

Экстракция кода из Agda в Haskell

Шабалин Александр

научный руководитель
доц. Москвин Д. Н.

Академический университет
2013 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** | $\forall x. \text{Type}(x)$

Зависимые типы

System F:

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

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

Зависимые типы:

Term ::= **Var**

| **Term** **Term**

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

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

| **(Term, Term)**

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

Язык с зависимыми типами и синтаксисом, похожим на Haskell.

Agda - примеры

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

```
  <> : List A
```

```
  _::_ : A → List A → List A
```

```
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*)

zip-vec : {*A B* : *Set*} {*n* : *Nat*} → *Vec* *A* *n* → *Vec* *B* *n* → *Vec* (*A* × *B*) *n*

zip-vec *nil* *nil* = *nil*

zip-vec (*cons* *x* *xs*) (*cons* *y* *ys*) = *cons* (*x*, *y*) (*zip-vec* *xs* *ys*)

zip-vec *nil* (*cons* *y* *ys*) = ...

zip-vec (*cons* *x* *xs*) *nil* = ...

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

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

Inductive $\text{vec} (A : \text{Set}) : \text{nat} \rightarrow \text{Set} := \text{nil} : \text{vec } A \ 0$

| $\text{cons} : \forall n : \text{nat}, A \rightarrow \text{vec } A \ n \rightarrow \text{vec } A \ (n + 1).$

Fixpoint $\text{vMap} (A : \text{Set})(B : \text{Set})n(f : A \rightarrow B)(v : \text{vec } A \ n) : \text{vec } B \ n :=$

match v **with** $\text{nil} \Rightarrow \text{nil}$ $_$

| $\text{cons } _ \ x \ xs \Rightarrow \text{cons } _ \ (f \ x) (\text{vMap } _ \ f \ xs)$

end.

data $\text{Vec } a = \text{Nil} \mid \text{Cons } \text{Nat } a (\text{Vec } a)$

$\text{vMap} :: \text{Nat} \rightarrow (a_1 \rightarrow a_2) \rightarrow \text{Vec } a_1 \rightarrow \text{Vec } a_2$

$\text{vMap } n \ f \ v = \text{case } v \text{ of}$

$\text{Nil} \rightarrow \text{Nil}$

$\text{Cons } n_0 \ x \ xs \rightarrow \text{Cons } n_0 \ (f \ x) (\text{vMap } n_0 \ f \ xs)$

Что хочется

Идея: Экспортировать только то, что при преобразовании типов не потеряет информацию(инварианты).

Сначала разрешаем только типы, которые имеют полный аналог в Haskell. Потом добавляем отдельные случаи зависимых типов, представляемых в Haskell.

Что сделано

С помощью прагм `{-# EXPORT agda-name haskell-name #-}` можно экспортировать типы и функции (тип которых выводится автоматически).

Но типы экспортируются только как абстрактные (`newtype`), а не с конструкторами (`data`).

TODO: go into detail here

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 $f : \{A : Set\} (n : Nat) \rightarrow A \rightarrow Vec\ a\ n$

Q&A