Functorial Syntax for All

Piotr Polesiuk
ppolesiuk@cs.uni.wroc.pl
Institute of Computer Science
University of Wrocław
Wrocław, Poland

Filip Sieczkowski
f.sieczkowski@hw.ac.uk
School of Mathematics and Computer Science
Heriot-Watt University
Edinburgh, United Kingdom

1 Functorial approach to binding

Variable binding, in its many forms, is ubiquitous in programming languages; therefore, approaches to representing binding structures and reasoning about them in theorem proving systems abound. From simple named representations, to nameless and locally nameless syntax via de Bruijn indices, to higher-order abstract syntax and nominal techniques, many approaches have been tried, and many libraries, plugins and formalisations developed. In this talk we present another library, based on the notion of functorial syntax, and report on our experience in its development and use across a number of formalisation projects.

Let us begin by introducing the functorial approach to binding and syntax, via the following representation of λ-terms.

```
Inductive term (X : Set) : Set :=
| var : X → term X
| lam : term (inc X) → term X
| app : term X → term X → term X.
```

The key idea of this representation is to parametrise the type of terms by a *set* `X` that describes a *scope*. The variable constructor (`var`) accepts only variables that are in the scope, while lambda-abstraction (`lam`) extends the scope by one element (type `inc` is isomorphic to `option`). A substitution operation substitutes for a variable added by an `inc` type, and is implemented via simultaneous substitution, which turns out to be a monadic `bind` function.

```
Fixpoint bind {X Y : Set}
  (f : X → term Y) (t : term X) : term Y.
Definition subst {X : Set} :
  term (inc X) → term X → term X.
```

Since our representation is well-scoped, we need a way to explicitly inject terms from one scope to a larger scope, *e.g.*, when we proceed to substitute under a binder. It turns out that this can be achieved by a `fmap` function, which implements a more general *variable renaming*.

```
Fixpoint fmap {X Y : Set}
  (f : X → Y) (t : term X) : term Y.
```

The observation that the renaming is a functorial action and that, therefore, the crucial aspect of syntax with binding is its functoriality with respect to the set of its allowed free variables (and their renamings) is the cornerstone of Fiore et al.'s functorial view of binding [3]. Interestingly, the choice to parameterise terms with sets (and general functions) is not crucial: we can make the construction more general by treating syntax (the type `term` in our example) as a functor from a chosen *renaming* category, whose objects represent scopes and arrows (which appear as the first argument of `fmap` above) represent valid renamings, into the category of sets: in other words, a preasheaf. The observation itself is not new; however, to the best of our knowledge it has not been utilised as a basis of a generic library for binding. In the following sections we sketch how this can be achieved, and what benefits can be garnered from this approach.

2 Type classes for parameterisation wrt. renaming categories

At the core of our approach lies the reification of the notion of a renaming category as a (set of) Coq typeclasses. This includes a notion of arrows (i.e., valid renamings for our domain), together with identity and composition, and their properties, and the notion of the functorial action of type constructors on these arrows, i.e., functoriality, which needs to be provided by the user for each of the types they define. In addition to this, the library provides a *second* category of substitutions, which is connected to the renamings via the usual embedding (which treats a renaming as a substitution) and properties. The action of type constructors on substitutions, which the user also needs to provide, is simultaneous substitution and gives the type constructor a monadic structure.

In order for these notions to work appropriately, a notion of *lifting* an arrow (in either category) over a binder is needed — and expressed via a subsidiary set of typeclasses. Note that this notion is also semantic, and thus not immediately tied to any particular notion of a binder — in particular, the lifting is not a priori limited to single binders (although default instances for treating this case are provided).

This secondary layer of typeclasses is then extended to allow for weakening and single substitution with respect to the kinds of binders that occur in the language under consideration. This, of course, assumes that the appropriate arrows can be defined: while the requisite typeclass to encode weakening exists, a linear instantiation, where all renamings need be bijective, would not be definable, and thus shifting would be disallowed for such an instantiation.

The remainder of the library is defined with respect to these fundamental typeclasses. This includes a number of

built-in instantiations (and constructions on these), derived lemmas and a rudimentary simplification algorithm, which makes term simplification automatic in most commonly encountered cases.

3 Benefits of the approach

We have successfully applied our library to develop some larger formalisations [1, 2, 4, 5] and several toy examples. Based on our experience, we can summarise benefits of our approach as follows.

Multiple instantiations of the framework. While in most cases parametrising terms by sets is sufficient, we found the flexibility of our approach useful in some scenarios. If we have more than one sort of variables (*e.g.*, regular variables and type variables), we can parametrise terms by a tuple of sets — one set for each variable namespace (see [2] for an example of such a formalisation). With a functorial approach we get many desired properties almost for free. For instance, substitutions for variables of different sort are represented by the same bind function, and their commutation is a simple consequence of bind laws.

As another example, we can define intrinsically-typed syntax, when the syntax form a functor on renaming category with typing contexts as objects. We can even go further and restrict a category of renamings. For example, if we restrict renamings to bijective functions, we can use variable-binding machinery to manage resources and formalise calculi with mutable state or other substructural calculi.

Finally, thanks to general approach based on type classes, we can have several instantiations in one formalisation. For instance, we can work with well-scoped terms typed by intrinsically well-kinded types.

Non-standard substitutions. Aside from instantiating the framework with different renaming categories, we can provide other substitution-like operations, as long as they can be expressed as a monadic bind. As an example, we can have variables that represents abstract contexts with one hole while the substitution plugs a term into substituted context, as in some presentation of Parigot's $\lambda\mu$ calculus [7]. See [1] for an example formalisation.

Functoriality. A number of benefits seems to stem from the notion of functoriality itself. We encountered a particularly striking case in a recent formalisation [5], where we were able to define a normalization-by-evaluation algorithm within the internal language of presheaves — crucially, building on top of the functorial syntax of neutral and normal forms. This allowed us a construction where a number of properties, usually proved post-hoc, was baked into the construction. We conjecture that similar benefits can be garnered in formalisation of continuation-passing-style transformations, where the functoriality of continuations is a crucial aspect of correctness of the transformation.

Binding arbitrary sets of variables. The central idea of our approach is to parametrise terms by their scope, while the variable binding phenomenon has a secondary role and is expressed as some operation on the scope. Therefore, we can work with variable binding schemes other than binding just a single variable, as long as we can express them as an operation on the scope. As an example, consider a lambda calculus where functions may bind multiple variables via patterns.

```
Inductive term (A : Set) : Type :=
| lam : ∀ B : Set,
  pattern B → term (B + A) → term A
...
```

In this example, patterns are described by a type indexed by a set of variables bound by the pattern. Note that terms and patterns must be defined in sort `Type`, but we can also define them in `Set` if we describe scopes by some inductive type of syntactic representations of finite sets.

4 Ongoing and future work

In its current state,[1] the library provides the basic functionality that is sufficient for its use in relatively sophisticated formalisations; however, there are several directions that we would like to pursue to allow for wider adoption.

- *Automation.* The library currently provides very limited amount of automation, via simplification tactics. This could be improved in two ways: most importantly, by adapting a better simplification algorithm, for which we could adapt the Autosubst approach [6] via reflection. The second aspect, which we do not pursue at the moment, would require building a plugin to allow defining certain typeclass instances automatically, thus reducing the boilerplate definitions.
- *Contexts via metavariables.* We are exploring an approach to defining (multi-hole) contexts and terms at once via metavariables, which could lessen the formalisation load in cases where general contexts are required. The initial experiments suggest that the approach is compatible with the binding library, although more work is required to design an optimal set of notions that would make the integration of contexts seamless.
- *Additional instantiations.* We are experimenting with additional instantiations and their integration with the library, from linear binders, to better support for binding multiple variables at once, to interesting interactions between well-typed and well-scoped syntax, and their impact on obtaining natural statements for common theorems.

[1] For the sources of a current version of the library, consult the `Binding` subdirectory of the recent formalization of [5], available at https://github.com/logsem/fw-rec-inj/.

References

[1] Dariusz Biernacki, Sergueï Lenglet, and Piotr Polesiuk. 2019. Proving Soundness of Extensional Normal-Form Bisimilarities. *Log. Methods Comput. Sci.* 15, 1 (2019). https://doi.org/10.23638/LMCS-15(1:31)2019

[2] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

[3] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. 1999. Abstract Syntax and Variable Binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 193–202. https://doi.org/10.1109/LICS.1999.782615

[4] Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Typed Equivalence of Effect Handlers and Delimited Control. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany (LIPIcs, Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:16. https://doi.org/10.4230/LIPICS.FSCD.2019.30

[5] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. 2024. The Essence of Generalized Algebraic Data Types. *Proc. ACM Program. Lang.* 8, POPL, *to appear* (2024). https://github.com/logsem/fw-rec-inj/

[6] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 166–180. https://doi.org/10.1145/3293880.3294101

[7] Kristian Støvring and Søren B. Lassen. 2009. A Complete, Co-inductive Syntactic Theory of Sequential Control and State. In *Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 5700)*, Jens Palsberg (Ed.). Springer, 329–375. https://doi.org/10.1007/978-3-642-04164-8_17