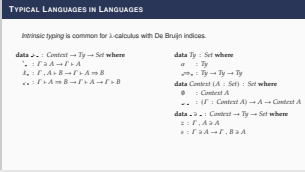
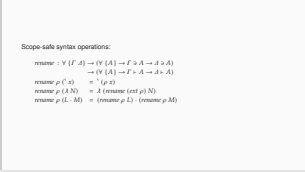
Syntax-Generic Operations, Reflectively Reified

Hello everyone, my name is Tzu-Chi, I am a research assistant from 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 what Reflectively reified means.

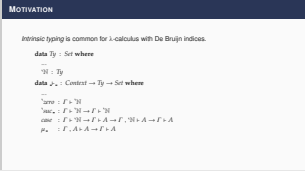


For starters, Let's take a look at a common thing dependently typed programmers would do, that is defining a programming language in a dependently typed programming language, such as Agda. It's a language in language. Here is an example of the syntax of a language defined in Agda. It's a simply-typed lambda calculus, and with the help of intrinsic typing, the syntax definition is very concise. We define a datatype called Ty to describe the possible types for this language, the term is either of 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 having a a typing rule. Here are three constructors, each of them corresponds to the variable rule, the abstraction rule, and the application rule.

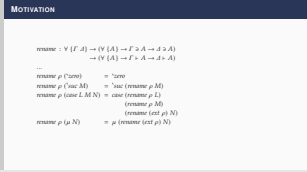
Syntax-Generic Operations, Reflectively Reified



Then let's define some operations on this syntax. Such operations are scope-safe. For example, let's define a rename function. 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 the term is still called recursively, but the variable mapping function is modified with ext.



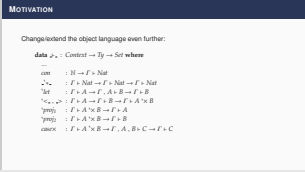
Now what if we extend the language? we might want natural number primitives in our language, so we extend the Ty datatype with a new type, we add four typing rules, two of them, zero and suc are for constructing natural numbers, and another two rules for branching and recursion.



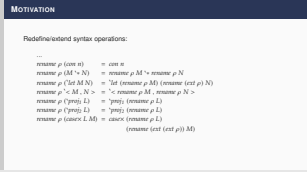
If we want a rename function for this extended language, we can copy and paste the rename we just defined and add a new clause for each constructor. We can notice here the patterns still follow, a renaming always take a term from a constructor to one with the same constructor, if encounter subterm, call 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

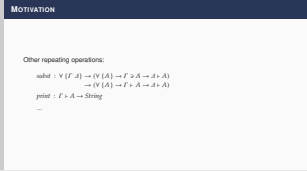
Motivation



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



Then, the *rename* function must be extended everytime we make some changes to the object language. The pattern still applies and it becomes a very repeating work.



There are other repeating operations that should be redefined for every change in the object language, if you have already defined a language with multiple such operations, you probaly would want some kind of generic program that works for every syntax, or a mechanism that generates these operatoins from any given languages.

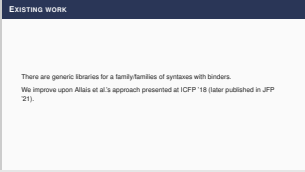
WHERE WE ARE GOING...

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

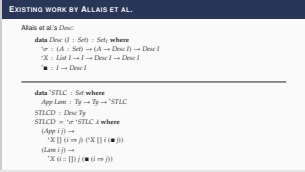
Now we know the problem we are dealing with. In the next section, I will introduce an existing work for eliminating the repetitions we just mentioned. After that I will illustrate how we develope an alternative approach based on this previous work, by incorporating elaborator reflection. We will make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

Existing Work for
Syntax-generic Operations

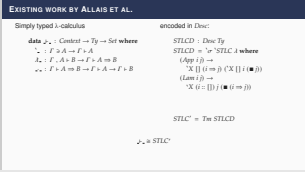
Let’s start with the existing work.



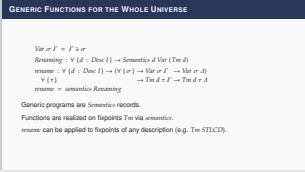
Now we turn our eyes on a work by Allais et al. They have developed a syntax-generic framework with a variety of syntax-generic operations, such as renaming, subsitution, printing and scope checking.



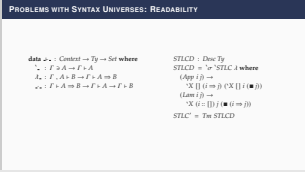
How have they achieved it? They provide a universe of descriptions called Desc that describes a family of syntaxes. We won't go into its details here, but we can say that every inhabitant in this universe represents a syntax, and simply-typed lambda calculus is one of them. The parameter I in the Desc 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 as a description indexed by the Ty datatype we have defined previously, saying that it's a syntax where every term in it has a type represented by Ty, and has a context that is a list of Ty.



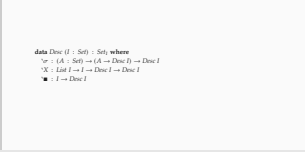
A simply-typed lambda calculus here on the left hand side can be rewritten as a description on the right hand side. In fact, to utilize such generic libraries, programmers are required to encode their syntax in the given description, because syntax-generic operations are defined on these descriptions. To acquire something structurally similar to the native symtax datatype, we can use the Tm type constructor, it that 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 some functor. But to use a generic library one must follow its rules.



With a universe they can and have defined 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. This semantics with uppercase S is just an encoding, the sementics function with lower case S is used to obtain a generic rename function, we can see for every description d, this function works on Tm d. It can be applied given any d, and of course it can be applied given 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 merit of intrinsic typing is that types of constructors closely resemble typing rules, and typing rules are less obvious for syntaxes defined in generic universes.

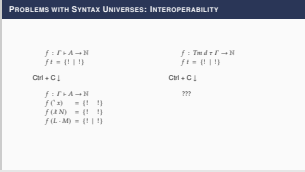


Secondly, to utilize such generic libraries, programmers are required to understand the generic universe instead of just defining syntaxes the way they want.

Problems with Syntax Universes: Interoperability

$STLCD : Desc\ Ty$
$STLCD = ...$
$STLCD' : Desc'\ ???$
$STLCD' = ???$

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



The interoperability with existing tools also suffer, for example, because every typing rule corresponds to one constructor, we can utilize the case-spliting mechanism provided by Agda's editor mode, by placing te cursor in th hole of f, and press Ctrl and C in the Emacs or VS Code editor, we can easily see how many clauses we should define for a function that takes a natural term. We can't benefit from these tools or existing IDE supports when using generic universes, case spliting on Tm would work, but it will look like a mess.

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.

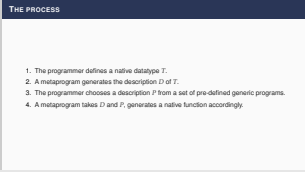
└Elaborator Reflection to the Rescue

Elaborator Reflection to the Rescue

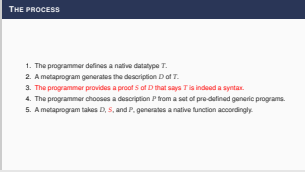
"Datatype-Generic Programming Meets Elaborator Reflection" by Josh Ko, Liang-Ting Chen, and Tzu-Chi Lin at 15:50, Tuesday.

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. Josh will present it on Tuesday. We have demostrated in that work on how to mix elaborator reflection with datatype-generic programming, such we can define programs that work for a family of datatypes, 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 can sometimes be seem as a subset of datatype-generic programs. In this case, we can constrain a subset of all datatypes such that datatypes in this subset are also describable by Allais et al.'s 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 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 of that datatype. Then the programmer can choose a generic program to reify. This generic program is pre-defined by the "geneic library programmers". 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.

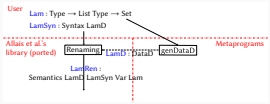


We know that the description D, generated from a syntax datatype is a syntax, but how does the generic library know? To pre-define a datatype-generic program that work specifically on a family of syntax, we must also pre-define a predicate that says a datatype is indeed a member of the syntax family. So the process we just mentioned is not suffient. The “geneic library” should also provide predicates that constrain general datatypes, saying what generic programs exist for what kinds of datatypes, in this case the 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 must provide proofs of the datatype T being a syntax.

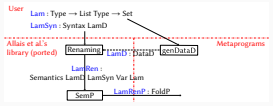
Let's do a rundown again, but with our actual definitions. We have three worlds here, the user's own world, the syntax-datatype-generic library, and metaprograms. Say the user defines a language called Lam, indexed by the type and context of a term.

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

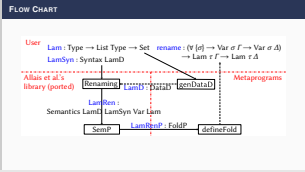
Then to use the generic library, the user must provide a proof that the description we just generated is syntax. Here Syntax is the predicate, and we call the proof LamSyn.



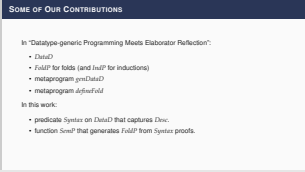
Then it's the job for the generic library. Renaming here is a function that generates syntax-generic programs. Syntax-generic programs are represented as Semantics.



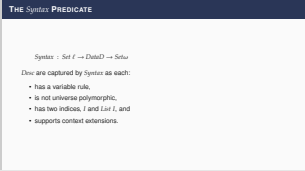
But our metaprograms that generates native functions are for datatype-generic programs. so we need another translation in the generic library that is from syntax-generic programs to datatype-generic programs. SemP is this translation and FoldP is the type of datatype-generic fold programs.



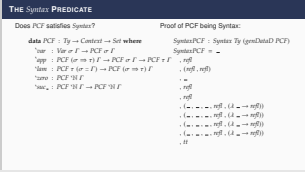
Finally, defineFold is a metaprogram that generates actual function definitions. So the user doesn't need to understand the detailed definitions in the generic libary, they can write native datatype and get native function definitions if they know what metaprograms and functions to call.



These are our contributions. These four are already provided in our other work that we have mentioned. DataD is datatype description, FoldP is description of datatype-generic programs which are folds, genDataD is the metaprogram for reading datatype description and defineFold is for generating native function from descriptions. In this work, we introduce a framework that ports syntax-generic libraries to more datatype-generic ones, by defining the predicate Syntax and some translation functions built around this predicate.



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



But we can shortly emphasize on something that concerned us when developing this framework, that is the complexity of the Syntax proof. If this proof is too complex, probaly no one would want to use it. Fortunately, it turns out proving something to be a syntax is actually pretty straightforward. Let's see an example. Suppose we define a language called 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. For example, in this case the parameters must be empty and there must be exactly two indices, a type and a list of type. Fortunately, these constraints are mostly equality proofs, and once the programmer provides a datatype that satisfies these conditions, these can be proved by simply using the refl constructor. Proofs of different datatypes being syntaxes look pretty much the same and there are some simple rules to follow, so it is possible to write yet another metaprogram for generating such proofs. Which is a future work we haven't done yet.

Do we really need syntax-generic libraries?

Finally, let's talk about the cons. More issues will arise if we start porting more syntax-generic libraries other than Allais et al.'s. First of all we are still limited by a universe, that is the Agda inductive datatype universe. What is expressible in other universes might not find their conterparts as native datatypes. Then for more complicated universes, the Syntax proofs could be more complicated and not so straightforward. Feel free to share your thoughts on this framework, does it look promising or are there critical issues making it undesirable? Please let us know.