CCC-Semantics

This is a formalisation of the interpretation of the simply-typed lambda calculus into cartesian closed categories.

Structure

The project is split into two parts, with three additional files, two files which connect the two parts, and a simple library used throughout.

- `CCCSemantics.Lambda`, which contains formalisation of the syntax, i.e. simply typed lambda calculus, together with substitutions. Also defines beta-eta equivalence of terms, which is used for showing that the interpretation is well-behaved.
- `CCCSemantics.Categories`, a small library for category theory, formalising up to and including the Yoneda lemma, as well as the notion of CCCs.
- `CCCSemantics.Interpretation` defines the interpretation of the syntax into cartesian closed categories, and show that this interpretation respects beta-eta equivalence of terms and substitutions. Also proves the substitution lemma.
- `CCCSemantics.SyntacticCategory` defines the syntactic category, with objects contexts and morphisms substitutions up to beta-eta equivalence. Also shows that every `Struct` corresponds to a cartesian closed functor from the syntactic category, and vice versa. I did have time to figure out a good notion of morphism between structures, so no equivelence between the category of structures and of cartesian closed functors was proved.
- `CCCSemantics.Library` provides basic constructions, such as equivalences of types, and some lemmas, for example congruence for binary functions.

`CCCSemantics.Lambda`

We define the types and signatures in `Lambda/Type.lean`. The types include the unit type, product types, function types, as well as base types, given by a paramter. Signatures, also defined there, are a collection of base types and terms, with a typing function from the base terms to any type, not just base types. Signatures do not have support for equations between terms, which could be considered as a future possible extension.

We define contexts over a signature in `Lambda/Context.lean`, together with variables over a context.

In `Lambda/Term.lean` we give the definition of well-typed terms in context, given as an inductive family indexed by context and types, with the expected term constructors for the corresponding types. Variables are given by De Bruijn indices, enforced to be of correct type in the given context.

Before we are able to define substitutions we need to be able to give weakening

for terms, which we allow for by giving renamings (`Lambda/Renaming.lean`). These give actions on variables and terms, as well as a composition operation. We show that renaming a composite corresponds to composing renamings, which shows that terms and variables form a presheaf over the category of renamings.

For substitutions (`Lambda/Substitution.lean`) we use the same structure as renamings, being indexed by two contexts, and are represented as linked lists of terms. We also give operations for composing renamings with substitutions on either side, called `weaken` and `contract`, which we use to prove that composition of substitutions behave well, i.e. is associative and has left/right units. We then show that substitutions for a cartesian category, with product the concatenation of contexts. Finally, we add constructs for exponential objects, which are used later in the construction of the syntactic category. These don't give rise to a cartesian closed category, since the laws for cartesian closure hold only up to beta-eta equivalence.

The beta-eta equivalence of terms and substitutions is defined in `Lambda/Reduction.lean`. These are used later for the definition of the syntactic category, and we show that interpretation into CCCs preserve the equivalence as equality, which lets us construct a mapping from the syntactic category to any CCC by way of a structure.

`CCCSemantics.Categories`

The category theory library defines categories (`Categories/Category.lean`) using a *bundled* representation, rather than following Mathlib's definition using type-classes. For better inference of operators in specific choices of categories, such as product categories, we use `unif_hint`s. This is not necessarily a better choice, but was more an experiment to see how well such a library would work.

Functors (`Categories/Functor.lean`) and natural transformations (`Categories/NaturalTransformation.le` are defined as standard, and by making use of the bundled representation we are able to more easily define certain categories, such as `Cat` the category of categories (`Categories/Instances/Cat.lean`).

We also define functor categories (`Categories/Instances/Func.lean`) and the category of types (`Categories/Instances/Types.lean`), so we have the category of presheaves. This then allows us to construct the Yoneda-embedding, which we make use of to show that presheaf categories are cartesian closed (`Categories/CartesianClosed/Presheaf.lean`). We prove the Yoneda lemma as a nice result, but it is not used for the rest of the formalisation.

We also define adjunctions (`Categories/Adjunction.lean`), using the unit-counit definition. There is a construction of adjunctions from the definition of naturally hom-set `Hom (F -) -` and `Hom - (G -)` as well, which is used to construct the hom-product adjunction for cartesian closed categories.

For cartesian categories (`Categories/CartesianClosed.lean`) we use the elementary definition, requireing an assignment for each pair of objects to

a product, defined explicitly, rather than through a general interface for limits. This is defined as a type-class indexed by categories. We define cartesian closed category as an extended type-class on cartesian categories, additionally having an assignment of exponentials to each pair of objects. We show how both the assigment of products and of exponentials give functors, and that these are adjoint as expected. We show that the category of types (`Categories/CartesianClosed/Types.lean`) and, as mentioned earlier, presheaf categories, are cartesian closed.

`CCCSemantics.Interpretation`

This module defines interpretation of the STLC into any CCC given a `Struct` over the category. This `Struct` contains the data for interpreting base types/terms.

Types are then interpreted by structural recursion, taking base types as given in the `Struct`, product types to products, function types to exponentials, and the unit type to the terminal object. This is extend to contexts by recursion on contexts, interpreting it as a finite product, with one occurence of an object for each type in the context.

Then we interpret variables in a context as projections, and terms are interpreted as would be expected, taking each introduction/elimination form to its corresponding morphism in the CCC. The interpretation of variables extends to renamings, and the interpretation of terms extends to substitutions.

We show the substitution lemma, which says that interpreting a term with substitution is the composite of the interpretation of the term with the interpretation of the substitution.

We then have theorems which say that interpreting terms respects beta-eta equivalence (`TmEquiv`) of terms, and the same for substitutions. The substitution lemma is required of the `arr_` rule. There are also lemmas showing that the interpretation of the concatenation of contexts is a product of the interpretation of each argument, and similarly for exponentiation. This is later to show that `Struct`s correspond to cartesian closed functors from the syntactic category into the target CCC.

`CCCSemantics.SyntacticCategory`

This module defines the syntacitc category, where objects are contexts and morphisms are substitutions quotiented by beta-eta equality. We show that it is cartesian, which follows from the fact that the category of substitutions is cartesian, and then that it is cartesian closed, where we make use of the fact that the category of substitutions has exponentials up to beta-eta equivalence. We then give a construction of cartesian closed functors $\mathrm{SynCat}_\Sigma \to \mathcal{C}$ for any given `Struct` over a CCC $\mathcal{C}$. We provide a construction in the other direction as well, giving a `Struct` for any cartesian closed functor from the syntactic category.

This should provide an equivalence of categories between a category of CC functors from the syntactic category (possibly by isomorphism) and the category of `Structs`, but that depends on both a good notion of morphism in both cases, which I did not have time to develop.