Syntax-Generic Operations, Reflectively Reified

Extended Abstract

Tzu-Chi Lin
vik@iis.sinica.edu.tw
Institute of Information Science, Academia Sinica
Taipei, Taiwan

Hsiang-Shang Ko
joshko@iis.sinica.edu.tw
Institute of Information Science, Academia Sinica
Taipei, Taiwan

Abstract

Libraries of generic operations on syntax trees with binders are emerging, and one of these is Allais et al.'s [2021] datatype-generic library in Agda, which provides syntax-generic constructions but not in a conventional form preferred by programmers. We port a core part of Allais et al.'s library to our new datatype-generic framework, which uses Agda's elaborator reflection to reify generic constructions to programs close to what programmers would write by hand. We hope that this work will make syntax-generic libraries such as Allais et al.'s more attractive, and stimulate discussion on the development of generic libraries.

1 Introduction

When implementing embedded domain-specific languages (DSLs), dependently typed programmers make use of the host languages' type systems to enforce properties of the syntaxes. In particular, when the syntaxes have binders and are typed, *intrinsic typing* has become a standard technique to make the programs well scoped and typed [Kokke et al. 2020, Part 2]. Such syntaxes share similar type structures, operations, and lemmas (with the simplest examples being renaming and substitution). Traditionally, programmers need to somewhat tediously redefine the operations for every distinct syntax. Recently, there have been generic libraries providing constructions that can be specialised for a whole family of syntaxes with binders [Allais et al. 2021; Fiore and Szamozvancev 2022; Ahrens et al. 2022], although it remains to be seen whether these libraries will be widely adopted.

We will focus on Allais et al.'s [2021] Agda library, which treats syntax-generic programs as special cases of *datatype-generic* programs [Gibbons 2007; Benke et al. 2003; Altenkirch and McBride 2003]. Their approach (recapped in Section 2) is more or less standard in Agda: The syntax of a DSL is specified as a 'description' d : Desc I (where I is the set of the DSL types), from which a datatype Tm d of syntax trees is derived. There is a semantics operator that traverses a syntax tree to compute a result; the operator is parametrised by a Semantics record, which specifies what computation to perform. The library provides various Semantics parametrised by d, which act as syntax-generic programs, and can be instantiated with semantics as operations on syntax trees of DSLs that can be described within the Desc universe.

One potential problem that may prevent Allais et al.'s library (and in general, libraries following the standard approach to datatype-genericity in Agda) from being widely adopted is the lack of interoperability: Programmers using Allais et al.'s library are restricted to using datatypes of the form Tm d, which are rather different from the kind of native datatypes that programmers would normally write; this prevents access to other datatype-generic libraries (which use their own universes instead of Desc), and makes language and editor support for native datatypes (such as representation optimisations [Brady et al. 2004] and interactive case-splitting) less effective. The problem also arises for the operations instantiated with semantics, which are not as easy to work with as the hand-written versions (in particular when the definitions need to be inspected). To address the problem (for datatype-generic libraries in general), the present authors (together with Liang-Ting Chen) proposed an Agda framework [Ko et al. 2022] which uses *elaborator reflection* [Christiansen and Brady 2016] to reify generic constructions as native datatypes and functions close to hand-written forms. With the framework, programmers can keep the conventional programming style, and replace some of the programs that had to be written by hand with similar-looking ones automatically generated from generic libraries.

Here we report (in Section 3) a small but successful experiment porting a core part of Allais et al.'s library to our framework, allowing programmers to write datatypes of syntaxes in conventional forms and then reify Allais et al.'s syntax-generic operations as natural-looking functions. We plan to give a demo at the workshop and show that our framework can potentially make syntax-generic libraries such as Allais et al.'s more attractive to programmers. Moreover, currently there are noticeable limitations of our framework and of Allais et al.'s library, which we hope will stimulate discussion on how the development of (syntax-)generic libraries can be pushed further (Section 4). Our Agda code is available at https://github.com/Zekt/Generic-Scoped-Syntax.

TyDe'22, September 11, 2022, Ljubljana, Slovenia

2 Allais et al.'s Approach

Before giving a simplified account of the user interface to Allais et al.'s library, we present our running example: simply typed λ-calculus. Traditionally, DSL programmers would manually define a datatype Lam in Figure 2, where variables are represented as well scoped and typed de Bruijn indices, defined by Var in Figure 1. Then the programmers would go on and define operations on Lam (renaming, substitution, printing, scope-checking, etc). One simplest example is the rename function in Figure 2, which takes an environment ρ represented as a function mapping variables in Γ to variables in Δ, and applies ρ to all the variables in a term of type Lam. Among the cases of rename, the lam case is more interesting: as ρ is pushed under the binder, it needs to be extended with a case mapping the new variable z to 'itself' (since we are renaming only free variables, whereas the new variable is bound), and the old variables in ρ should be incremented to skip over the binder; this new environment for renaming the body is computed by extend in Figure 1.

Allais et al. show that operations like rename can be implemented generically for a family of syntaxes, and programmers need not redefine them for every new syntax as long as the syntax has a 'description', which is an inhabitant of

```
data Desc (I : Set) : Set₁
```

where I is the set of the types used in the syntax. Instead of giving the definition of Desc, we only give a taste of what descriptions look like by showing a description STLC of simply typed λ-calculus, whose details are not important:

```
data `STLC : Set where
  `App `Lam : Type → Type → `STLC

STLC : Desc Type
STLC = σ `STLC λ where
  (`App σ τ) → `X [] (σ `→ τ) (`X [] σ (■ τ))
  (`Lam σ τ) → `X (σ :: []) τ (■ (σ `→ τ))
```

The point here is that descriptions capture the structure of syntaxes as data, from which we can then compute types and functions for the described syntaxes.

In place of native datatypes like Lam, DSL programmers write descriptions like STLC and use a (fixed-point) operator

```
data Tm (d : Desc I) : I → List I → Set where
  var : Var i Γ → Tm d i Γ
  con : ⟦ d ⟧ (Scope (Tm d )) i Γ → Tm d i Γ
```

to derive syntax datatypes like Tm STLC. Again the details of Tm are not important. We only make a remark that the Lam constructors other than var are encoded by a generic constructor con here; the encoding could be disguised as native constructors using pattern synonyms [Pickering et al. 2016], but only to an extent — for example, the encoding still shows up during interactive case-splitting.

```
data Var : I → List I → Set where
  z : Var i (i :: Γ)
  s : Var i Γ → Var i (j :: Γ)

extend : (∀ {i} → Var i Γ → Var i Δ)
         → Var j (k :: Γ) → Var j (k :: Δ)
extend ρ z     = z
extend ρ (s v) = s (ρ v)
```

Figure 1. Well scoped and typed de Bruijn indices

```
data Type : Set where
  α     : Type
  _`→_ : Type → Type → Type

data Lam : Type → List Type → Set where
  var : Var σ Γ → Lam σ Γ
  app : Lam (σ `→ τ) Γ → Lam σ Γ → Lam τ Γ
  lam : Lam τ (σ :: Γ) → Lam (σ `→ τ) Γ

rename : (∀ {σ} → Var σ Γ → Var σ Δ)
         → Lam τ Γ → Lam τ Δ
rename ρ (var x)   = var (ρ x)
rename ρ (app x y) = app (rename ρ x) (rename ρ y)
rename ρ (lam x)   = lam (rename (extend ρ) x)
```

Figure 2. Simply typed λ-calculus and renaming

For datatypes of the form Tm d, Allais et al. provide a generic traversal function

```
semantics : Semantics d V C
            → (∀ {j} → Var j Γ → V j Δ)
            → Tm d i Γ → C i Δ
```

which is abstracted from the computation pattern of operations like rename. The type of the contents stored in the environment and the type of the result are abstracted as V and C respectively. The first argument of type

```
record Semantics (d : Desc I) (V C : I → List I → Set) : Set
```

specifies the computation to be performed during the traversal. For example, the renaming operation can be provided generically (being parametrised by d) in the form

```
Renaming : (d : Desc I) → Semantics d Var (Tm d)
```

with which we can specialise semantics to rename:

```
rename : (∀ {σ} → Var σ Γ → Var σ Δ)
         → Tm STLC τ Γ → Tm STLC τ Δ
rename = semantics (Renaming STLC)
```

3 Reifying Syntax-Generic Operations

It is not particularly pleasant to program with Tm STLC and rename defined in terms of semantics from Section 2, because the (implementation) details of the generic library would keep showing up in these definitions, complicating subsequent constructions. Our approach is to regard the

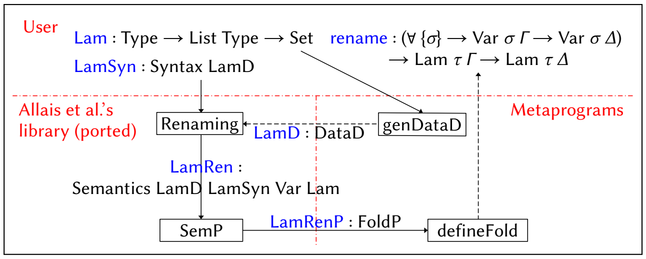

Figure 3. How DSL programmers derive rename from Lam using Allais et al.'s library ported to our framework

generic entities as residing at a 'meta-level', and use metaprograms to perform partial evaluation and generate, at a different level, code specialised for specific syntaxes/datatypes such as Lam. This process is conceptually straightforward in Agda, because in functional settings, partial evaluation is just normalisation [Filinski 1999], which is directly supported by Agda's elaborator reflection.

To demonstrate, we have adapted Semantics, semantics, and Renaming in Section 2 to our framework, enabling programmers to write Lam and then derive rename for Lam. Below we sketch (a slightly simplified version of) the derivation process, which is depicted in Figure 3.

The first step is to derive from Lam a description LamD of type DataD using a macro genDataD:

```
LamD = genDataD Lam
```

Somewhat unsatisfactorily, currently DSL programmers still need to understand DataD descriptions, because they need to provide a proof that Lam is a syntax datatype:

```
LamSyn : Syntax LamD
```

The predicate Syntax holds for d : DataD essentially when there exists a Desc that translates to d. (For other kinds of generic libraries, similar predicates may be defined more directly on the structure of DataD.)

Now programmers can apply Renaming to both LamD and LamSyn to instantiate a renaming Semantics for Lam:

```
LamRen = Renaming LamD LamSyn
```

Allais et al.'s semantics function is replaced by SemP, which computes 'fold programs' of type FoldP from a Semantics:

```
LamRenP = SemP LamD LamSyn LamRen
```

Unlike semantics, fold programs themselves are not executable, but can be reified as native functions by a metaprogram defineFold in conjunction with unquoteDecl, an Agda primitive that defines a given name (in this case rename):

```
unquoteDecl rename = defineFold LamRenP rename
```

This rename function has a definition close to the one in Figure 2, and can be used just like manually defined functions. Notably, the generic entities LamD, LamSyn, LamRen, and LamRenP are just 'meta-level' artefacts for deriving rename, and do not interfere with the subsequent development.

4 Discussion

To address the interoperability problem (Section 1), our framework allows programmers to work with native datatypes (such as Lam) while deriving operations with natural definitions (such as rename) from generic libraries; moreover, by showing that the DataD description of a datatype satisfies several predicates, we gain access to the corresponding libraries all at once. The technique of (re-)defining Allais et al.'s Desc universe as a predicate/subset of our DataD universe is generally applicable to universes of other generic libraries (as long as they are not more expressive than DataD), and makes it easier to use those libraries with our reification metaprograms (compared to reimplementing the metaprograms for each library). Currently our framework is implemented in Agda, but the essential idea depends only on elaborator reflection, and should work in more languages as elaborator reflection becomes more popular.

The reported experiment is small but already reveals some limitations of our framework and of Allais et al.'s library. By discussing these limitations, we hope to illuminate some possible directions for developing (syntax-) generic libraries.

For the framework: It is not so convenient having to carefully apply generic programs to the right arguments and reifying them one at a time — there should be a better user interface (which may require significant changes to Agda's design though). Proofs that DataD descriptions satisfy Syntax are straightforward but tedious, and should be automated, probably also with elaborator reflection. And it may be beneficial to introduce stages explicitly into the framework, for example to reason about the 'cleanness' of generated code [Pickering et al. 2020, Section 4.1].

For syntax-generic libraries: Currently the largest development done with Allais et al.'s library seems to be a strong normalisation proof for simply typed λ-calculus with disjoint sums [Abel et al. 2019, Section 4.3], whose features are standard. While it is conceivable that the universe can be expanded to encode more datatypes, the DSL features covered will always be predetermined when defining the universe. If the intended users include programming language researchers, who invent new features that are unlikely to be covered by existing libraries, then libraries targeting a fixed universe of syntaxes may not be too useful. Here are some possible scenarios where syntax-generic libraries might help: Users might start with a standard syntax definition and then modify it to accommodate new features; this is currently supported by our framework, which allows definitions to be printed (rather than unquoted) and then copied and pasted into the users' files. Or, exploiting Agda's interactive capabilities, we could generate partial definitions with holes, although there is still the problem of where the holes should appear, which is perhaps no less difficult than the problem of composing syntaxes or type theories [Delaware et al. 2013; Forster and Stark 2020], on which much work is still needed.

Acknowledgments

The work is supported by the Ministry of Science and Technology of Taiwan under grant MOST 109-2222-E-001-002-MY3.

References

Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark Reloaded: Mechanizing Proofs by Logical Relations. *Journal of Functional Programming* 29 (2019), e19:1–43. https://doi.org/10.1017/S0956796819000170

Benedikt Ahrens, Ralph Matthes, and Anders Mörtberg. 2022. Implementing a Category-Theoretic Framework for Typed Abstract Syntax. In *International Conference on Certified Programs and Proofs (CPP)*. ACM, 307–323. https://doi.org/10.1145/3497775.3503678

Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2021. A Type- and Scope-Safe Universe of Syntaxes with Binding: Their Semantics and Proofs. *Journal of Functional Programming* 31 (2021), e22:1–51. https://doi.org/10.1017/S0956796820000076

Thorsten Altenkirch and Conor McBride. 2003. Generic Programming within Dependently Typed Programming. In *Generic Programming (IFIP — The International Federation for Information Processing, Vol. 115)*. Springer, 1–20. https://doi.org/10.1007/978-0-387-35672-3_1

Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nordic Journal of Computing* 10, 4 (2003), 265–289. https://www.mimuw.edu.pl/~ben/Papers/universes.pdf

Edwin Brady, Conor McBride, and James McKinna. 2004. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs*. Lecture Notes in Computer Science, Vol. 3085. Springer, 115–129. https://doi.org/10.1007/978-3-540-24849-1_8

David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *International Conference on Functional Programming (ICFP)*. ACM, 284–297. https://doi.org/10.1145/3022670.2951932

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-Theory à la Carte. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 207–218. https://doi.org/10.1145/2429069.2429094

Andrzej Filinski. 1999. A Semantic Account of Type-Directed Partial Evaluation. In *International Conference on Principles and Practice of Declarative Programming (PPDP) (Lecture Notes in Computer Science)*. Springer, 378–395. https://doi.org/10.1007/10704567_23

Marcelo Fiore and Dmitrij Szamozvancev. 2022. Formal Metatheory of Second-Order Abstract Syntax. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 53:1–29. https://doi.org/10.1145/3498715

Yannick Forster and Kathrin Stark. 2020. Coq à la Carte: A Practical Approach to Modular Syntax with Binders. In *International Conference on Certified Programs and Proofs (CPP)*. ACM, 186–200. https://doi.org/10.1145/3372885.3373817

Jeremy Gibbons. 2007. Datatype-Generic Programming. In *International Spring School on Datatype-Generic Programming (SSDGP) 2006*. Lecture Notes in Computer Science, Vol. 4719. Springer, 1–71. https://doi.org/10.1007/978-3-540-76786-2_1

Hsiang-Shang Ko, Liang-Ting Chen, and Tzu-Chi Lin. 2022. Datatype-Generic Programming Meets Elaborator Reflection. Conditionally accepted for the *International Conference on Functional Programming (ICFP)*. https://github.com/Zekt/Type-Embellishment/raw/master/tex/ICFP22/NDGP.pdf

Wen Kokke, Philip Wadler, and Jeremy G. Siek. 2020. Programming Language Foundations in Agda. http://plfa.inf.ed.ac.uk

Matthew Pickering, Gergo Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. In *International Symposium on Haskell*. ACM, 80–91. https://doi.org/10.1145/2976002.2976013

Matthew Pickering, Andres Löh, and Nicolas Wu. 2020. Staged Sums of Products. In *International Symposium on Haskell*. ACM, 122–135. https://doi.org/10.1145/3406088.3409021