The following full text is a preprint version which may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/161450

Please be advised that this information was generated on 2016-11-30 and may be subject to change.
Overview of the Talk

In this talk, we will develop a type theory that is based solely on dependent inductive and coinductive types. By this we mean that the only way to form new types is by specifying the type of their corresponding constructors or destructors, respectively. From such a specification, we get the corresponding recursion and corecursion principles. One might be tempted to think that such a theory is relatively weak as, for example, there is no function space type. However, as it turns out, the function space is definable as a coinductive type. In fact, we can encode the connectives of intuitionistic predicate logic: falsity, conjunction, disjunction, dependent function space, existential quantification, and equality. Further, well-known types like natural numbers, vectors etc. arise as well. The presented type theory is based on ideas from categorical logic that have been investigated before by the first author, and it extends Hagino’s categorical data types to a dependently typed setting. By basing the type theory on concepts from category theory we maintain the duality between inductive and coinductive types.

The reduction relation on terms consists solely of a rule for recursion and a rule for corecursion. We can then derive the usual computation rules for encoded types these basic rules. This results in a type theory with a small set of rules, while still being fairly expressive. To further support the introduction of this new type theory, we prove subject reduction and strong normalisation of the reduction relation.

Why do we need another type theory, especially since Martin-Löf type theory (MLTT) or the calculus of inductive constructions (CoIC) are well-studied frameworks for intuitionistic logic? The main reason is that the existing type theories have no explicit dependent coinductive types.

Other reasons for considering a new type theory are of foundational interest. First, taking inductive and coinductive types as core of the type theory reduces the number of deduction rules considerably. For each type former one needs the corresponding type rule, and introduction and elimination rules. This makes for a considerable amount of rules in MLTT with W- and M-types, while our theory only has 6 relevant deduction rules. Second, it is an interesting fact that the (dependent) function space can be described as a coinductive type. This seems to be
widely accepted but we do not know of any formal treatment of this fact. Thus the presented
type theory allows us to deepen our understanding of coinductive types.

Contributions Having discussed the raison d’être of this work, let us briefly mention the
technical contributions. First of all, we introduce the type theory and show how important
logical operators can be represented in it. We also discuss some other basic examples, including
one that shows the difference to existing theories with coinductive types. Second, we show
that computations of terms, given in form of a reduction relation, are meaningful, in the sense
that the reduction relation preserves types (subject reduction) and that all computations are
terminating (strong normalisation). Thus, under the propositions-as-types interpretation, our
type theory can serve as formal framework for intuitionistic reasoning.

Related Work A major source of inspiration for the setup of our type theory is categorical
logic. Especially, the use of fibrations helped a great deal in understanding how coinductive
types should be treated. Another source of inspiration is the view of type theories as internal
language or even free model for categories. This view is especially important in topos theory,
where final coalgebras have been used as foundation for predicative, constructive set theory.

Let us briefly discuss other type theories that the present work relates to. Especially close
is the copattern calculus, as there the coinductive types are also specified by the types of their
destructors. However, said calculus does not have dependent types, and it is based on systems
of equations to define terms, whereas the calculus in the present work is based on recursion and
corecursion schemes.

To ensure strong normalisation, the copatterns have been combined with size annotations.
Due to the nature of the reduction relation in these copattern-based calculi, strong normalisation
also ensure productivity for coinductive types or, more generally, well-definedness. As another
way to ensure productivity, guarded recursive types have been proposed and guarded recursion
has been extended to dependent types. Guarded recursive types are not only applicable to
strictly positive types, which we restrict to here, but also to positive and even negative types.
However, it is not clear how one can include inductive types into such a type theory, which are,
in the authors opinion, crucial to mathematics and computer science.

Sneak Preview

Let us briefly peek at the calculus we are going to see in the talk. An important type formation
rule is that for coinductive types:

\[
\frac{k = 1, \ldots, n \quad \sigma_k : \Gamma_k \Rightarrow \Gamma \quad \Theta, X : \Gamma \Rightarrow * \mid \Gamma_k \vdash A_k : *}
\quad \Theta \mid \emptyset \vdash \nu(X : \Gamma \Rightarrow * ; \sigma ; \bar{A}) : \Gamma \Rightarrow *
\]

Here, \( n \) is a positive natural number and each \( \sigma_k \) is a substitution for the variables of the context
\( \Gamma \) by terms in context \( \Gamma_k \). The notation \( X : \Gamma \Rightarrow * \) indicates a type constructor variable that
can be instantiated with terms according to \( \Gamma \). The judgement \( \Theta, X : \Gamma \Rightarrow * \mid \Gamma_k \vdash A_k : * \)
then says that each \( A_k \) is a type with free type constructor variables in \( \Theta \) extended with \( X \),
and free term variables in \( \Gamma_k \). The intuition is that \( \nu(X : \Gamma \Rightarrow * ; \sigma ; \bar{A}) \) has destructors that can
only be applied to elements of this type instantiated with terms that match with \( \sigma_k \) and with
output of type \( A_k[\nu/X] \). These destructors are formally terms for each \( k = 1, \ldots, n \):

\[
\xi_k : (\Gamma_k, x : \nu \odot \sigma_k) \Rightarrow A_k[\nu/X],
\]

where \( \nu \odot \sigma_k \) denotes the instantiation of the type with terms in \( \sigma_k \).