

# Экстракция кода из Agda в Haskell

Шабалин Александр

научный руководитель  
доц. Москвин Д. Н.

Академический университет  
2014 г.

# Мотивирующий пример

TODO: Пример, который легко ломается в хаскеле и легко чинится зависимыми типами.

# Зависимые типы

System F:

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

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

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

# Зависимые типы

System F:

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

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

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

Зависимые типы:

**Term** ::= **Var**

| **Term** **Term**

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

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

| **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**

$\langle \rangle : \text{List } A$

$\_ :: \_ : A \rightarrow \text{List } A \rightarrow \text{List } A$

$\{ \dots \}$  — необязательный аргумент (компилятор сам подставит)

$\text{list-length} : \{ A : \text{Set} \} \rightarrow \text{List } A \rightarrow \text{Nat}$

$\text{list-length } \langle \rangle = 0$

$\text{list-length } (x :: xs) = \text{list-length } xs + 1$

## 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

$\{a_1\ a_2 : A\}(b : B) \rightarrow C$  — синтаксический сахар для  
 $\{a_1 : A\} \rightarrow \{a_2 : A\} \rightarrow (b : B) \rightarrow C$

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

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

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

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

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

эти 2 клоза даже написать нельзя - типы не сойдутся



TODO: Пример с первого слайда

# Компилятор MAlonzo

**data**  $Vec\ (A : Set) : Nat \rightarrow Set$  **where**

$nil : Vec\ A\ 0$

$cons : \{n : Nat\} \rightarrow A \rightarrow Vec\ A\ n \rightarrow Vec\ A\ (n + 1)$

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

$vec\text{-}map\ f\ nil = nil$

$vec\text{-}map\ f\ (cons\ x\ xs) = cons\ (f\ x)\ (vec\text{-}map\ f\ xs)$

**data**  $T_1\ a_0\ a_1\ a_2 = C_2 \mid C_3\ a_0\ a_1\ a_2$

$d_4\ v_0\ v_1\ v_2\ v_3\ (C_2) = cast\ C_2$

$d_4\ v_0\ v_1\ v_2\ v_3\ (C_3\ v_4\ v_5\ v_6) = cast\ (C_3\ (cast\ v_4)\ (cast\ (v_3\ (cast\ v_5))))$   
 $(cast\ (d_4\ (cast\ v_0)\ (cast\ v_1)\ (cast\ v_4)\ (cast\ v_3)\ (cast\ v_6))))$

**ghci**> :t d4

$d_4 :: a \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow (T_1\ t\ t_1\ t_2) \rightarrow b$

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

*vecMap* :: *Nat* → (*a*<sub>1</sub> → *a*<sub>2</sub>) → *Vec* *a*<sub>1</sub> → *Vec* *a*<sub>2</sub>

*vecMap* *n f v* =

**case** *v* **of**

Nil → Nil

Cons *n*<sub>0</sub> *x xs* → Cons *n*<sub>0</sub> (*f x*) (*vecMap* *n*<sub>0</sub> *f xs*)

# Что хочется

Нужно сохранить инварианты, поддерживаемые системой типов.

## Идея

1. Компилируем 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**  $T (A : Set) : Set \rightarrow Set$  **where** породит **newtype**  $T a_0 a_1 = \langle \text{hidden} \rangle$
- ▶  $f : \{A : Set\} \rightarrow (\{B : Set\} \rightarrow B) \rightarrow A$  породит  $f :: \forall a_0. (\forall b_0. b_0) \rightarrow a_0$

# 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**  $T_1$   $a_0 = \dots$

**data**  $T_2$   $a_0 = \dots$

$f : \{A : Set\} \rightarrow (T_1(T_2 A)) \rightarrow A$

$f :: \forall a. T_1(T_2 a) \rightarrow a$

# Недоработки

- ▶ Haskell поддерживает **data**  $T (a :: * \rightarrow *)$  аналог в Agda **data**  $T (A : Set \rightarrow Set)$
- ▶ 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.

## А что можно попробовать

1. разрешить экспортировать типы с конструкторами
2. зависимые типы вроде  $f : \{A\ B : Set\} \{n : Nat\} \rightarrow Vec\ A\ n \rightarrow Vec\ B\ n \rightarrow Vec\ (A \times B)\ n$  могут быть экспортированы, если  $n$  не используется внутри  $f$
3. есть трюк, позволяющий экспортировать типы вроде  $f : \{A : Set\} (n : Nat) \rightarrow A \rightarrow Vec\ a\ n$



# Gotchas

- ▶ termination checking, totality

# Q&A