\alpha,n}\ (unzipGTree_{\alpha,n}\ t))$$

It turns out that the use of $\int_{\emptyset, GTree\ \alpha\ n}$ is not well-typed in this version (specifically because it requires n to be in both Φ and $\overline{\alpha}$ in the context of F in the Lan-introduction rule), and so we must define *alphaRename*.

$$gleaf : \mathsf{Nat}^{\alpha}\ (\alpha)\ (GTree\ \alpha\ Z)$$
$$gleaf = L_{\alpha}\ t.(\mathsf{in}_{GTree})_{\alpha,Z}\ (\mathsf{inl}\ (\int_{(\gamma,Z),\gamma}\ t))$$

$$gnode : \mathsf{Nat}^{\alpha,n}\ (GTree\ \alpha\ n \times GTree\ \alpha\ n)\ (GTree\ \alpha\ (S\ n))$$
$$gnode = L_{\alpha,n}\ t.(\mathsf{in}_{GTree})_{\alpha,(S\ n)}\ (\mathsf{inr}\ (\int_{(\gamma_1, S\ \gamma_2), GTree\ \gamma_1\ \gamma_2 \times GTree\ \gamma_1\ \gamma_2}\ t))$$

$$ptreeToGTree : \mathsf{Nat}^{\alpha}\ (PTree\ \alpha)\ (\mathsf{Lan}^{n}_{\emptyset}\ GTree\ \alpha\ n)$$
$$ptreeToGTree = (\mathsf{fold}_{\beta+\phi\ (\beta\times\beta)})_{\emptyset}\ alg$$

$$alg : \mathsf{Nat}^{\alpha}(\alpha + (\mathsf{Lan}^{n}_{\emptyset}\ GTree\ (\alpha \times \alpha)\ n))\ (\mathsf{Lan}^{n}_{\emptyset}\ GTree\ \alpha\ n)$$
$$alg = L_{\alpha}\ s.\mathsf{case}\ s\ \mathsf{of}\ \{$$
$$\quad lf : \alpha \mapsto \int_{\emptyset, GTree\ \alpha\ n}(gleaf_{\ \alpha}\ lf);$$
$$\quad nd : (\mathsf{Lan}^{n}_{\emptyset}\ GTree\ (\alpha \times \alpha)\ n) \mapsto alphaRename_{GTree\ \alpha\ \beta}\ (\partial^{(\mathsf{Lan}^{m}_{\emptyset}\ GTree\ \alpha\ m),\emptyset}_{GTree\ (\alpha\times\alpha)\ n}\ helper\ t)$$
$$\}$$

$$helper : \mathsf{Nat}^{\alpha,n}\ (GTree\ (\alpha \times \alpha)\ n)\ (\mathsf{Lan}^{m}_{\emptyset}\ GTree\ \alpha\ m)$$
$$helper = L_{\alpha,n}\ t.\ \int_{\emptyset, GTree\ \alpha\ m}\ (gnode_{\alpha,n}\ (unzipGTree_{\alpha,n}\ t))$$

$$unzipGTree : \mathsf{Nat}^{\alpha,n}\ (GTree\ (\alpha \times \alpha)\ n)\ (GTree\ \alpha\ n \times GTree\ \alpha\ n)$$
$$unzipGTree = L_{\alpha,n}\ t.(((\mathsf{map}^{\alpha\times\alpha,\alpha}_{GTree\ \beta\ n})_{\emptyset}\ (L_{\alpha,n} x.\pi_1 x))_{\alpha,n}\ t,$$
$$\qquad ((\mathsf{map}^{\alpha\times\alpha,\alpha}_{GTree\ \beta\ n})_{\emptyset}\ (L_{\alpha,n} x.\pi_2 x))_{\alpha,n}\ t)$$

$$existsGTree : \mathsf{Nat}^{\alpha,n}(GTree\ \alpha\ n)\ (\mathsf{Lan}^{m}_{\emptyset}\ GTree\ \alpha\ m)$$
$$existsGTree = L_{\alpha,n}\ t.\ \int_{\emptyset, GTree\ \alpha\ m}\ t$$

$$alphaRename : \mathsf{Nat}^{\phi}(\mathsf{Lan}^{m}_{\emptyset}\ \phi\ m)\ (\mathsf{Lan}^{n}_{\emptyset}\ \phi\ n)$$
$$alphaRename = L_{\phi}\ t.\partial^{(\mathsf{Lan}^{n}_{\emptyset}\ \phi\ n),\emptyset}_{(\phi\ m)}\ exists\ t$$

$$exists : \mathsf{Nat}^{\phi,m}(\phi\ m)\ (\mathsf{Lan}^{n}_{\emptyset}\ \phi\ n)$$
$$exists = L_{\phi,m}\ t.\int_{\emptyset,\phi\ n}\ t$$

Let $\emptyset; \alpha \vdash G = (\mathsf{Lan}^{n}_{\emptyset}\ GTree\ \alpha\ n)$. Is this allowed?

Typing of Lan-elimination in *nd* case of *alg*

$$\frac{\emptyset;\emptyset \mid \Delta \vdash \eta : \mathsf{Nat}^{\alpha,n}\ (GTree\,(\alpha \times \alpha)\,n)\ (\mathsf{Lan}^{m}_{\emptyset}\ GTree\,\alpha\,m) \qquad \emptyset;\alpha \mid \Delta \vdash t : (\mathsf{Lan}^{n}_{\emptyset}\ GTree\,(\alpha \times \alpha)\,n)}{\emptyset;\alpha \mid \Delta \vdash \partial^{(\mathsf{Lan}^{m}_{\emptyset}\ GTree\,\alpha\,m),\emptyset}_{GTree\,(\alpha\times\alpha)\,n}\,\eta\,t : (\mathsf{Lan}^{m}_{\emptyset}\ GTree\,\alpha\,m)}$$

Typing of Lan-elimination in *alphaRename*

$$\frac{\emptyset;\emptyset \mid t \vdash \eta : \mathsf{Nat}^{\phi,m}\ (\phi\,m)\ (\mathsf{Lan}^{n}_{\emptyset}\,\phi\,n) \qquad \emptyset;\phi \mid t \vdash t : (\mathsf{Lan}^{m}_{\emptyset}(\phi\,m))}{\emptyset;\phi \mid t \vdash \partial^{(\mathsf{Lan}^{n}_{\emptyset}\,\phi\,n),\emptyset}_{(\phi\,m)}\,\eta\,t : (\mathsf{Lan}^{n}_{\emptyset}\,\phi\,n)}$$

Typing of Lan-introduction in *exists*

$$\frac{\emptyset;\phi,m,n \vdash \phi\,n \qquad \emptyset;\phi,m \vdash m \qquad \emptyset;\phi,m \mid t \vdash t : \phi n[n := m]}{\emptyset;\phi,m \mid t \vdash \int_{\emptyset,\phi\,n} t : (\mathsf{Lan}^{n}_{\emptyset}\phi\,n)}$$

3.2 Varying K in Existential Types

We can also represent existential types with $(\mathsf{Lan}^{\overline{\alpha}}_{\mathbb{1}}\,F)\mathbb{1}$, or by using any other constant functor as a subscript. This is because using a constant functor in the subscript restricts the introduction rule so that the instantiating types $\overline{A}$ are hidden in the resulting Lan-type.

Lan-introduction rule with constant subscript

$$\frac{\Gamma;\Phi,\overline{\alpha} \vdash F \qquad \overline{\Gamma;\overline{\alpha} \vdash K} \qquad \overline{\Gamma;\Phi \vdash A} \qquad \Gamma;\Phi \mid \Delta \vdash t : F[\overline{\alpha := A}]}{\Gamma;\Phi \mid \Delta \vdash \int_{\overline{K},F} t : (\mathsf{Lan}^{\overline{\alpha}}_{\overline{K}}F)\,\overline{K}}$$

Varying K changes the elimination rule as well. In particular, the types $\overline{B}$ must be $\overline{K}$,

Lan-elimination rule with constant subscript

$$\frac{\Gamma;\emptyset \mid \Delta \vdash \eta : \mathsf{Nat}^{\Phi,\overline{\alpha}}\,F\,G[\overline{\beta := K}] \qquad \Gamma;\Phi \mid \Delta \vdash t : (\mathsf{Lan}^{\overline{\alpha}}_{\overline{K}}F)\,\overline{K}}{\Gamma;\Phi \mid \Delta \vdash \partial^{G,\overline{K}}_{F}\eta\,t : G[\overline{\beta := K}]}$$

The most general existential type is $(\mathsf{Lan}^{\overline{\alpha}}_{\emptyset}\,F)$, or equivalently $(\mathsf{Lan}^{\overline{\alpha}}_{\mathbb{1}}\,F)\mathbb{1}$, but if we change the subscript to $\mathbb{N}$, for example, then we are somewhat restricted in what we can compute with this type due to the elimination rule.

4 EQUALITY TYPES

A common use of GADTs is to define a datatype representing type equality:

```
data Eql (A : Set) (B : Set) : Set where
  refl : Eql A A
```

We can define this in our calculus as $Eql\,\beta_1\,\beta_2 = (\mathsf{Lan}^{\gamma}_{\gamma,\gamma}1)(\beta_1,\beta_2)$. Looking at the Lan-introduction rule tells us that we can only produce a term of this type when β_1 and β_2 are the same type:

$$\frac{\Gamma;\Phi,\gamma \vdash \mathbb{1} \qquad \overline{\Gamma;\gamma \vdash \gamma} \qquad \Gamma;\Phi \vdash A \qquad \vdash t : \mathbb{1}}{\Gamma;\Phi \mid \emptyset \vdash \int_{\overline{K},F} t : (\mathsf{Lan}^{\gamma}_{\gamma,\gamma}\,\mathbb{1})\,(A,A)}$$

We can use the elimination rule to show that this equality is symmetric, but so far, I have not found many useful applications of this type. It seems these types are mostly useful in practical settings with type refinement/unification.

Generic Lan-**elimination rule for Eql**

$$\frac{\Gamma;\emptyset \mid \Delta \vdash \eta : \mathsf{Nat}^{\Phi,\gamma}\ \mathbb{1}\ G[\beta_1 := \gamma][\beta_2 := \gamma] \qquad \Gamma;\Phi \vdash A \qquad \Gamma;\Phi \vdash B \qquad \Gamma;\Phi \mid \Delta \vdash t : (\mathsf{Lan}^{\gamma}_{\gamma,\gamma}\mathbb{1})\,(A,B)}{\Gamma;\Phi \mid \Delta \vdash \partial_F^{G,\overline{K}}\eta\, t : G[\beta_1 := A][\beta_2 := B]}$$

Lan-**elimination rule for Eql with G =** $(Eql\,\beta_2\,\beta_1)$

$$\frac{\Gamma;\emptyset \mid \Delta \vdash \eta : \mathsf{Nat}^{\Phi,\gamma}\ \mathbb{1}\ (Eql\,\gamma\,\gamma) \qquad \Gamma;\Phi \vdash A \qquad \Gamma;\Phi \vdash B \qquad \Gamma;\Phi \mid \Delta \vdash t : Eql\,A\,B}{\Gamma;\Phi \mid \Delta \vdash \partial_F^{G,\overline{K}}\eta\, t : Eql\,B\,A}$$

We can also use the *Eql* type to define GADTs, e.g., an analogue of *GTree*:

```
data ETree (A : Set) (N : Set) : Set where
      eleaf : A → Eql N Z → ETree A N
      enode : Eql N (S M) → ETree A M → ETree A M → ETree A N
```

$$ETree\,\beta_1\,\beta_2 = (\mu\phi.\lambda\alpha, n.\alpha \times (\mathsf{Lan}^{\gamma}_{\gamma,\gamma}\mathbb{1})(n, Z) + (\mathsf{Lan}^{\alpha',n',m}_{\alpha',n'}(\mathsf{Lan}^{\gamma}_{\gamma,\gamma}\mathbb{1})(n', S\,m) \times \phi\,(\alpha', m) \times \phi\,(\alpha', m))(\alpha, n))(\beta_1, \beta_2)$$

I have not yet explored *ETree* or similar types thoroughly.