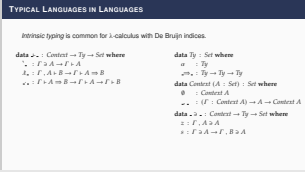
Syntax-Generic Operations, Reflectively Reified

Hello everyone, my name is Tzu-Chi, I am a research assistant at Academia Sinica in Taiwan. Today I will be talking about the joint work of my advisor's and mine, the title is "Syntax-generic Operations, Reflectively Reified". So firstly we will see what are syntax-generic operations, then I will elaborate on what Reflectively reified means.



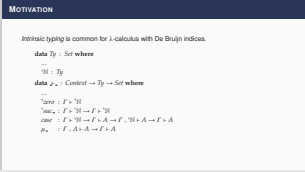
For starters, Let's take a look at a common thing dependently typed programmers would do, that is defining a programming language in a dependently typed programming language, such as Agda. Here is an example of the syntax of a language defined in Agda. It's a simply-typed lambda calculus. We define a datatype called Ty to describe the possible types for this language, a term has either a single type or a function type. Here *Context* is defined as the List of Ty. And we have a simple "has" relation that represents variables. For if a variable is typed it must be in the context. And we know that a simply typed lambda calculus has three constructions, each corresponding a a typing rule. Here are three constructors, each of them corresponds to the variable rule, the abstraction rule, and the application rule. We can see for a term to be constructed, it must be well-typed by definition, that's what we call intrinsic typing.

Syntax-Generic Operations, Reflectively Reified

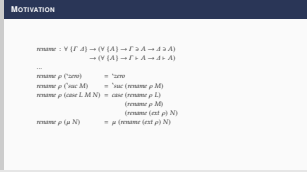
Scope-safe syntax operations:

$$
\begin{aligned}
&\mathit{rename} : \forall\ \{\Gamma\ \Delta\} \to (\forall\ \{A\} \to \Gamma \ni A \to \Delta \ni A) \\
&\qquad\qquad \to (\forall\ \{A\} \to \Gamma \vdash A \to \Delta \vdash A) \\
&\mathit{rename}\ \rho\ (\hat{\ }\ x) \quad = \hat{\ }\ (\rho\ x) \\
&\mathit{rename}\ \rho\ (A\ N) \quad = A\ (\mathit{rename}\ (\mathit{ext}\ \rho)\ N) \\
&\mathit{rename}\ \rho\ (L \cdot M) \quad = (\mathit{rename}\ \rho\ L) \cdot (\mathit{rename}\ \rho\ M)
\end{aligned}
$$

Then let's define some operations on this syntax. Again, because of intrinsic typing, such operations are scope-safe. For example, a *rename* function says that if there's a mapping from every variable in a context Γ to the other context Δ, we can always find a mapping from every term that is typed in Γ to a term typed in Δ. Because we can remap every variable in that term. If we encounter a subterm that has a extended context, for example, the case of lambda abstraction, the variable mapping should be extended accordingly. We can see that rename is still called recursively, but the variable mapping function is modified with ext.



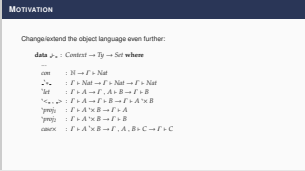
Now what if we extend the language? we might want natural number primitives in our language, so we extend the Ty datatype with a new type, we add four typing rules, two of them, zero and suc are for constructing natural numbers, and another two rules are for branching and recursion.



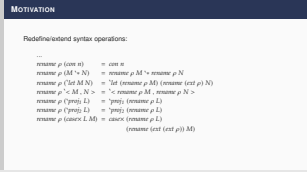
If we want a rename function for this extended language, we can copy and paste the rename we just defined and add a new clause for each constructor. The patterns still follow, a renaming always take a term from a constructor to one with the same constructor, if encounter subterm, call rename recursively, if there's context extension, extend the variable mapping as well. So no matter how many constructors there are we follow the same logic.

Syntax-Generic Operations, Reflectively Reified

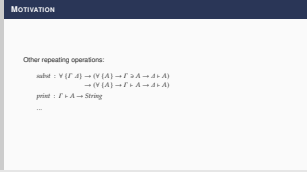
Motivation



What if we extend the language even further? Here we add even more rules. syntax sugaring, pairs and projections, etc.



The *rename* function must be extended as well, so everytime we make some changes to the object language, it becomes a very repeating work.



There are other repeating operations that should be redefined for every change in the object language, you probaly would want some kind of generic programs that work for every syntax, or a mechanism that generates these operatoins from any given datatypes of this kind.

WHERE WE ARE GOING...

1. Existing Work for Syntax-generic Operations
2. Elaborator Reflection to the Rescue
3. Discussion

Now we know the problem we are dealing with. In the next section, I will introduce an existing work that eliminates the repetitions we just mentioned. After that I will illustrate how we develope an alternative approach based on this previous work, by incorporating elaborator reflection. We will make some comparisons along the way, and finally we raise some questions regarding our own work, and hopefully get some feedbacks from you.

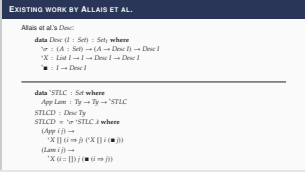
Let’s start with the existing work.

Existing work

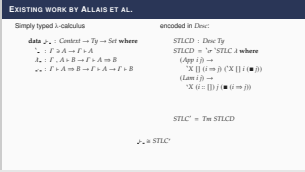
There are generic libraries for a family/families of syntaxes with binders.

We improve upon Allais et al.'s approach presented at ICFP '18 (later published in JFP '21).

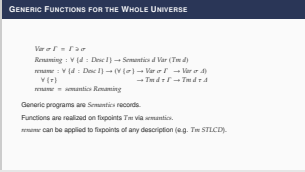
There have been some libraries that provide generic operations for the problem we introduced. Those programs are syntax-generic, that's the first part of our title. We focus on one of such works, that is the generic library by Allais et al. They have developed a framework with a variety of syntax-generic operations, such as renaming, subsitution, printing and scope checking.



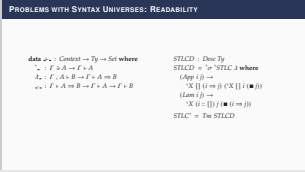
How do they achieve it? They provide a universe of descriptions called Desc that describes a family of syntaxes. We won't go into its details here, but we can say that every inhabitant in this universe represents a syntax, and simply-typed lambda calculus is one of them. The parameter I in the Desc universe says that a syntax is intrinsically typed by I and has a context of List of I. So simply-typed lambda calculus can be encoded in the universe Desc



A simply-typed lambda calculus here on the left hand side has just been rewritten as a description on the right hand side. In fact, to use such generic libraries, programmers must encode their syntax in the given description, because syntax-generic operations are defined on these descriptions. To acquire something structurally similar to the native symtax datatype, we can use the Tm type constructor, which takes the fixpoint of the functor of a description. We may prove that a syntax defined as a isolated datatype is isomorphic to the fixpoint of such functor.

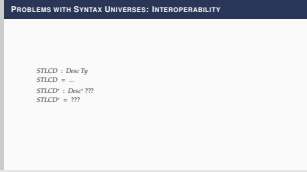


With such a universe, Allais they can and have defined some generic functions that work for a whole family of syntaxes. For example, they have defined a generic renaming function. A semantics datatype with the upper case S is a description of a generic function given a syntax description d. You can see Renaming here is generic as it's quantified over d. The sementics function with lower case S is used to obtain the actual generic rename function, we can see for every description d, this function works on Tm d. It essentially says that there's a rename for every given d, and of course there's a rename for the description of simply-typed lambda calculus.



We quickly summerize what we think are the reasons these libraries are not widely adopted. First of all is readability. One good thing about of intrinsic typing is that types of constructors closely resemble typing rules, and typing rules are less obvious for syntaxes defined in generic universes.

Secondly, to utilize such generic libraries, programmers are required to understand the generic universe instead of just defining syntaxes the way they want.



And they need to learn a new generic representation everytime they want some features that only exist in another generic library, the syntax they are working on must be redefined. This leads us to te third problem, interoperability. It would be hard to use two or more generic libraries at once.

Therefore, we want the best from both worlds, we want programmers and researchers to use native definitions whenever possible, while generic programs can still be invoked. We acheive this by elaborator reflection. Elaborator reflection is the metaprogramming mechanism provided by Agda, it allows us to read and define datatype and function definitions.

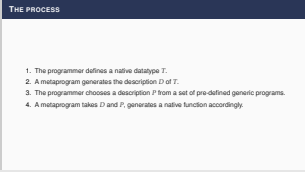
└Elaborator Reflection to the Rescue

Elaborator Reflection to the Rescue

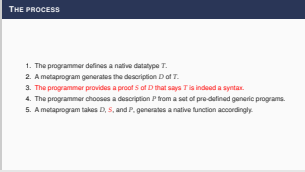
"Datatype-Generic Programming Meets Elaborator Reflection" by Josh Ko, Liang-Ting Chen, and Tzu-Chi Lin at 15:50, Tuesday.

Syntax-generic operations are Datatype-generic programs with constraints.

I would like to shamelessly promote the other work we are presenting at ICFP, Datatype-Generic Programming Meets Elaborator Reflection. My advisor Josh will present it on Tuesday. We have demostrated in that work that we can define programs on a family of datatypes with our program description, while using elaborator reflection to reify such programs as natural function definitions defined on native datatypes. What does this have to do with syntax-generic libraries that we spent so much time intorducing? It turns out, syntax-generic programs sometimes datatype-generic programs. We can constrain a subset of all datatypes such that datatypes in this subset are also describable bya generic universe, in this case Allais their library.

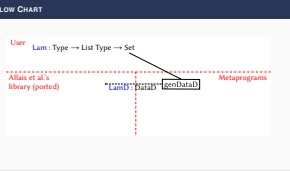


Now let us skip the introduction of datatype-generic programming and metaprograms. To get the whole picture, let's do a rundown of our alternative process for a programmer to invoke generic programs. Firstly they define a native datatype T that we know to be a syntax, instead of relying on any generic description. Then by metaprograms in our datatype-generic library, they get the datatype description D of T. Then the programmer can choose a generic program P to reify. This generic program is pre-defined by the geneic library. Lastly, another metaprogram takes D and P, and gives the programmer a native, reifed funtion definition that works on S, as if everything is defined by hands.



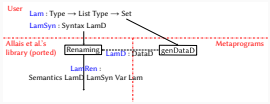
But this process is actually not sufficient, how does the generic library know S is a syntax? To pre-define a datatype-generic program that work specifically on a family of syntax, we must also pre-define a predicate that says a datatype is indeed a syntax. So the geneic library should also provide predicates that constrain general datatypes, saying they are syntaxes. And the programmer must provide a proof of this predicate when choosing a generic program to reify. So the process we just showed actually requires an extra step, that is the programmer providing a proof of the datatype T being a syntax.

Let's do a rundown again, but with our actual definitions. We have three worlds here, the user's own world, the syntax-datatype-generic library, and metaprograms. Say the user defines a language called Lam, indexed by the type and context of a term.

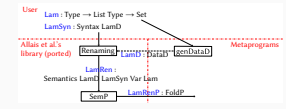


They can invoke a metaprogram genDataD, which generates a datatype description of type DataD, we call it LamD.

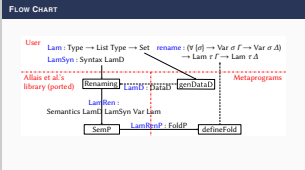
Then to use the generic library, the user must provide a proof that the description we just generated is syntax. Here Syntax is the predicate, and we call the proof LamSyn.



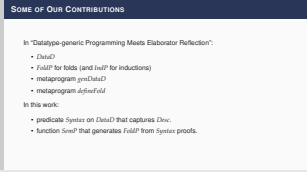
Then it's the job for the generic library. Renaming here is a function that generates syntax-generic programs. Syntax-generic programs are represented as Semantics.



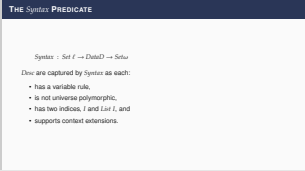
But our metaprograms that generates native functions are for datatype-generic programs. so we need another translation in the generic library that is from syntax-generic programs to datatype-generic programs. SemP is this translation and FoldP is the type of datatype-generic fold programs.



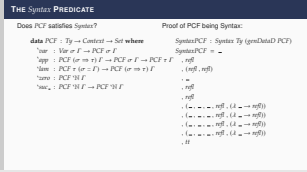
Finally, defineFold is a metaprogram that generates actual function definitions. So the user doesn't need to understand the detailed definitions in the generic libary, they can write native datatype and get native function definitions if they know what metaprograms and functions to call.



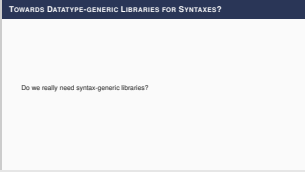
The datatype and program descriptions are provided by our other work we mentioned, so are the metaprograms. In this work we provide the Syntax predicate and the translations built arount it.



Unfortunately our time is limited, we can't go into the details of the Syntax predicate.



We were worried that the Syntax proof might be too complicated, because if it's too complicated it would not save any efforts and nobody would want to use it. Fortunately in this case it's pretty straightforward. Suppose we define a language PCF, which has five constructors. For this PCF datatype to be a syntax, a lot of things has to be considered, like the datatype's universe level, numbers of parameters and indices, and constraints on each field of each constructor. In this case the parameters must be empty and there must be exactly two indices, a type and a list of type. It turns out most of these are equality proofs, and if the datatype truely is a syntax, these can be proved by simply using the refl constructor. Proofs of any datatypes being syntaxes look pretty much the same, so it is possible to write yet another metaprogram for generating such proofs. Which is a future work we haven't done yet.



As you can see, our work has a lot to be done. Since we are running out of time, I would like to address one issue that's probably the elephant in the room. What if people actually don't want syntax-generic operations at all? Maybe they only define a language once in a while, and it's not worth the time looking up what libraries they can use. Or, since researchers define languages with new features all the time, maybe it's common for them to come up something no generic universes can cover. Our framework could still be help in that case, maybe our metaprograms can analyse the constructors in a datatype and determine which of them fit in a universe, and generate functions that are partially defined, then leave the uncertain parts to the programmer. So what do you think about it? Please share with us your conerns or what you think this framework can be going. Thank you all for listening.