Science of Computer Programming 114 (2015) 57–73

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Combinators and type-driven transformers in Objective Caml ☆

Dmitry Boulytchev

*St. Petersburg State University, 198504, Universitetski pr., St. Petersburg, Russia*

ARTICLE INFO

*Article history:*
Received 30 April 2014
Received in revised form 1 July 2015
Accepted 21 July 2015
Available online 29 July 2015

*Keywords:*
Compilers
Functional programming
Datatype-generic programming
Combinators
Modularization and code reuse

ABSTRACT

We describe an implementation of LDTA 2011 Tool Challenge tasks in Objective Caml language. Instead of using some dedicated domain-specific tools we utilize typical functional programming machinery such as polymorphic functions, monads and combinators; in addition we extensively use an idiom of type-driven transformers, which can be considered as a form of datatype-generic programming. Our implementation provides a good example of utilization of Objective Caml specific features such as open and implicitly defined types. As a result we provide a highly modular implementation built up of separately compiled components combined in a type-safe manner.



## 1. Introduction

Objective Caml or, in short, OCaml [1,2] is a high-level programming language which has been developed, distributed and supported by INRIA (Institut National de Recherche en Informatique et en Automatique, France) since 1985. As a member of ML family this language provides a number of well-known statically typed functional programming features such as first-class high-order functions, parametric polymorphism and type inference. In addition OCaml consistently incorporates object-oriented traits — objects and classes with structural subtyping, rich module system with first-class modules and *functors* (that is, modules parameterized by other modules) which makes it a full-fledged multiparadigm programming language. The OCaml programming environment is equipped with syntax extension tools — CamlP4[1]/CamlP5[2] — which allows an end-user to extend the host language with domain-specific constructs.

We argue that in its present state OCaml is already rich enough to be directly used as an implementation formalism for all the tasks listed for the tool challenge. Apart from parsing and pretty-printing which can be easily handled in a standard way using parser- and printer combinators [3–5] the other tasks are rather simple transformations of abstract syntax trees. We utilized the datatype-generic programming approach [6] in the form of *type-driven transformers* to implement most of these transformations. "Type-driven" means that the semantics of transformers is completely determined by a type being transformed so all transformers can be constructed mechanically from type definitions. Another property of transformers is their extensibility: each transformer operates on a data structure of a partially-defined, or *open*, type. Transformers for open types can be combined by a certain primitive to provide a transformer for the union type which still remains open and so can be combined later on. As a result we can construct a highly modular structure of transformers each of which implements some task on a certain trait of some common data structure.

☆ The research was supported by Russian Foundation for Basic Research grant No. 13-01-00506.
*E-mail address:* dboulytchev@math.spbu.ru.
[1] http://brion.inria.fr/gallium/index.php/Camlp4.
[2] http://pauillac.inria.fr/~ddr/camlp5/.

http://dx.doi.org/10.1016/j.scico.2015.07.008
0167-6423/

In our implementation we used type-driven transformers as a programming idiom — all transformers for all types were hand-written. Later we refined this approach and implemented a separate datatype-generic programming tool[3] making it possible to generate the majority of implementation code from type declarations. During this refinement, however, the details have changed significantly, so the implementation we present here is essentially different from that which would be generated by the tool. Since we did not use any generic tool we strictly speaking did not need type declarations. Fortunately, OCaml permits some sort of implicit type definitions — polymorphic variant and object types need not to be declared. We heavily utilized this property — actually there is not a single type definition in our implementation: all types are introduced implicitly and inferred from the transformation functions thus giving us a good example of a typeful programming with no type declarations. In particular, there are no type definitions for abstract syntax trees — their types are inferred by the compiler from the bodies of AST-processing functions.

The rest of the paper is structured as follows: in the next section we briefly describe polymorphic variant and object types in OCaml, then we introduce the idiom of type-driven transformers which is essential for understanding tool challenge tasks implementation description presented in Section 4. Section 5 reviews constructed artifacts and their metrics; the final section concludes.

We would like to thank anonymous reviewers for their comments and remarks which helped us to improve this paper.

## 2. Open and implicitly defined types in OCaml

Our implementation essentially relies on open and implicitly defined types which are supported in OCaml in forms of polymorphic variants and objects. We use implicitly-defined polymorphic variants to represent types of abstract syntax trees. This approach allows us to completely reuse the type of base language AST when we extend it with a new constructs; moreover using type-driven transformers we sum up not only types, but also their transformations.

Polymorphic variant types [7] were introduced in OCaml version 3.0. In short, polymorphic variants make it possible to share the same constructor between different variant types with possibly different types and numbers of arguments. For example, the following code snippet

```
type a = [`A of int | `B of string]
type b = [`A of string | `B of int]
```

is legitimate in OCaml despite these two types sharing the same constructors within the same scope.[4] Like regular variant types polymorphic variants can be matched against patterns, but need not be explicitly declared, so the following fragment

```
let implicit_a = function `A n → n | `B s → int_of_string s
```

typechecks even with no prior type declaration.[5] The value declaration with inferred type, printed by the compiler, is shown below:

```
val implicit_a : [< `A of int | `B of string ] → int
```

Note that the inferred type for the argument of `implicit_a` describes a *subtype* of the polymorphic variant type with constructors `` `A of int `` and `` `B of string ``, i.e. any type with fewer constructors, hence "`[<`" in its type description.

Another feature of polymorphic variant types is that they can be open — only some of their constructors may be known:

```
let opened_a = function
| `A n → n
| `B s → int_of_string s
| _    → 0
val opened_a : [> `A of int | `B of string ] → int
```

Here the inferred type for the argument of `opened_a` corresponds to any *supertype* of the polymorphic variant type with constructors `` `A of int `` and `` `B of string ``, hence "`[>`"; thus, function `opened_a` can be applied to values of polymorphic variant types with a wider set of constructors besides `` `A `` and `` `B ``.

Another interesting feature of polymorphic variants is that they can be summed up: if `t1` and `t2` are two polymorphic variant types, then `[ t1 | t2 ]` is their sum (provided that their constructors do not "contradict" each other). Moreover, this sum can later be decomposed into its counterparts using the pattern-matching against special patterns:

```
match x with #t1 as a → ··· | #t2 as b → ··
```

Here `x` has type `[ t1 | t2]`, `a` — type `t1`, `b` — type `t2`.

It is worth mentioning that recursive functions can implicitly introduce recursive polymorphic variant types. For example, for the argument of the following function (which converts Peano-encoded naturals to regular integer values) a recursive type is inferred:

[3] https://github.com/dboulytchev/generic-transformers.

[4] Polymorphic variant type constructors are lexically distinguished from the regular ones by the backquote as their first character.

[5] In OCaml, keyword "`function`" can be used to define a function with one argument by case analysis — the single argument is matched against a sequence of patterns, specified in the definition.