

Экстракция кода из Agda в Haskell

Шабалин Александр

научный руководитель
доц. Москвин Д. Н.

Академический университет
2014 г.

Мотивирующий пример

```
elemAt :: [a] → Int → a
```

```
elemAt [] 3 = ???
```

```
elemAt :: [a] → Int → Maybe a
```

```
elemAt [] 3 = Nothing
```

```
elemAt :: (xs :: [a]) → (n :: Int) →
```

```
    (n < length xs) → a
```

```
elemAt [] 3 (???) = ...
```

Зависимые типы

System F:

Term ::= **Var** | $\lambda x. \text{Term}(x)$ | **Term** **Term** | (**Term**, **Term**)

Type ::= **TVar** | **Type** \rightarrow **Type** | $\forall x. \text{Type}(x)$ | **Type** \times **Type**

$\Gamma \vdash \text{Term} : \text{Type}, \quad \Gamma = \text{Var} : \text{Type}, \dots$

Зависимые типы

System F:

Term ::= **Var** | $\lambda x. \text{Term}(x)$ | **Term** **Term** | (**Term**, **Term**)

Type ::= **TVar** | **Type** \rightarrow **Type** | $\forall x. \text{Type}(x)$ | **Type** \times **Type**

$\Gamma \vdash \text{Term} : \text{Type}, \quad \Gamma = \text{Var} : \text{Type}, \dots$

Зависимые типы:

Term ::= **Var**

| **Term** **Term**

| $\lambda x. \text{Term}(x)$ | $(x : \text{Term}) \rightarrow \text{Term}(x)$

| (**Term**, **Term**) | $(x : \text{Term}) \times \text{Term}(x)$

| **Set**

$\Gamma \vdash \text{Term} : \text{Term}, \quad \Gamma = \text{Var} : \text{Term}, \dots$

Язык с зависимыми типами и синтаксисом, похожим на Haskell.

Agda - примеры

data работает аналогично *GADT* в *Haskell*

```
data List (A : Set) : Set where
```

```
  <> : List A
```

```
  _::_ : A → List A → List A
```

{...} — необязательный аргумент (компилятор сам подставит)

```
list-length : {A : Set} → List A → Nat
```

```
list-length <> = 0
```

```
list-length (x :: xs) = list-length xs + 1
```

Agda - примеры

```
data Vec (A : Set) : Nat → Set where  
  nil : Vec A 0  
  cons : {n : Nat} → A → Vec A n → Vec A (n + 1)  
  
list-to-vec : {A : Set} → (xs : List A) →  
  Vec A (list-length xs)  
list-to-vec <> = nil  
list-to-vec (x :: xs) = cons x (list-to-vec xs)
```

Agda - примеры

$\{a1\ a2 : A\}(b : B) \rightarrow C$ — синтаксический сахар для

$\{a1 : A\} \rightarrow \{a2 : A\} \rightarrow (b : B) \rightarrow C$

$\text{zip-vec} : \{A\ B : \text{Set}\} \{n : \text{Nat}\} \rightarrow \text{Vec}\ A\ n \rightarrow \text{Vec}\ B\ n \rightarrow$
 $\text{Vec}\ (A \times B)\ n$

$\text{zip-vec}\ \text{nil}\ \text{nil} = \text{nil}$

$\text{zip-vec}\ (\text{cons}\ x\ xs)\ (\text{cons}\ y\ ys) =$
 $\text{cons}\ (x\ ,\ y)\ (\text{zip-vec}\ xs\ ys)$

$\text{zip-vec}\ \text{nil}\ (\text{cons}\ y\ ys) = \dots$

$\text{zip-vec}\ (\text{cons}\ x\ xs)\ \text{nil} = \dots$

эти 2 клоза даже написать нельзя - типы не сойдутся


```

data _<_ : Nat → Nat → Set where
  leq-zero : {n : Nat} → (0 < n + 1)
  leq-suc   : {n m : Nat} → (n < m) → (n + 1 < m + 1)

elemAt : {A : Set}(xs : List A)(n : Nat) →
  (n < list-length xs) → A
elemAt <> _ ()
elemAt (x :: xs) 0 _ = x
elemAt (_ :: xs) (n + 1) (leq-suc p) = elemAt xs n p

```

Компилятор MAlonzo

```
data Vec (A : Set) : Nat → Set where
  nil : Vec A 0
  cons : {n : Nat} → A → Vec A n → Vec A (n + 1)
vec-map : {A B : Set}{n : Nat} → (A → B) →
  Vec A n → Vec B n
vec-map f nil = nil
vec-map f (cons x xs) = cons (f x) (vec-map f xs)

data T1 a0 a1 a2 = C2 | C3 a0 a1 a2
d4 v0 v1 v2 v3 (C2) = cast C2
d4 v0 v1 v2 v3 (C3 v4 v5 v6) = cast (C3 (cast v4) ...)
ghci>:t d4
d4 :: a → a1 → a2 → a3 → (T1 t t1 t2) → b
```

```
data Vec a = Nil | Cons Nat a (Vec a)
```

```
vecMap :: Nat → (a1 → a2) → Vec a1 → Vec a2
```

```
vecMap n f v =
```

```
  case v of
```

```
    Nil → Nil
```

```
    Cons n0 x xs → Cons n0 (f x) (vecMap n0 f xs)
```

Что хочется

Идея

Нужно сохранить инварианты, поддерживаемые системой типов.

1. Экспортируем весь код в Haskell с помощью MAlonzo.
2. Для выбранных имен пытаемся найти соответствующие типы на Haskell и генерируем обертки над скомпилированным кодом.

Простое решение

Утверждение

Типы Agda, лежащие в System F, представимы в Haskell и имеют ту же семантику.

Формально не доказывал, но, вроде, очевидно.

Указание на «экспорт» дается с помощью прагм

```
{-# EXPORT agda-name haskell-name #-}
```

- ▶ **data** T (A : Set) : Set → Set **where** породит
newtype T a0 a1 = <hidden>
- ▶ $f : \{A : \text{Set}\} \rightarrow (\{B : \text{Set}\} \rightarrow B) \rightarrow A$ породит
 $f :: \forall a0. (\forall b0. b0) \rightarrow a0$

Недоработки

Некоторые типы можно экспортировать вместе с конструкторами:

```
{-# EXPORT_DATA agda-name hs-name hs-con-name ... #-}
```

Проблема оборачивания функций:

- ▶ **newtype** гарантирует, что внутреннее представление совпадает с оборачиваемым типом \Rightarrow `unsafeCoerce`.
- ▶ **data** не может дать никаких похожих гарантий \Rightarrow генерирование разбора по конструкторам.

Из этого же — невозможно экспортировать `f`:

```
newtype T1 a0 = ...
```

```
data T2 a0 = ...
```

```
f : {A : Set}  $\rightarrow$  T1 (T2 A)  $\rightarrow$  A
```

```
f ::  $\forall a$ . T1 (T2 a)  $\rightarrow$  a
```

Недоработки

- ▶ Haskell поддерживает **data** $T (a :: * \rightarrow *)$ аналог в Agda **data** $T (A : \text{Set} \rightarrow \text{Set})$
- ▶ У Agda есть FFI в Haskell и есть набор встроенных типов. При компиляции они обрабатываются специально.
- ▶ В Agda есть параметризованные модули:
module Main (A : Set) **where**
- ▶ Сейчас все экспорты генерируются в том же модуле, что и скомпилированный код(MAlonzo.Code.Module.Name.
Лучше все это вынести в отдельное поддереву
MAlonzo.Export.Module.Name, чтобы экспортировать
newtype абстрактно(скрыв реализацию от пользователя) и
чтобы автоматически можно было сгенерировать
документацию без мусора.

Что еще можно попробовать

На Haskell можно проэмулировать некоторые зависимые типы

1. Если n не используется внутри `vec-zip`, то

$$\begin{aligned} \text{vec-zip} : \{A\ B : \text{Set}\} \{n : \text{Nat}\} &\rightarrow \text{Vec}\ A\ n \rightarrow \\ &\text{Vec}\ B\ n \rightarrow \text{Vec}\ (A \times B)\ n \end{aligned}$$

эмулируется без проблем.

2. есть трюк, позволяющий эмулировать типы вроде

$$\text{replicate} : \{A : \text{Set}\} (n : \text{Nat}) \rightarrow A \rightarrow \text{Vec}\ A\ n$$

3. ...

Сложности

- ▶ В Agda все функции должны быть полными и завершающимися.

$$f : \{A\ B : \text{Set}\}(g : A \rightarrow B)(\text{proof} : \text{Bijection } g) \rightarrow B \rightarrow A$$

Что делать, если мы передадим g , которая зацикливается?

- ▶ На самом деле в Agda могут быть не завершающиеся функции с использованием коиндукции.

Q&A