Generic Programming with Combinators and Objects йцукен[*]

Dmitry Kosarev
St. Petersburg State University
JetBrains Research
St. Petersburg, Russia
Dmitrii.Kosarev@protonmail.ch

Dmitry Boulytchev
St. Petersburg State University
JetBrains Research
St. Petersburg, Russia
dboulytchev@math.spbu.ru

We present a generic programming framework for OCaml which makes it possible to implement extensible transformations for a large scale of type definitions. Our framework makes use of object-oriented features of OCaml, utilising late binding to override the default behaviour of generated transformations. The support for polymorphic variant types complements the ability to describe composable data types with the ability to implement composable transformations.

В данной работе представлен подход для языка программирования OCaml, который позволяет реализовывать расширяемые трансформации для различных видов определений типов. Этот подход использует объектно-ориентированные возможности OCaml, а именно позднее связывание, чтобы изменять поведение по-умолчанию автоматически сгенерированных трансформаций. Поддержка полиморфных вариантных типов позволяет композиционально описывать типы данных с возможностью реализовать композициональные преобразования.

1. Introduction

Фредерк Брукс (Frederic Brooks) в своей известной книге по инженерии программ "Мифический человекомесяц" ("The Mythical Man-Month") [7] охаректеризовал сущзность программирования следующим образом:

> "The programmer, like the poet, works only slightly removed from pure thought-stuff. Он строит свои замки в воздухе, из воздуха, творя только усилием воображения. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realising grand conceptual structures. (As we shall see later, this very tractability has its own problems.)"

Действительно, нематериальность программ и гибкость их представления призывает к структурированию; отсутствие подходящей структуры логком может привести к катастрофическим последствиям (как это случалось с некоторыми промыщленными проектами в прошлом). One of commonly used ways to bring a structure in software are *data types*. Data types allow to describe the properties of data, what can and cannot be done, and to some extent they prescribe the semantics to data structures. Being kept in runtime, data types make it possible to implement meta-transformations by analysing types (*introspection*) or even creating new types on the fly (*reflection*).

[*]This work was partially supported by the grant 18-01-00380 from the Russian Foundation for Basic Research.

Submitted to:
ML 2018

However, in statically typed languages, as a rule, types are completely erased after the compilation and do not retained in runtime. This has a huge advantage over dynamic typing as, first, programs do not need to inspect types at runtime anymore and, second, a whole class of bad runtime behaviours — type errors — is eliminated. The other side of the coin, however, is that now some transformations, which in untyped languages can be implemented "once and for all", can not be typed and have to be re-implemented for each type of interest. One way to overcome this deficiency is to develop a more powerful type system in which more functions can be typed; as an example we may mention the support for *ad-hoc* polymorphism in Haskell in the forms of type classes [31] and type families [16]. However, due to the totality of type checking and fundamental undecidability results there will always be some "good" programs which cannot be typed. Another approach, *datatype-generic programming* [13], is aimed at developing techniques for implementation of practically important families of type-indexed functions using existing language features. For example, types can be encoded in a substrate language [14, 8, 3], or a part of type information can be saved for runtime [20, 6], or generic functions for a given type can be generated at compile-time automatically [34, 2]. The two approaches we mentioned are in fact complementary — the more powerful type system is the more means for datatype-generic programming the language can incorporate natively. For example, parametric polymorphism makes it possible to natively express many generic functions like length of list of arbitrary elements, etc.

We present a generic programming library GT[1] (*Generic Transformers*), which has been in an active development and use since 2014. One of the important observations, which motivated the development of our framework, was that many generic functions can be considered as a modifications of some other generic functions. While our approach is generative — we generate generic functionality from type definitions — it also makes possible for end users to easily derive variants of generated functions by redefining some parts of their functionality. This is achieved using a method-per-constructor encoding for concrete transformations, which resembles the approach of object algebras [24].

The main properties of our solution are as follows:

- each transformation is expressed in terms of a *traversal function* and a *transformation object*, which encapsulate the "interesting part" of the transformation;
- the traversal function is unique for given type and all transformation objects for the type are instances of a unique class;
- both the traversal function and the class are generated from type definition; we support regular ADTs, structures, polymorphic variants and predefined types;
- we provide a number of plugins which generate practically important transformations in the form of concrete transformation classes;
- the plugin system is extensible: end users can implement their own plugins.

The library we present is an inheritor of our earlier work [6] on implementation of "Scrap Your Boilerplate" approach [20, 21, 22]. However, our experience has shown, that the expressivity and extensibility of SYB is insufficient; in addition the uniform transformations, based solely on type discrimination, turned out to be inconvenient to use. Our idea initially was to combine combinator and object-oriented approaches — the former would provide means for

[1] `https://github.com/kakadu/GT/tree/ppx`

parameterization, while the latter — for extensibility via late binding utilisation. This idea in the form of a certain design pattern was successfully evaluated [5] and then reified in a library and a syntax extension [4]. Our follow-up experience with the library [19] has (once again) shown some flaws in the implementation. The version we present here is almost a complete re-implementation with these flaws fixed.

The rest of the paper is organised as follows. In the next section we give an informal, example-driven exposition of the approach we use. Then we describe the implementation in more details, highlighting some aspects which we find important or interesting. In the following section we consider some examples, implemented with the aid of the library. Related works section observes the relevant approaches and frameworks and compares them to ours. The final section discusses the directions for future development.

Список литературы

[1] *camlp5*. `https://camlp5.github.io`.

[2] *ppxlib*. `https://github.com/ocaml-ppx/ppxlib`.

[3] Florent Balestrieri & Michel Mauny (2018): *Generic Programming in OCaml.* In Kenichi Asai & Mark Shinwell, editors: *Proceedings ML Family Workshop / OCaml Users and Developers workshops, Nara, Japan, September 22-23, 2016*, *Electronic Proceedings in Theoretical Computer Science* 285, Open Publishing Association, pp. 59–100, doi:10.4204/EPTCS.285.3.

[4] Dmitri Boulytchev (2018): *Code Reuse With Transformation Objects. CoRR* abs/1802.01930. Available at `http://arxiv.org/abs/1802.01930`.

[5] Dmitry Boulytchev (2015): *Combinators and Type-driven Transformers in Objective Caml. Sci. Comput. Program.* 114(C), pp. 57–73, doi:10.1016/j.scico.2015.07.008. Available at `https://doi.org/10.1016/j.scico.2015.07.008`.

[6] Dmitry Boulytchev & Sergey Mechtaev (2011): *Efficiently Scrapping Boilerplate Code in OCaml.* In: *Workshop on ML*, ML '11.

[7] Frederick P. Brooks, Jr. (1995): *The Mythical Man-month (Anniversary Ed.).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[8] Manuel M. T. Chakravarty, Gabriel Ditu & Roman Leshchinskiy (2009): *Instant Generics : Fast and Easy.*

[9] Jean-Christophe Filliâtre & Sylvain Conchon (2006): *Type-safe Modular Hash-consing.* In: *Proceedings of the 2006 Workshop on ML*, ML '06, ACM, New York, NY, USA, pp. 12–19, doi:10.1145/1159876.1159880. Available at `http://doi.acm.org/10.1145/1159876.1159880`.

[10] Jacques Garrigue (1998): *Programming with Polymorphic Variants.* In: *Workshop on ML.*

[11] Jacques Garrigue (2000): *Code reuse through polymorphic variants.* In: *In Workshop on Foundations of Software Engineering.*

[12] Jeremy Gibbons (2002): *Calculating Functional Programs*, pp. 151–203. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/3-540-47797-7_5. Available at `https://doi.org/10.1007/3-540-47797-7_5`.

[13] Jeremy Gibbons (2007): *Datatype-generic Programming.* In: *Proceedings of the 2006 International Conference on Datatype-generic Programming*, SSDGP'06, Springer-Verlag, Berlin, Heidelberg, pp. 1–71. Available at `http://dl.acm.org/citation.cfm?id=1782894.1782895`.

[14] Ralf Hinze (2006): *Generics for the Masses. J. Funct. Program.* 16(4-5), pp. 451–483, doi:10.1017/S0956796806006022. Available at `http://dx.doi.org/10.1017/S0956796806006022`.

[15] Graham Hutton (1999): *A Tutorial on the Universality and Expressiveness of Fold. J. Funct. Program.* 9(4), pp. 355–372, doi:10.1017/S0956796899003500. Available at `http://dx.doi.org/10.1017/S0956796899003500`.

[16] Oleg Kiselyov, Simon Peyton Jones & Chung-chieh Shan (2010): *Fun with Type Functions*, pp. 301–331. Springer London, London, doi:10.1007/978-1-84882-912-1_14. Available at `https://doi.org/10.1007/978-1-84882-912-1_14`.

[17] Oleg Kiselyov & Ralf Lämmel (2005): *Haskell's overlooked object system. CoRR* abs/cs/0509027. Available at `http://arxiv.org/abs/cs/0509027`.

[18] Donald E. Knuth (1968): *Semantics of context-free languages. Mathematical systems theory* 2(2), pp. 127–145, doi:10.1007/BF01692511. Available at `https://doi.org/10.1007/BF01692511`.

[19] Dmitry Kosarev & Dmitry Boulytchev (2016): *Typed Embedding of a Relational Language in OCaml.* In: *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016.*, pp. 1–22, doi:10.4204/EPTCS.285.1. Available at `https://doi.org/10.4204/EPTCS.285.1`.

[20] Ralf Lämmel & Simon Peyton Jones (2003): *Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. SIGPLAN Not.* 38(3), pp. 26–37, doi:10.1145/640136.604179. Available at `http://doi.acm.org/10.1145/640136.604179`.

[21] Ralf Lämmel & Simon Peyton Jones (2004): *Scrap More Boilerplate: Reflection, Zips, and Generalised Casts.* In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, ACM, New York, NY, USA, pp. 244–255, doi:10.1145/1016850.1016883. Available at `http://doi.acm.org/10.1145/1016850.1016883`.

[22] Ralf Lämmel & Simon Peyton Jones (2005): *Scrap Your Boilerplate with Class: Extensible Generic Functions. SIGPLAN Not.* 40(9), pp. 204–215, doi:10.1145/1090189.1086391. Available at `http://doi.acm.org/10.1145/1090189.1086391`.

[23] Erik Meijer, Maarten Fokkinga & Ross Paterson (1991): *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire.* Springer-Verlag, pp. 124–144.

[24] Bruno C. d. S. Oliveira & William R. Cook (2012): *Extensibility for the Masses: Practical Extensibility with Object Algebras.* In: *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, Springer-Verlag, Berlin, Heidelberg, pp. 2–27, doi:10.1007/978-3-642-31057-7_2. Available at `http://dx.doi.org/10.1007/978-3-642-31057-7_2`.

[25] François Pottier (2017): *Visitors Unchained. Proc. ACM Program. Lang.* 1(ICFP), pp. 28:1–28:28, doi:10.1145/3110272. Available at `http://doi.acm.org/10.1145/3110272`.

[26] Didier Rémy (2002): *Using, Understanding, and Unraveling the OCaml Language From Practice to Theory and Vice Versa.* In Gilles Barthe, Peter Dybjer, Luís Pinto & João Saraiva, editors: *Applied Semantics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 413–536.

[27] Tillmann Rendel, Jonathan Immanuel Brachthäuser & Klaus Ostermann (2014): *From Object Algebras to Attribute Grammars. SIGPLAN Not.* 49(10), pp. 377–395, doi:10.1145/2714064.2660237. Available at `http://doi.acm.org/10.1145/2714064.2660237`.

[28] Peter Sestoft (2002): *The Essence of Computation.* chapter Demonstrating Lambda Calculus Reduction, Springer-Verlag New York, Inc., New York, NY, USA, pp. 420–435. Available at `http://dl.acm.org/citation.cfm?id=860256.860276`.

[29] Wouter Swierstra (2008): *Data Types à La Carte. J. Funct. Program.* 18(4), pp. 423–436, doi:10.1017/S0956796808006758. Available at `http://dx.doi.org/10.1017/S0956796808006758`.

[30] Marcos Viera, S. Doaitse Swierstra & Wouter Swierstra (2009): *Attribute Grammars Fly First-class: How to Do Aspect Oriented Programming in Haskell.* In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, ACM, New York, NY, USA, pp. 245–256, doi:10.1145/1596550.1596586. Available at `http://doi.acm.org/10.1145/1596550.1596586`.

[31] P. Wadler & S. Blott (1989): *How to Make Ad-hoc Polymorphism Less Ad Hoc.* In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, ACM, New York, NY, USA, pp. 60–76, doi:10.1145/75277.75283. Available at `http://doi.acm.org/10.1145/75277.75283`.

[32] Philip Wadler (1998): *The Expression Problem.*

[33] Leo White, Frédéric Bour & Jeremy Yallop (2015): *Modular implicits. Electronic Proceedings in Theoretical Computer Science* 198, doi:10.4204/EPTCS.198.2.

[34] Jeremy Yallop (2007): *Practical Generic Programming in OCaml.* In: *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, ACM, New York, NY, USA, pp. 83–94, doi:10.1145/1292535.1292548. Available at `http://doi.acm.org/10.1145/1292535.1292548`.

[35] Jeremy Yallop (2017): *Staged Generic Programming. Proc. ACM Program. Lang.* 1(ICFP), pp. 29:1–29:29, doi:10.1145/3110273. Available at `http://doi.acm.org/10.1145/3110273`.

[36] Haoyuan Zhang, Zewei Chu, Bruno C.d. S. Oliveira & Tijs van der Storm (2015): *Scrap Your Boilerplate With Object Algebras.* In: *Proceedings of the Object-oriented Programming, Systems, Languages, and Applications (OOPSLA, 2015)*, New York, United States. Available at `https://hal.inria.fr/hal-01261477`.