Some constructions on ω-groupoids

Thorsten Altenkirch
Computer Science
University of Nottingham
txa@cs.nott.ac.uk

Nuo Li
Computer Science
University of Nottingham
nzl@cs.nott.ac.uk

Ondřej Rypáček
University of Oxford
United Kingdom
ondrej.rypacek@gmail.com

ABSTRACT

Weak ω-groupoids are the higher dimensional generalisation of setoids and are an essential ingredient of the constructive semantics of Homotopy Type Theory [10]. Following up on our previous formalisation [3] and Brunerie's notes [5], we present a new formalisation of the syntax of weak ω-groupoids in Agda using heterogeneous equality. We show how to recover basic constructions on ω-groupoids using suspension and replacement. In particular we show that any type forms a groupoid and we outline how to derive higher dimensional composition. We present a possible semantics using globular sets and discuss the issues which arise when using globular types instead.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Lambda calculus and related systems, Mechanical theorem proving

General Terms

Theory

Keywords

Type Theory, Homotopy Type Theory, Category Theory, Formalisation, Higher dimensional structures, Agda

1. INTRODUCTION

In Type Theory, a type can be interpreted as a setoid which is a set equipped with an equivalence relation [1]. The equivalence proof of the relation consists of reflexivity, symmetry and transitivity whose proofs are unique. However in Homotopy Type Theory, we reject the principle of uniqueness of identity proofs (UIP). Instead we accept the univalence axiom which says that equality of types is weakly equivalent to weak equivalence. Weak equivalence can be seen as a refinement of isomorphism without UIP [3]. For example, a weak equivalence between two objects A and B in a 2-category is a morphism $f : A \to B$ which has a corresponding inverse morphism $g : B \to A$, but instead of the proofs of isomorphism $f \circ g = 1_B$ and $g \circ f = 1_A$ we have two 2-cell isomorphisms $f \circ g \cong 1_B$ and $g \circ f \cong 1_A$.

Voevodsky proposed the univalence axiom which basically says that isomorphic types are equal. This can be viewed as a strong extensionality axiom and it does imply functional extensionality. However, adding univalence as an axiom destroys canonicity, i.e. that every closed term of type $\mathbb{N}$ is reducible to a numeral. In the special case of extensionality and assuming a strong version of UIP we were able to eliminate this issue [1, 2] using setoids. However, it is not clear how to generalize this in the absence of UIP to univalence which is incompatible with UIP. To solve the problem we should generalise the notion of setoids, namely to enrich the structure of the identity proofs.

The generalised notion is called weak ω-groupoids and was proposed by Grothendieck 1983 in a famous manuscript *Pursuing Stacks* [7]. Maltsiniotis continued his work and suggested a simplification of the original definition which can be found in [9]. Later Ara also presents a slight variation of the simplication of weak ω-groupoids in [4]. Categorically speaking an ω-groupoid is an ω-category in which morphisms on all levels are equivalences. As we know that a set can be seen as a discrete category, a setoid is a category where every morphism is unique between two objects. A groupoid is more generalised, every morphism is isomorphism but the proof of isomorphism is unique, namely the composition of a morphism with its inverse is equal to an identity morphism. Similarly, an n-groupoid is an n-category in which morphisms on all levels are equivalence. ω-groupoids which are also called ∞-groupoids is an infinite version of n-groupoids. To model Type Theory without UIP we also require the equalities to be non-strict, in other words, they are not definitionally equalities. Finally we should use weak ω-groupoids to interpret types and eliminate the univalence axiom.

There are several approaches to formalise weak ω-groupoids in Type Theory. For instance, Altenkirch and Rypáček [3], and Brunerie's notes [5]. This paper explains an implementation of weak ω-groupoids following Brunerie's approach in Agda which is a well-known theorem prover and also a variant of intensional Martin-Löf type theory. This is the first attempt to formalise this approach in a dependently typed language like Agda or Coq. The approach is to specify when a globular set is a weak ω-groupoid by first defining a type theory called $\mathcal{T}_{\infty-groupoid}$ to describe the internal language of Grothendieck weak ω-groupoids, then interpret it with a globular set and a dependent function. All coherence laws of the weak ω-groupoids are derivable from the syntax, we will present some basic ones, for example reflexivity. One of the main contributions of this paper is to use heterogeneous equality for terms to overcome some difficult problems we encountering when using the normal homogeneous one. In this paper, we omit some complicated and less important programs, namely the proofs of some lemmas or definitions of some auxiliary functions. It is still possible for the reader who is interested in the details to check the code online[1]

Agda

Agda is a programming language and development environment based on Martin-Löf Type Theory [11]. Readers with background in Type Theory (e.g. from reading the introductory chapters of [10]) should find it easy to read the Agda code presented in this paper. Some hints: Π-types are written in a generalized arrow notation $(x : A) \to B$ for $\Pi x : A.B$, implicit arguments are indicated by curly brackets, eg. $\{x : A\} \to B$, in this case the Agda will try to generate the argument automatically and we don't supply it to make the code more readable. If we don't want to supply A because it can be inferred we write $\forall x$ or $\forall\{x\}$. Agda uses a flexible mixfix notation where the position of arguments are indicated by underline characters. e.g. $\_ \Rightarrow \_$ is one identifier which can be applied to two arguments as in $A \Rightarrow B$. The underlined characters can also be used as wildcards, if something can be automoatically inferred by Agda. We use data **data** to define constructor based datatypes (both inductive and coinductive) and **record** to define dependent record types (this generalizes Σ-types. The representation of coinductive types and more generally mixed inductive/coinductive types [6] uses the type constructor ∞ whose elements are computations of type A which are written as $\sharp a$ where |a| is an expression which can be evaluated to an element of type A.

Acknowledgements

The first and second author would like to thank the organizers and other participants of the special year on homotopy type theory at the Institute for Advanced Study where they had many interesting discussion topics related to the work presented in this paper. Especially we all would like to acknowledge Guillaume Brunerie's proposal which made this work possible.

2. SYNTAX OF WEAK ω-GROUPOIDS

We develop the type theory of ω-groupoids formally, following [5]. This is a Type Theory with only one type former which we can view as equality types and interpret as the homsets of the ω-groupoid. There are no definitional equalities, this corresponds to the fact that we consider weak ω-groupoids. None of the groupoid laws on any levels are strict (i.e. definitional) but all are witnessed by terms. Compared to [3] the definition is very much simplified by the observation that all laws of a weak ω-groupoid follow from the existence of coherence constants for any contractible context.

[1]The source code is available on `github.com/nzl-nott`.

In our formalisation we exploit the more liberal way to do mutual definitions in Agda, which was implemented recently following up a suggestion by the first author. It allows us to first introduce a type former but give its definition later.

Since we are avoiding definitional equalities we have to define a syntactic substitution operation which we need for the general statement of the coherence constants. However, defining this constant requires us to prove a number of substitution laws at the same time. We address this issue by using a heterogeneous equality which exploits UIP. Note that UIP holds for the syntax because all components defined here are sets in the sense of Homotopy Type Theory.

2.1 Basic Objects

We first declare the syntax of our type theory which is called $\mathcal{T}_{\infty-groupoid}$ namely the internal language of weak ω-groupoids. Since the definitions of syntactic objects involve each others, it is essential to define them in a inductive-inductive way. Agda allows us to state the types and constructors separately for involved inductive-inductive definitions. The following declarations in order are contexts as sets, types are sets dependent on contexts, terms and variables are sets dependent on types, context morphisms and contractible contexts.

```
data Con          : Set
data Ty (Γ : Con) : Set
data Tm           : {Γ : Con}(A : Ty Γ) → Set
data Var          : {Γ : Con}(A : Ty Γ) → Set
data _⇒_          : Con → Con → Set
data isContr      : Con → Set
```

Contexts are inductively defined as either an empty context or a context with a type in it.

```
data Con where
  ε   : Con
  _,_ : (Γ : Con)(A : Ty Γ) → Con
```

Types are defined as either $*$ which we call 0-cells, or a equality type between two terms of some type A. If the type A is an n-cell then we call its equality type an $(n+1)$-cell.

```
data Ty Γ where
  *     : Ty Γ
  _=h_  : {A : Ty Γ}(a b : Tm A) → Ty Γ
```

2.2 Heterogeneous Equality for Terms

One of the big challenges we encountered at first is the difficulty to formalise and reason about the equalities of terms, which is essential when defining substitution. When the usual homogeneous identity types are used, one has to use substitution to unify the types on both sides of equality types. This results in *subst* to appear in terms, about which one has to state substitution lemmas. This further pollutes syntax requiring lemmas about lemmas, lemmas about lemmas about lemmas, etc. For example, we have to prove using *subst* consecutively with two equalities of types is propositionally equal to using *subst* with the composition of these two equalities. There are more and more lemmas needed as the complexity of the proofs grows. The resulting recurring pattern has been identified and implemented in [3] for the special cases of coherence cells for associativity, units and interchange. However it is not clear how that approach could be adapted to the present, much more economical formulation of weak ω-groupoids. Moreover, the complexity brings the Agda type checker to its limits and correctness into question.

The idea of heterogenous equality, which we use to resolve this issue, is that one can define equality for terms of different types, but its inhabitants only for terms of definitionally equal types. However, the corresponding elimination principle relies on UIP. In Intensional Type Theory, UIP is not provable in general, namely not all types are h-sets (homotopy 0-types). However every type with decidable equality is an h-set. From Hedberg's Theorem [8] we know that inductive types with finitary constructors have decidable equality. In our case, the types which stand for syntactic objects (contexts, types, terms) are all inductive-inductive types with finitary constructors and it is therefore safe to assume that UIP holds for them. In summary, the equality of syntactic types is unique, so it is safe to use heterogeneous equality and proceed without using substitution lemmas which would otherwise be necessary to match terms of different types. From a computational perspective, it means that every equality of types can be reduced to *refl* and using *subst* to construct terms is proof-irrelevant, which is expressed in the definition of heterogeneous equality for terms.

```
data _≅_ {Γ : Con}{A : Ty Γ} :
     {B : Ty Γ} → Tm A → Tm B → Set where
  refl : (b : Tm A) → b ≅ b
```

Once we have heterogeneous equality for terms, we can define a proof-irrelevant substitution which we call coercion since it gives us a term of type A if we have a term of type B and the two types are equal. We can also prove that the coerced term is heterogeneously equal to the original term. Combining these definitions, it is much more convenient to formalise and reason about term equations.

```
_⟦_⟫         : {Γ : Con}{A B : Ty Γ}(a : Tm B)
             → A ≡ B → Tm A
a ⟦ refl ⟫   = a

cohOp        : {Γ : Con}{A B : Ty Γ}{a : Tm B}(p : A ≡ B)
             → a ⟦ p ⟫ ≅ a
cohOp refl   = refl _
```

2.3 Substitutions

In this paper we usually define a set of functions together and we name a function x as xC for contexts, xT for types, xV for variables xtm for terms and xS (or xcm) for context morphisms. For example the substitutions are declared as follows:

```
_[_]T    : ∀{Γ Δ} → Ty Δ → Γ ⇒ Δ → Ty Γ
_[_]V    : ∀{Γ Δ A} → Var A → (δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
_[_]tm   : ∀{Γ Δ A} → Tm A → (δ : Γ ⇒ Δ) → Tm (A [ δ ]T)
```

Indeed, composition of context morphisms can be understood as substitution for context morphisms as well.

```
_⊚_ : ∀{Γ Δ Θ} → Δ ⇒ Θ → (δ : Γ ⇒ Δ) → Γ ⇒ Θ
```

Context morphisms are defined inductively similarly to contexts. A context morphism is a list of terms corresponding to the list of types in the context on the right hand side of the morphism.

```
data _⇒_ where
  •     : ∀{Γ} → Γ ⇒ ε
  _,_   : ∀{Γ Δ}(δ : Γ ⇒ Δ){A : Ty Δ}(a : Tm (A [ δ ]T))
        → Γ ⇒ (Δ , A)
```

2.4 Weakening

We can freely add types to the contexts of any given type judgments, term judgments or context morphisms. These are the weakening rules.

```
_+T_    : ∀{Γ}(A : Ty Γ)(B : Ty Γ) → Ty (Γ , B)
_+tm_   : ∀{Γ A}(a : Tm A)(B : Ty Γ) → Tm (A +T B)
_+S_    : ∀{Γ Δ}(δ : Γ ⇒ Δ)(B : Ty Γ) → (Γ , B) ⇒ Δ
```

2.5 Terms

A term can be either a variable or a coherence constant (coh).

We first define variables separately using the weakening rules. We use typed de Bruijn indices to define variables as either the rightmost variable of the context, or some variable in the context which can be found by cancelling the rightmost variable along with each vS.

```
data Var where
  v0 : ∀{Γ}{A : Ty Γ}       → Var (A +T A)
  vS : ∀{Γ}{A B : Ty Γ}(x : Var A) → Var (A +T B)
```

The coherence constants are one of the major part of this syntax, which are primitive terms of the primitive types in contractible contexts which will be introduced later. Indeed it encodes the fact that any type in a contractible context is inhabited, and so are the types generated by substituting into a contractible context.

```
data Tm where
  var   : ∀{Γ}{A : Ty Γ} → Var A → Tm A
  coh   : ∀{Γ Δ} → isContr Δ → (δ : Γ ⇒ Δ)
        → (A : Ty Δ) → Tm (A [ δ ]T)
```

2.6 Contractible contexts

With variables defined, it is possible to formalise another core part of the syntactic framework, *contractible contexts*. Intuitively speaking, a context is contractible if its geometric realization is contractible to a point. It either contains one variable of the 0-cell $*$ which is the base case, or we can extend a contractible context with a variable of an existing type and an n-cell, namely a morphism, between the new variable and some existing variable. The graph can be drawn like branching trees.

```
data isContr where
  c*    : isContr (ε , *)
  ext   : ∀{Γ} → isContr Γ → {A : Ty Γ}(x : Var A)
        → isContr (Γ , A , (var (vS x) =h var v0))
```

2.7 Lemmas

Since contexts, types, variables and terms are all mutually defined, most of their properties have to be proved simultaneously.

The following lemmas are essential for the constructions and theorem proving later. The first set of lemmas states that to substitute a type, a variable, a term, or a context morphism with two context morphisms consecutively, is equivalent to substitute with the composition of the two context morphisms:

```
[⊚]T      : ∀{Γ Δ Θ A}{ϑ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
          → A [ ϑ ⊚ δ ]T ≡ (A [ ϑ ]T)[ δ ]T

[⊚]v      : ∀{Γ Δ Θ A}(x : Var A){ϑ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
          → x [ ϑ ⊚ δ ]V ≅ (x [ ϑ ]V) [ δ ]tm

[⊚]tm     : ∀{Γ Δ Θ A}(a : Tm A){ϑ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
          → a [ ϑ ⊚ δ ]tm ≅ (a [ ϑ ]tm) [ δ ]tm

⊚assoc    : ∀{Γ Δ Θ Ω}(γ : Θ ⇒ Ω){ϑ : Δ ⇒ Θ}{δ : Γ ⇒ Δ}
          → (γ ⊚ ϑ) ⊚ δ ≡ γ ⊚ (ϑ ⊚ δ)
```

The second set states that weakening inside substitution is equivalent to weakening outside:

```
[+S]T     : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}
          → A [ δ +S B ]T ≡ (A [ δ ]T) +T B

[+S]tm    : ∀{Γ Δ A B}(a : Tm A){δ : Γ ⇒ Δ}
          → a [ δ +S B ]tm ≅ (a [ δ ]tm) +tm B

[+S]S     : ∀{Γ Δ Θ B}{δ : Δ ⇒ Θ}{γ : Γ ⇒ Δ}
          → δ ⊚ (γ +S B) ≡ (δ ⊚ γ) +S B
```

We can cancel the last term in the substitution for weakened objects since weakening doesn't introduce new variables in types and terms.

```
+T[,]T      : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}{b : Tm (B [ δ ]T)}
            → (A +T B) [ δ , b ]T ≡ A [ δ ]T

+tm[,]tm    : ∀{Γ Δ A B}{δ : Γ ⇒ Δ}{c : Tm (B [ δ ]T)}
            → (a : Tm A)
            → (a +tm B) [ δ , c ]tm ≅ a [ δ ]tm
```

Most of the substitutions are defined as usual, except the one for coherence constants. In this case, we substitute in the context morphism part and one of the lemmas declared above is used.

```
var x          [ δ ]tm = x [ δ ]V
coh cΔ γ A     [ δ ]tm = coh cΔ (γ ⊚ δ) A ⟦ sym [⊚]T ⟫
```

3. SOME IMPORTANT DERIVABLE CONSTRUCTIONS

In this section we show that it is possible to reconstruct the structure of a (weak) ω-groupoid from the syntactical framework presented in Section 2 in the style of [3]. To this end, let us call a term a : Tm A an n-cell if level $A \equiv n$, where

```
level                   : ∀ {Γ} → Ty Γ → ℕ
level *                 = 0
level (_=h_ {A} _ _)    = suc (level A)
```

In any ω-category, any n-cell a has a domain (source), $s^n_m\,a$, and a codomain (target), $t^n_m\,a$, for each $m \leq n$. These are, of course, (n-m)-cells. For each pair of n-cells such that for some m, $s^n_m a \equiv t^n_m b$, there must exist their composition $a \circ^n_m b$ which is an n-cell. Composition is (weakly) associative. Moreover for any (n-m)-cell x there exists an n-cell $\mathsf{id}^n_m\,x$ which behaves like a (weak) identity with respect to $\circ^n_m$. For the time being we discuss only the construction of cells and omit the question of coherence.

For instance, in the simple case of bicategories, each 2-cell a has a horizontal source $s^1_1\,a$ and target $t^1_1\,a$, and also a vertical source $s^2_1\,a$ and target $t^2_1 a$, which is also the source and target, of the horizontal source and target, respectively, of a.

There is horizontal composition of 1-cells $\circ^1_1$: $x \xrightarrow{f} y \xrightarrow{g} z$, and also horizontal composition of 2-cells $\circ^2_1$, and vertical composition of 2-cells $\circ^2_2$. There is a horizontal identity on a, $\mathsf{id}^1_1\,a$, and vertical identity on a, $\mathsf{id}^2_1\,a = \mathsf{id}^2_2\mathsf{id}^1_1\,a$.

Thus each ω-groupoid construction is defined with respect to a *level*, m, and depth n-m and the structure of an ω-groupoid is repeated on each level. As we are working purely syntactically we may make use of this fact and define all groupoid structure only at level $m = 1$ and provide a so-called *replacement operation* which allows us to lift any cell to an arbitrary type A. It is called 'replacement' because we are syntactically replacing the base type $*$ with an arbitrary type, A.

An important general mechanism we rely on throughout the development follows directly from the type of the only non-trivial constructor of Tm, coh, which tells us that to construct a new term of type $\Gamma \vdash A$, we need a contractible context, Δ, a type $\Delta \vdash T$ and a context morphism $\delta : \Gamma \Rightarrow \Delta$ such that

$$T\,[\,\delta\,]\mathsf{T} \equiv A$$

Because in a contractible context all types are inhabited we may in a way work freely in Δ and then pull back all terms to A using δ. To show this formally, we must first define identity context morphisms which complete the definition of a *category* of contexts and context morphisms:

```
IdCm : ∀{Γ} → Γ ⇒ Γ
```

It satisfies the following property:

```
IC-T : ∀{Γ}{A : Ty Γ} → A [ IdCm ]T ≡ A
```

The definition proceeds by structural recursion and therefore extends to terms, variables and context morphisms with analogous properties. It allows us to define at once:

```
Coh-Contr         : ∀{Γ}{A : Ty Γ} → isContr Γ → Tm A
Coh-Contr isC     = coh isC IdCm _ ⟦ sym IC-T ⟫
```

We use Coh-Contr as follows: for each kind of cell we want to define, we construct a minimal contractible context built out of variables together with a context morphism that populates the context with terms and a lemma that states an equality between the substitution and the original type.

3.1 Suspension and Replacement

For an arbitrary type A in Γ of level n one can define a context with $2n$ variables, called the *stalk* of A. Moreover one can define a morphism from Γ to the stalk of A such that its substitution into the maximal type in the stalk of A gives back A. The stalk of A depends only on the level of A, the terms in A define the substitution. Here is an example of stalks of small levels: ε (the empty context) for $n = 0$; $(x_0 : *, x_1 : *)$ for $n = 1$; $(x_0 : *, x_1 : *, x_2 : x_0 =_{\mathsf{h}}\ x_1, x_3 :$

$x_0 =_h x_1)$ for $n = 2$, etc.

This is the $\Delta = \varepsilon$ case of a more general construction where in we *suspend* an arbitrary context Δ by adding $2n$ variables to the beginning of it, and weakening the rest of the variables appropriately so that type $*$ becomes $x_{2n-2} =_h x_{2n-1}$. A crucial property of suspension is that it preserves contractibility.

3.1.1 Suspension

Suspension is defined by iteration level-A-times the following operation of one-level suspension. ΣC takes a context and gives a context with two new variables of type $*$ added at the beginning, and with all remaining types in the context suspended by one level.

```
ΣC : Con → Con
ΣT : ∀{Γ} → Ty Γ → Ty (ΣC Γ)

ΣC ε          = ε , * , *
ΣC (Γ , A)    = ΣC Γ , ΣT A
```

The rest of the definitions is straightforward by structural recursion. In particular we suspend variables, terms and context morphisms:

```
Σv    : ∀{Γ}{A : Ty Γ} → Var A → Var (ΣT A)
Σtm   : ∀{Γ}{A : Ty Γ} → Tm A → Tm (ΣT A)
Σs    : ∀{Γ Δ} → Γ ⇒ Δ → ΣC Γ ⇒ ΣC Δ
```

The following lemma establishes preservation of contractibility by one-step suspension:

```
ΣC-Contr : ∀ Δ → isContr Δ → isContr (ΣC Δ)
```

It is also essential that suspension respects weakening and substitution:

```
ΣT[+T]      : ∀{Γ}(A B : Ty Γ)
              → ΣT (A +T B) ≡ ΣT A +T ΣT B

Σtm[+tm] : ∀{Γ A}(a : Tm A)(B : Ty Γ)
              → Σtm (a +tm B) ≅ Σtm a +tm ΣT B

ΣT[Σs]T     : ∀{Γ Δ}(A : Ty Δ)(δ : Γ ⇒ Δ)
              → (ΣT A) [ Σs δ ]T ≡ ΣT (A [ δ ]T)
```

General suspension to the level of a type A is defined by iteration of one-level suspension. For symmetry and ease of reading the following suspension functions take as a parameter a type A in Γ, while they depend only on its level.

```
ΣC-it     : ∀{Γ}(A : Ty Γ) → Con → Con

ΣT-it     : ∀{Γ Δ}(A : Ty Γ) → Ty Δ → Ty (ΣC-it A Δ)

Σtm-it    : ∀{Γ Δ}(A : Ty Γ){B : Ty Δ} → Tm B
            → Tm (ΣT-it A B)
```

Finally, it is clear that iterated suspension preserves contractibility.

```
ΣC-it-Contr    : ∀ {Γ Δ}(A : Ty Γ) → isContr Δ
                 → isContr (ΣC-it A Δ)
```

By suspending the minimal contractible context, $*$, we obtain a so-called *span*. They are stalks with a top variable added. For example $(x_0 : *)$ (the one-variable context) for $n = 0$; $(x_0 : *, x_1 : *, x_2 : x_0 =_h x_1)$ for $n = 1$; $(x_0 : *, x_1 : *, x_2 : x_0 =_h x_1, x_3 : x_0 =_h x_1, x_4 : x_2 =_h x_3)$ for $n = 2$, etc. Spans play an important role later in the definition of composition.

3.1.2 Replacement

After we have suspended a context by inserting an appropriate number of variables, we may proceed to a substitution which fills the stalk for A with A. The context morphism representing this substitution is called filter. In the final step we combine it with Γ, the context of A. The new context contains two parts, the first is the same as Γ, and the second is the suspended Δ substituted by filter. However, we also have to drop the stalk of A becuse it already exists in Γ.

This operation is called *replacement* because we can interpret it as replacing $*$ in Δ by A. Geometrically speaking, the resulting context is a new context which corresponds to the pasting of Δ to Γ at A.

As always, we define replacement for contexts, types and terms:

```
rpl-C     : ∀{Γ}(A : Ty Γ) → Con → Con
rpl-T     : ∀{Γ Δ}(A : Ty Γ) → Ty Δ → Ty (rpl-C A Δ)
rpl-tm    : ∀{Γ Δ}(A : Ty Γ){B : Ty Δ} → Tm B
            → Tm (rpl-T A B)
```

Replacement for contexts, rpl-C, defines for a type A in Γ and another context Δ a context which begins as Γ and follows by each type of Δ with $*$ replaced with (pasted onto) A.

```
rpl-C {Γ} A ε       = Γ
rpl-C A (Δ , B)     = rpl-C A Δ , rpl-T A B
```

To this end we must define the substitution filter which pulls back each type from suspended Δ to the new context.

```
filter    : ∀{Γ}(Δ : Con)(A : Ty Γ)
            → rpl-C A Δ ⇒ ΣC-it A Δ

rpl-T A B = ΣT-it A B [ filter _ A ]T
```

3.2 First-level Groupoid Structure

We can proceed to the definition of the groupoid structure of the syntax. We start with the base case: 1-cells. Replacement defined above allows us to lift this structure to an arbitrary level n (we leave most of the routine details out). This shows that the syntax is a 1-groupoid on each level. In the next section we show how also the higher-groupoid structure can be defined.

We start by an essential lemma which formalises the discussion at the beginning of this section: to construct a term in a type A in an arbitrary context, we first restrict attention to a suitable contractible context Δ and use lifting and substitution – replacement – to pull the term built by coh in Δ back. This relies on the fact that a lifted contractible context is also contractible, and therefore any type lifted from a contractible context is also inhabited.

```
Coh-rpl   : ∀{Γ Δ}(A : Ty Γ)(B : Ty Δ) → isContr Δ
            → Tm (rpl-T A B)
Coh-rpl {_} {Δ} A _ isC = coh (ΣC-it-ε-Contr A isC) _ _
```

Next we define the reflexivity, symmetry and transitivity terms of any type. Let's start from some base cases. Each of the base cases is derivable in a different contractible context with Coh-Contr which gives you a coherence constant for any type in any contractible context.

Reflexivity (identity) It only requires a one-object context.

```
refl*-Tm : Tm {x:*} (var v0 =h var v0)
refl*-Tm = Coh-Contr c*
```

Symmetry (inverse) It is defined similarly. Note that the intricate names of contexts, as in Ty x:*,y:*,α:x=y indicate their definitions which have been hidden. Recall that Agda treats all sequences of characters uninterrupted by whitespace as identifiers. For instance x:*,y:*,α:x=y is a name of a context for which we are assuming the definition: x:*,y:*,α:x=y = ε , * , * , (var (vS v0) =h var v0).

```
sym*-Ty : Ty x:*,y:*,α:x=y
sym*-Ty = vY =h vX

sym*-Tm : Tm {x:*,y:*,α:x=y} sym*-Ty
sym*-Tm = Coh-Contr (ext c* v0)
```

Transitivity (composition)

```
trans*-Ty : Ty x:*,y:*,α:x=y,z:*,β:y=z
trans*-Ty = (vX +tm _ +tm _) =h vZ

trans*-Tm : Tm trans*-Ty
trans*-Tm = Coh-Contr (ext (ext c* v0) (vS v0))
```

To obtain these terms for any given type in any give context, we use replacement.

```
refl-Tm     : {Γ : Con}(A : Ty Γ)
              → Tm (rpl-T {Δ = x:*} A (var v0 =h var v0))
refl-Tm A   = rpl-tm A refl*-Tm

sym-Tm : ∀ {Γ}(A : Ty Γ) → Tm (rpl-T A sym*-Ty)
sym-Tm A = rpl-tm A sym*-Tm

trans-Tm : ∀ {Γ}(A : Ty Γ) → Tm (rpl-T A trans*-Ty)
trans-Tm A = rpl-tm A trans*-Tm
```

For each of reflexivity, symmetry and transitivity we can construct appropriate coherence 2-cells witnessing the groupoid laws. The base case for variable contexts is proved simply using contractibility as well. However the types of these laws are not as trivial as the proving parts. We use substitution to define the application of the three basic terms we have defined above.

```
Tm-right-identity* : Tm {x:*,y:*,α:x=y}
    (trans*-Tm [ IdCm , vY , reflY ]tm =h vα)
Tm-right-identity* = Coh-Contr (ext c* v0)

Tm-left-identity* : Tm {x:*,y:*,α:x=y}
    (trans*-Tm [ ((IdCm ⊚ pr1 ⊚ pr1) , vX) ,
        reflX , vY , vα ]tm =h vα)
Tm-left-identity* = Coh-Contr (ext c* v0)

Tm-right-inverse* : Tm {x:*,y:*,α:x=y}
    (trans*-Tm [ (IdCm , vX) , sym*-Tm ]tm =h reflX)
Tm-right-inverse* = Coh-Contr (ext c* v0)

Tm-left-inverse* : Tm {x:*,y:*,α:x=y}
    (trans*-Tm [ ((• , vY) , vX , sym*-Tm , vY) , vα ]tm =h reflY)
Tm-left-inverse* = Coh-Contr (ext c* v0)

Tm-G-assoc* : Tm Ty-G-assoc*
Tm-G-assoc* = Coh-Contr (ext (ext (ext c* v0) (vS v0)) (vS v0))
```

Their general versions are defined using replacement. For instance, for associativity, we define:

```
Tm-G-assoc     : ∀{Γ}(A : Ty Γ) → Tm (rpl-T A Ty-G-assoc*)
Tm-G-assoc A   = rpl-tm A Tm-G-assoc*
```

Following the same pattern, the n-level groupoid laws can be obtained as the coherence constants as well.

3.3 Higher Structure

In the previous text we have shown how to define 1-groupoid structure on an arbitrary level. Here we indicate how all levels also bear the structure of n-groupoid for arbitrary n. The rough idea amounts to redefining telescopes of [3] in terms of appropriate contexts, which are contractible, and the different constructors for terms used in [3] in terms of coh.

To illustrate this we consider the simpler example of higher identities. Note that the domain and codomain of n+1-iterated identity are n-iterated identities. Hence we proceed by induction on n. Denote a span of depth n S_n. Then there is a chain of context morphisms $S_0 \Rightarrow S_1 \Rightarrow \cdots \Rightarrow S_n$. Each S_{n+1} has one additional variable standing for the identity iterated n+1-times. Because S_{n+1} is contractible, one can define a morphism $S_n \Rightarrow S_{n+1}$ using coh to fill the last variable and variable terms on the first n levels. By composition of the context morphisms one defines n new terms in the basic one variable context $*$ – the iterated identities. Using suspension one can lift the identities to an arbitrary level.

Each n-cell has n-compositions. In the case of 2-categories, 1-cells have one composition, 2-cells have vertical and horizontal composition. Two 2-cells are horizontally composable only if their 1-cell top and bottom boundaries are composable. The boundary of the composition is the composition of the boundaries. Thus for arbitrary n we proceed using a chain of V-shaped contractible contexts. That is contexts that are two spans conjoined at the base level at a common middle variable. Each successive composition is defined using contractibility and coh.

To fully imitate the development in [3], one would also have to define all higher coherence laws. But the sole purpose of giving an alternative type theory in this paper is to avoid that.

4. SEMANTICS

4.1 Globular Types

To interpret the syntax, we need globular types [2] . Globular types are defined coinductively as follows:

[2]Even though we use the Agda Set, this isn't necessarily a set in the sense of Homotopy Type Theory.

```
record Glob : Set₁ where
  constructor _||_
  field
    |_|   : Set
    hom   : |_| → |_| → ∞ Glob
```

If all the object types (|_|) are indeed sets, i.e. uniqueness of identity types holds, we call this a globular set.

As an example, we could construct the identity globular type called Idω.

```
Idω     : (A : Set) → Glob
Idω A   = A || (λ a b → ♯ Idω (a ≡ b))
```

Note that this is usually not a globular set.

Given a globular type G, we can interpret the syntactic objects.

```
record Semantic (G : Glob) : Set₁ where
  field
    ⟦_⟧C   : Con → Set
    ⟦_⟧T   : ∀{Γ} → Ty Γ → ⟦ Γ ⟧C → Glob
    ⟦_⟧tm : ∀{Γ A} → Tm A → (γ : ⟦ Γ ⟧C) → | ⟦ A ⟧T γ |
    ⟦_⟧cm : ∀{Γ Δ} → Γ ⇒ Δ → ⟦ Γ ⟧C → ⟦ Δ ⟧C
    π      : ∀{Γ A} → Var A → (γ : ⟦ Γ ⟧C) → | ⟦ A ⟧T γ |
```

π provides the projection of the semantic variable out of a semantic context.

Following are the computation laws for the interpretations of contexts and types.

```
⟦_⟧C-β1   : ⟦ ε ⟧C ≡ ⊤
⟦_⟧C-β2   : ∀ {Γ A} → ⟦ Γ , A ⟧C ≡
   Σ ⟦ Γ ⟧C (λ γ    → | ⟦ A ⟧T γ |)

⟦_⟧T-β1   : ∀{Γ}{γ : ⟦ Γ ⟧C} → ⟦ * ⟧T γ ≡ G
⟦_⟧T-β2   : ∀{Γ A u v}{γ : ⟦ Γ ⟧C}
          → ⟦ u =h v ⟧T γ ≡
            ♭ (hom (⟦ A ⟧T γ) (⟦ u ⟧tm γ) (⟦ v ⟧tm γ))
```

Semantic substitution and semantic weakening laws are also required. The semantic substitution properties are essential for dealing with substitutions inside interpretation,

```
semSb-T   : ∀ {Γ Δ}(A : Ty Δ)(δ : Γ ⇒ Δ)(γ : ⟦ Γ ⟧C)
          → ⟦ A [ δ ]T ⟧T γ ≡ ⟦ A ⟧T (⟦ δ ⟧cm γ)

semSb-tm : ∀{Γ Δ}{A : Ty Δ}(a : Tm A)(δ : Γ ⇒ Δ)
(γ : ⟦ Γ ⟧C)
          → subst |_| (semSb-T A δ γ) (⟦ a [ δ ]tm ⟧tm γ)
≡ ⟦ a ⟧tm (⟦ δ ⟧cm γ)

semSb-cm : ∀ {Γ Δ Θ}(γ : ⟦ Γ ⟧C)(δ : Γ ⇒ Δ)(ϑ : Δ ⇒ Θ)
          → ⟦ ϑ ⊚ δ ⟧cm γ ≡ ⟦ ϑ ⟧cm (⟦ δ ⟧cm γ)
```

Since the computation laws for the interpretations of terms and context morphisms are well typed up to these properties.

```
⟦_⟧tm-β1   : ∀{Γ A}{x : Var A}{γ : ⟦ Γ ⟧C}
           → ⟦ var x ⟧tm γ ≡ π x γ

⟦_⟧cm-β1   : ∀{Γ}{γ : ⟦ Γ ⟧C}
→ ⟦ • ⟧cm γ ≡ coerce ⟦_⟧C-β1 tt

⟦_⟧cm-β2   : ∀{Γ Δ}{A : Ty Δ}{δ : Γ ⇒ Δ}{γ : ⟦ Γ ⟧C}
{a : Tm (A [ δ ]T)} → ⟦ δ , a ⟧cm γ
            ≡ coerce ⟦_⟧C-β2 ((⟦ δ ⟧cm γ) ,
    subst |_| (semSb-T A δ γ) (⟦ a ⟧tm γ))
```

The semantic weakening properties should actually be deriavable since weakening is equivalent to projection substitution.

```
semWk-T   : ∀ {Γ A B}(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
          → ⟦ A +T B ⟧T (coerce ⟦_⟧C-β2 (γ , v)) ≡
            ⟦ A ⟧T γ

semWk-cm   : ∀ {Γ Δ B}{γ : ⟦ Γ ⟧C}{v : | ⟦ B ⟧T γ |}
           → (δ : Γ ⇒ Δ) → ⟦ δ +S B ⟧cm
              (coerce ⟦_⟧C-β2 (γ , v)) ≡ ⟦ δ ⟧cm γ


semWk-tm : ∀ {Γ A B}(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
         → (a : Tm A) → subst |_| (semWk-T γ v)
           (⟦ a +tm B ⟧tm (coerce ⟦_⟧C-β2 (γ , v)))
             ≡ (⟦ a ⟧tm γ)
```

Here we declare them as properties because they are essential for the computation laws of function π.

```
π-β1   : ∀{Γ A}(γ : ⟦ Γ ⟧C)(v : | ⟦ A ⟧T γ |)
       → subst |_| (semWk-T γ v)
         (π v0 (coerce ⟦_⟧C-β2 (γ , v))) ≡ v

π-β2   : ∀{Γ A B}(x : Var A)(γ : ⟦ Γ ⟧C)(v : | ⟦ B ⟧T γ |)
       → subst |_| (semWk-T γ v) (π (vS {Γ} {A} {B} x)
         (coerce ⟦_⟧C-β2 (γ , v))) ≡ π x γ
```

The only part of the semantics where we have any freedom is the interpretation of the coherence constants:

```
⟦coh⟧   : ∀{Θ} → isContr Θ → (A : Ty Θ)
        → (ϑ : ⟦ Θ ⟧C) → | ⟦ A ⟧T ϑ |
```

However, we also need to require that the coherence constants are well behaved wrt to substitution which in turn relies on the interpretation of all terms. To address this we state the required properties in a redundant form because the correctness for any other part of the syntax follows from the defining equations we have already stated. There seems to be no way to avoid this.

If the underlying globular type is not a globular set we need to add coherence laws, which is not very well understood. On the other hand, restricting ourselves to globular sets means that our prime examle Idω is not an instance anymore. We should still be able to construct non-trivial globular sets, e.g. by encoding basic topological notions and defining higher homotopies as in a classical framework. However, we don't currently know a simple definition of a globular set which is a weak ω-groupoid. One possibility would be to use the syntax of type theory with equality types. Indeed, we believe that this would be an alternative way to formalize weak ω-groupoids.

5. CONCLUSION

In this paper, we present an implementation of weak ω-groupoids following Brunerie's work. Briefly speaking, we define the syntax of the type theory $\mathcal{T}_{\infty-groupoid}$, then a weak ω-groupoid is a globular set with the interpretation of the syntax. To overcome some technical problems, we use heterogeneous equality for terms, some auxiliary functions and loop context in all implementation. We construct the identity morphisms and verify some groupoid laws in the syntactic framework. The suspensions for all sorts of objects are also defined for other later constructions.

There is still a lot of work to do within the syntactic framework. For instance, we would like to investigate the relation between the $\mathcal{T}_{\infty-groupoid}$ and a Type Theory with equality types and J eliminator which is called $\mathcal{T}_{eq}$. One direction is to simulate the J eliminator syntactically in $\mathcal{T}_{\infty-groupoid}$ as we mentioned before, the other direction is to derive J using *coh* if we can prove that the $\mathcal{T}_{eq}$ is a weak ω-groupoid. The syntax could be simplified by adopting categories with families. An alternative may be to use higher inductive types directly to formalize the syntax of type theory.

We would like to formalise a proof of that Idω is a weak ω-groupoid, but the base set in a globular set is an h-set which is incompatible with Idω. Perhaps we could solve the problem by instead proving a syntactic result, namely that the theory we have presented here and the theory of equality types with J are equivalence. Finally, to model the Type Theory with weak ω-groupoids and to eliminate the univalence axiom would be the most challenging task in the future.

6. REFERENCES

[1] Thorsten Altenkirch. Extensional Equality in Intensional Type Theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM.

[3] Thorsten Altenkirch and Ondrej Rypacek. A syntactical Approach to Weak ω-groupoids. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012*, 2012.

[4] D. Ara. On the homotopy theory of Grothendieck ∞-groupoids. *ArXiv e-prints*, June 2012.

[5] Guillaume Brunerie. Syntactic Grothendieck weak ∞-groupoids, 2013.

[6] Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively; an exercise in mixed induction and coinduction. In *proceedings of the Tenth International Conference on Mathematics of Program Construction (MPC 10)*, 2010.

[7] Alexander Grothendieck. Pursuing Stacks. 1983. Manuscript.

[8] Michael Hedberg. A coherence theorem for Martin-Löf's type theory. 1998.

[9] G. Maltsiniotis. Grothendieck ∞-groupoids, and still another definition of ∞-categories. *ArXiv e-prints*, September 2010.

[10] The Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics.* 2013.

[11] The Agda Wiki. Main page, 2010. [Online; accessed 13-April-2010].