Full Title*

Subtitle†

ANONYMOUS AUTHOR(S)

Text of abstract ....

Additional Key Words and Phrases: datatype-generic programming, object-oriented programming, code reuse

1 SORTS OF TYPES

```
type (a1, ..., an) typename =
  (* primitive ones *)
  | int | string | ...
  (* named aliases *)
  | type_ as 'identifier
  (* constructors of algebraic types *)
  | C1 of t11 * ... * t1n1 | ... | Ck of tk1 * ... * tknk
  (* tuples can be viewed as specific predefined constructors *)
  | (t1, ..., tk)
  (* polymorphic variants with concrete constructors  *)
  | [ `C1 of t11 * ... * t1n1 | ... | `Ck of tk1 * ... * tknk ] | [ (t1,...,tl) u ]
  (* union of polymorphic variants *)
  | [ (t1,...,tl) τ | ... ]
  (* type abbrevations (a.k.a. type aliases) : application of types to other types *)
  | (t1, ..., tn) τ
```

Current limitations:

- Using object types, module types and functional types in type declarations.
- Constraints on type declrations are not being taken to account
- GADTs are treated as simple algebraic data types.
- ADT with record constructors.
- Generated code will typecheck only for **regular** type declarations.
- Type declarations with `nonrec` keyword.

2 Какие объявления генерируются по объявлению типа

Для каждого объявления типа `type` $(a_1, ..., a_n)$ `t` генерируются в файле реализации:

- *Interface class* который определяет тип всех трансформаций, которые могут быть применены для значения типа `t`.
- *Generic catamorphism*, который позволяет применить конкретную трансформацию к значению типа `t`.

*Title note
†Subtitle note

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/

А также два соответсвующих объявления в файле интерфейса.

Для каждого вида трансформации типа type $(a_1, ..., a_n)$ t в файле реализации генерируется:

- *Класс трансформации*, который наследует interface class, принимает функции-трансформации аргементов в момент конструирования и определяет как именно осуществляется трансформация для значения типа type $(a_1,...,a_n)$ t.
- Функция трансформации для type $(a_1, ..., a_n)$ t, которая принимает функции-трансформации аргументов, инстанциирует класс трансформации и применяет трансформацию.

Аналогично, два объявления в файле интерфейса.

Чтобы сгенерировать реализицию некоторого вида трансформации для type $(a_1, ..., a_n)$ t, необходимо в пространстве имениметь трасформации для всех типов, которые входят в RHS объявления типа. Если же объявление типа является type abbrevation (т.е. type $(a_1, ..., a_n)$ t $= (b_1, ..., b_m)$ τ или для полиморфных вариантов type $(a_1, ..., a_n)$ t $=$ [$(b_1,$ то, дополнительно необходимо иметь класс трансформации для типа τ.

Все классы трансформации принимают (n+1) параметров: одна функция трансформации рекрсивного вхождения объявления типа (здесь мы сужаем множество поддерживаемых определений типов только до регулярных) и функции-трансформации для типовых параметров. Для процесса генерации является существенным, чтобы все классы, определяющие трансформацию, имели сходную сигнатуру, выводимую только из имени типа и количества его параметров. Поэтому, функцию-трансформацию рекурсивного вхождения типа нельзя опустить для нерекурсивных типов.

Для взаимно-рекурсивных определений типов затруднительно иметь трансформацию для всех объявляемых типов в момент объявления класса трансформации (OCaml не поддерживает взаимно-рекурсивные определения между классами и функциями). Поэтому в файле реализации для каждого объявления типа генерируются два класса, где первый имеет дополнительные параметры для функций-трансформаций взаимно-рекурсивных определений. После определения функций-трансформаций для объявляемых типов генерируются ещё несколько классов, которые наследуют предыдущие определения, конкретизируя функции-трансформации для объявляемых взаимно-рекурсивных типов. Таким образом, все сгенерированные классы для взаимно-рекурсивных определений имеют такую же форму в сигнатуре, как и классы для одиночных определений типов.

В GT также поддерживаются объявление полиморфных вариантов, которые оказывают влияние на форму сигнатуры всех генерируемых классов. Для поддержки возможности объединения полиморфных вариантов при объявлении типа все классы имеют дополнительный параметр, который является свободной типовой переменной в случае класса для обычных объявлений типов, и является *открытой* версией типа если класс соответствует полиморфному вариантному типу. Также, полиморфный вариантный тип требует дополнительных аннотаций открытости при объявлении сигнатуры функции-трансформации для трасформации вида gmap (a.k.a. functor).

3 Что конкретно генерируется для разных объявлений типов

type $(a_1, ..., a_n)$ t $=$ ....

- В *interface class* для типа t методы соответсвуют конструкторам агебраического типа или полиморфного варианта.

```
class virtual [ class arguments ] class_t = object
  (* either *)
```

```
    (** for every constructor (of normal or poly variant) called
      * C_i of t_i1 * ... * t_in_1
      *)
    (* method virtual c_C_i: 'inh → t_i1 → ... → t_in → 'syn*)
    method virtual c_C_i: 'inh → curried constructor arguments → 'syn
end
```

Для type abbrevation `type` $(a_1, ..., a_n)$ `t` $= (t_1, ..., t_n)\ \tau$ используется наследование, чтобы объявить класс :

```
class virtual [ class arguments ] class_t = object
  inherit [with application−specific arguments] class_τ
end
```

For every n-parametric type ($n >= 0$) there are $3 * n + 2 + 1$ *class arguments*. They are:

$$\overline{'a_i,'ia_i,'sa_i}, 'inh, 'syn, 'extra$$

where
– a_i is an i-th paramter of the type being processed;
– sa_i is a synthesized attribute for type parameter a_i;
– ia_i is an inherited attribute for type parameter a_i;
– $'syn$ is a synthesized attribute for the whole type.
– $'extra$ это открытый полиморфный вариант, который сейчас объявляется, либо wildcard для остальных объявлений типов.
Although there some differences for type declarations that involve mutal recursion.

- *Generic catamorphism* $gcata_t$ (для объявления типа t) который принимает значение типа t и применяет трансформацию. It is defined using pattern-matching for the variant types or as a composition with generic catamorphism $gcata_\tau$ if type t is constructed as an application of type τ.
- Для каждого вида трансформации tr concrete класс с реализацией этой трансформации:

```
class [ tr class arguments ] tr_t self fa_1 ... fa_n = object
  inherit [ inherited class arguments ] class_t
  (* implementation of virtual methods if any *)
end
```

where
– transformation $self$ is a current transformation; class is defined in open recursion style and will receive it after tying the knot
– fa_i of type $ia_i \to a_i \to sa_i$ are transformation functions for type parameters
– sa_i is a synthesized attribute for type parameter a_i
– ia_i is an inherited attribute for type parameter a_i
– $'syn$ is a synthesized attribute for the whole type

- For every transformation method tr a function which takes transformations fa_i for every type parameter a_i, and a value of type t that will be transformed

4 CONSTRUCTION BLOCKS FOR A TRANSFORMATION FUNCTION

A class declaration for transformation method requires transformations of for parameters of type $(a_1, ..., a_n)t$ that is being processed; these transformation functions will be passed as arguments of class's constructor.

For all types that was used in declaration of type t we suppose that transformation functions are present in the scope where the type t is declared. As we declare class for transformation object just after type declaration, these functions will be in scope of type declaration too.

For recursive type declarations it is required to call a transformation function for this type inside the class that represents a transformation and will be used to get this function. OCaml language doesn't allow declaring mutally recusive functions and classes, so we break the recursion by explicitly passing self-transformation as first of constructor's arguments. More precisely, if we have a recursive type

```
type ('a, 'b) alist = ...
type 'a list = ('a, 'a list) alist

class ['a, 'b] show_alist : 'self → (unit → 'a → string) →
                                    (unit → 'b → string) →
  object
    method c_Cons : unit → 'a → 'b → string
    method c_Nil  : unit → string
  end
```

we add extra constructor argument `self` to the declaration of class

```
class ['a] show_list self fa = object
  inherit ['a, unit, string, unit, string] class_list
  inherit ['a, 'a list] show_alist "anything" fa self
end
```

and use this function to specify a transformation of second argument during instantiation of the class `show_alist`.

For an example with mutal recursion see demo05 in the repository.

5 CREATING TRANSFORMATIONS FOR TYPE ALIASES

Let us have

```
type (a_1, ..., a_m)τ = ...
class virtual ['a_i,'sa_i,'ia_i, 'inh, 'syn] class_τ = object .. end
```

and want to generate transformations for the type

```
type (a_1, ..., a_n)t = (b_1, ..., b_m)τ as 'smth
```

which is a *type alias* of type τ with m type arguments $b_1 ... b_m$. The term `as 'smth` which constraints type variable `'smth` to be equal to left hand side of type declaration is optional.

Values of type t can be constructed using the constructors from type τ, so the generic catamorphism will be the same:

```
let rec gcata_t tr inh x = gcata_τ tr inh x
```

Where `tr` is a transformation object, `inh` is an initial inherited attribute and `x` is the value being transformed.

Implementation of an interface class for `t` is just an inheritance of τ's interface class with right type parameters and arguments. We assume below that all types used on right hand side of the declaration of type `t` are available in the scope where we declare a class, although it could be not true (see **??** for details). The τ's interface class has a structure like

```
class ['a_i,'sa_i,'ia_i, 'inh, 'syn] class_τ = object..end
```

and derived interface class will have these implementation

```
class ['a_i,'sa_i,'ia_i, 'inh, 'syn] class_t = object
    inherit [ (3*m) type parameters, 'inh, 'syn] class_τ
      transformation function for self
      ...
      m transformation functions for type parameters
end
```

Let's deal with type parameters which are: `m` normal type parameters, `m` types of inherited attributes, `m` types of synthesized attributes.

- Every i-th normal type paramater is just a type which was used as i-th argument of τ in t declaration, i.e. b_i.
- Every i-th inherited type parameter is a b_i where every a_i is replaced by associated type inherited attribute: $b_i['ia_i \leftarrow 'a_i]$.
- The same logic is applied to synthesized attributes: $b_i['sa_i \leftarrow 'a_i]$.

We are also able to construct transformation functions for every b_i:

- For all named types we suppose that good transformation function with right arity is in scope.
- For allpolymorphic type variables we will use transformation function passed as constructor's arguments for this class.
- There is no polymorphic type parameters that doesn't have transformation function because it contradicts typechecking process of type declaration because Haskell-style existentional types has another syntax in OCaml.

6 FOUR WAYS TO ADD TYPES TO VISITOR CLASSES

Виды типизации методов класса:

(1) Monomorphic approach (visitors)

```
method virtual visit_'a : _
method visit_container :
  'env . 'env → 'a container → ...
```

- Pros: Не нужно генерировать аннотации типов, что упрощает генерацию.
- Cons: Некомпозициональны (ссылка на главу у Потье)

(2) Polymorphic approach (visitors)

```
method visit_container :
  'env 'a .
    ('env → 'a → ...) →
    'env → 'a container → ...
```

- Pros: композиционность, в отличие от (1).
- Pros: Функции, передаваясь в каждый метод явно, позволяют поддержать нерегулярные типы. (Хотя остается вопрос, зачем тогда универсальная квантификация)
- Cons: Нужны явные аннотации типов.
- Cons: Не поддерживаются для fold в библиотеке visitors (похоже там что-то с типами, в никаких статьях про это вроде не писалось).
- Cons: В некоторых случаях генерируют код, не проходящий проверку типов

```
type ('a,'b) t = Foo of 'a * 'b (* OK *)
type 'a t2 = ('a, int) t (* compilation error in generated code *)
```

  или пример из руководства по библиотеке (глава 3.1 из [**?**])

```
type expr = E of expr oexpr [@@unboxed]
[@@deriving visitors { variety = "map"; polymorphic = true }]
```

  Для обеих примеров генерируемый код можно исправить руками, потерев универсальные квантификации методов

(3) Polymorphic apprach without universal quantification of methods

```
method visit_container :
    ('env → 'a → ...) →
    'env → 'a container → ...
```

- Pros: Два примера выше начинают работать.
- Cons: Нельзя выразить gmap через fold (ссылка на главу Потье).

(4) Подход библиотеки Generic Transformes: функции-трансформации для аргументов передаются явно при создании класса.

- Cons: не поддерживаются нерегулярные типы
- Можно выразить gmap через fold ссылка на код.

Виды типизации самого класса:

(1) À la Потье

```
class ['self] classname = object (self: 'self) ... end
```

- Pros: Кратко: только один типовый параметр
- Cons: Невозможно сгенерировать сигнатуру в файле интерфейса
- Cons: Не получится поддержать объединение полиморфных вариантов: там требуются аннотоации открытости (для этого у нас используется дополнительный параметр) которые в данном подходе нет возможности вставить.

(2) À la GT

```
class ['a_i,'ia_i,'sa_i, 'inh, 'syn,'extra] classname = object ... end
```

- Cons: Длиннее
- Pros: Сигнатура генерируется
- Pros: Объединение полиморфных вариантов работает.

7 Техничесие особенности GT

Возможно, надо будет привести пример, что если совсем хочется использовать неподходящие типы, то можно абстрагироваться до:

```
type ('a,'b) t = ('a → 'b) τ
```

И показать пример синтаксиса как указать кастомную функцию-трансформацию для преобразования типового параметра τ.

8 OPEN QUESTIONS

- What is generic catamorphism for `type 'a antiphantom = 'a`?

Text of appendix ...