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 :: (xs :: [a]) → (n :: Int) →
   (n < (length xs)) → a
elemAt [] 3 (???) = ...
```

Зависимые типы

System F:

$$\mathbf{Term} ::= \mathbf{Var} \mid \lambda x.\ \mathbf{Term}(x) \mid \mathbf{Term}\ \mathbf{Term} \mid (\mathbf{Term},\ \mathbf{Term})$$

$$\mathbf{Type} ::= \mathbf{TVar} \mid \mathbf{Type} \to \mathbf{Type} \mid \forall x.\ \mathbf{Type}(x) \mid \mathbf{Type} \times \mathbf{Type}$$

$$\boldsymbol{\Gamma} \vdash \mathbf{Term} : \mathbf{Type}, \quad \boldsymbol{\Gamma} = \mathbf{Var} : \mathbf{Type}, \ldots$$

Зависимые типы

System F:

$$\mathbf{Term} ::= \mathbf{Var} \mid \lambda x.\ \mathbf{Term}(x) \mid \mathbf{Term}\ \mathbf{Term} \mid (\mathbf{Term},\ \mathbf{Term})$$
$$\mathbf{Type} ::= \mathbf{TVar} \mid \mathbf{Type} \to \mathbf{Type} \mid \forall x.\ \mathbf{Type}(x) \mid \mathbf{Type} \times \mathbf{Type}$$

$$\mathbf{\Gamma} \vdash \mathbf{Term} : \mathbf{Type}, \quad \mathbf{\Gamma} = \mathbf{Var} : \mathbf{Type}, \ldots$$

Зависимые типы:

$$\begin{aligned}\mathbf{Term} ::= {} & \mathbf{Var} \\ & \mid \mathbf{Term}\ \mathbf{Term} \\ & \mid \lambda x.\ \mathbf{Term}(x) \mid (x : \mathbf{Term}) \to \mathbf{Term}(x) \\ & \mid (\mathbf{Term}, \mathbf{Term}) \mid (x : \mathbf{Term}) \times \mathbf{Term}(x) \\ & \mid \mathbf{Set}\end{aligned}$$

$$\mathbf{\Gamma} \vdash \mathbf{Term} : \mathbf{Term}, \quad \mathbf{\Gamma} = \mathbf{Var} : \mathbf{Term}, \ldots$$

Agda

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) → C` — *синтаксический сахар для*

```
    {a1 : A} → {a2 : A} → (b : B) → C
```

```
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 = ...
```

эти 2 клоза даже написать нельзя - типы не сойдутся

Agda

```
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
```

Coq

```
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 : Set} → ({B : Set} → B) → A` породит
  `f :: ∀a0. (∀b0. b0) → 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} → T1 (T2 A) → A
f :: ∀a. T1 (T2 a) → a
```

Недоработки

- Haskell поддерживает **data** `T (a :: * → *)` аналог в Agda **data** `T (A : Set → 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 поддерево `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 `f`, то

   $$\texttt{f : \{A B : Set\}\{n : Nat\}} \to \texttt{Vec A n} \to \texttt{Vec B n} \to \texttt{Vec (A} \times \texttt{B) n}$$

   эмулируется без проблем.
2. есть трюк, позволяющий эмулировать типы вроде

   $$\texttt{f : \{A : Set\}(n : Nat)} \to \texttt{A} \to \texttt{Vec A n}$$

3. ...

Сложности

- В Agda все функции должны быть полными и завершающимися.

  ```
  f : {A B : Set}(g : A → B)(proof : Bijection g) →
      B → A
  ```

  Что делать, если мы передадим g, которая зацикливается?
- На самом деле в Agda могут быть не завершающиеся функции с использованием коиндукции.

Q&A