

Учреждение Российской Академии наук  
Санкт-Петербургский академический университет –  
Научно-образовательный центр нанотехнологий РАН

На правах рукописи

Диссертация допущена к защите  
Зав. кафедрой

---

«        » \_\_\_\_\_ 2014 г.

Диссертация  
на соискание ученой степени  
магистра

Тема «Экстракция кода из Agda в Haskell»

Направление: 010600.68 — Прикладные математика и физика

Магистерская программа: «Математические и информационные  
технологии»

Выполнил студент

Шабалин А. Л.

Руководитель

к.ф.-м.н, доцент

Москвин Д. Н.

Рецензент

???, ???

Малаховски Я. М.

Санкт-Петербург  
2014

# Содержание

<b>1</b>	<b>Введение</b>	<b>2</b>
1.1	Haskell и Agda . . . . .	2
1.2	TODO: Зависимые типы? . . . . .	2
1.3	Экстракция кода . . . . .	2
1.4	Применение экстракции . . . . .	2
<b>2</b>	<b>Постановка задачи</b>	<b>3</b>
2.1	Цель . . . . .	3
2.2	Существующие решения . . . . .	4
2.2.1	Для Coq . . . . .	4
2.2.2	Для Agda . . . . .	6
2.3	Анализ MAlonzo . . . . .	8
2.4	Задачи . . . . .	8
<b>3</b>	<b>Реализация</b>	<b>9</b>
3.1	Генерирование ограниченного интерфейса . . . . .	9
3.2	Встраивание в MAlonzo . . . . .	9
3.3	Выполняемая кодогенерация . . . . .	10
3.3.1	Типы данных . . . . .	10
3.3.2	Функции . . . . .	10
<b>4</b>	<b>Заключение</b>	<b>11</b>
4.1	Выводы . . . . .	11
4.2	Дальнейшая разработка . . . . .	11
<b>5</b>	<b>Список литературы</b>	<b>12</b>
<b>A</b>	<b>Формальное определение трансформаций</b>	<b>13</b>
<b>B</b>	<b>Доказательство корректности</b>	<b>14</b>

# 1 Введение

## 1.1 Haskell и Agda

Haskell<sup>1</sup> — функциональный язык программирования общего назначения.

Agda<sup>2</sup> — функциональный язык программирования с зависимыми типами и, одновременно, — система компьютерного доказательства теорем.

## 1.2 TODO: Зависимые типы?

## 1.3 Экстракция кода

Термин «экстракция программ» пришел из языка/системы доказательства теорем Coq<sup>3</sup>, похожего на Agda, и означает генерацию функционального кода из доказательств [1].

## 1.4 Применение экстракции

Можно выделить 2 основных причины для реализации механизма экстракции:

### 1. Техника генерирования верифицированных библиотек

На системах с зависимыми типами вроде Agda и Coq можно строить сложные логические утверждения, которые будут проверяться на этапе проверки типов (за счет чего эти системы помогают формально доказывать теоремы). Таким образом, можно написать библиотеку на таком языке с набором доказанных свойств и после этого сделать экстракцию в язык вроде Haskell или ML, на которых проще писать «реальные» программы.

### 2. Бесплатная компилируемость

Скомпилированный код как правило работает быстрее интерпретации, а умение транслировать код в компилируемый язык освобождает от сложной задачи написания компилятора с нуля.

В этой работе фокус ставится на первый пункт.

---

<sup>1</sup><http://haskell.org>

<sup>2</sup><http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.HomePage>

<sup>3</sup><http://coq.inria.fr>

## 2 Постановка задачи

### 2.1 Цель

Разработать способ вызывать код, написанный на Agda, из Haskell, не нарушая внутренних инвариантов, установленных Agda.

Про сохранение внутренних инвариантов нужно объяснить подробнее. Рассмотрим пример:

```
data Nat : Set where
```

```
  zero : Nat
```

```
  succ : Nat → Nat
```

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

```
  nil : List A
```

```
  cons : A → List A → List A
```

```
length : ∀ {A} → List A → Nat
```

```
length nil = zero
```

```
length (cons _ xs) = succ (length xs)
```

```
data Fin : Nat → Set where
```

```
  finzero : ∀ {n} → Fin (succ n)
```

```
  finsucc : ∀ {n} → Fin n → Fin (succ n)
```

```
elemAt : ∀ {A} (xs : List A) → Fin (length xs) → A
```

```
elemAt nil ()
```

```
elemAt (cons x _) finzero = x
```

```
elemAt (cons _ xs) (finsucc n) = elemAt xs n
```

В этом коде определяются три типа данных: натуральные числа, список и конечные числа (тип *Fin n* имеют числа, меньшие *n*) и 2 функции: длина списка и получение элемента из списка по индексу. Рассмотрим вторую функцию. Она принимает 2 аргумента: список и число, меньшее длины списка. Это гарантирует, что элемент с таким индексом существует. Этот инвариант

используется в самом первом клозе: при попытке написать код для пустого списка, Agda замечает, что нет способа построить терм с типом *Fin zero* и поэтому вместо тела пишется (). А так как система типов гарантирует, что этот клоз не будет вызван, то никакой ошибки на этапе исполнения быть не может.

При экстракции в Haskell хочется сохранить это свойство. Но *elemAt* использует зависимые типы, которые не получится воспроизвести на Haskell. Код, сгенерированный из Agda, будет поддерживать это свойство, но для внешнего кода дать гарантий не получится. Поэтому, необходимо запретить вызов этой функции.

## 2.2 Существующие решения

### 2.2.1 Для Coq

Как было сказано в пункте 1.3 в Coq есть технология «экстракции программ». Но текущая реализация стирает все зависимые типы и код аналогичный 2.1: (TODO: I'm having trouble implementing *elemAt* in Coq)

**Inductive** *Nat* : *Set* :=

| *zero* : *Nat*

| *succ* : *Nat* → *Nat*.

**Inductive** *List* (*A* : *Type*) : *Type* :=

| *nil* : *List A*

| *cons* : *A* → *List A* → *List A*.

**Fixpoint** *length* (*A* : *Type*) (*xs* : *List A*) {*struct xs*} : *Nat* :=

**match** *xs* **with**

| *nil* ⇒ *zero*

| *cons* \_ *xs'* ⇒ *succ* (*length* \_ *xs'*)

**end**.

**Inductive** *Fin* : *Nat* → *Set* :=

| *finzero* : **forall** *n* : *Nat*, *Fin* (*succ n*)

| *finsucc* : **forall** *n* : *Nat*, *Fin n* → *Fin* (*succ n*).

**Lemma** *emptyfin* : **forall** *f* : *Fin zero*, *False*.

**Proof.** *intros H; inversion H. Qed.*

**Fixpoint** *elemAt* (*A* : *Type*) (*xs* : *List A*) (*n* : *Fin* (*length* \_ *xs*)) : *A* :=

**match** *xs*, *n* **with**

| *nil*, \_ ⇒ **match** *emptyfin n* **with** **end**

| *cons x* \_, *finzero* \_ ⇒ *x*

| *cons* \_ *xs'*, *finsucc* \_ *n'* ⇒ *elemAt* \_ *xs'* *n'* (\* TODO: This case fails \*)

будет преобразован в:

```
data Nat = Zero | Succ Nat
```

```
data List a = Nil | Cons a (List a)
```

```
length :: List a1 → Nat
```

```
length Nil = Zero
```

```
length (Cons a xs') = Succ (length xs')
```

```
data Fin = Finzero Nat | Finsucc Nat Fin
```

```
-- TODO: Was not able to implement correctly in Coq
```

```
elemAt :: List a1 → Fin → a1
```

```
elemAt Nil n = error "absurd case"
```

```
elemAt (Cons x _) (Finzero _) = x
```

```
elemAt (Cons _ xs') (Finsucc _ n') = elemAt xs' n'
```

И теперь можно вызвать *elemAt* от пустого списка и получить ошибку на этапе выполнения, что нежелательно.

### 2.2.2 Для Agda

На Agda есть компилятор MAlonzo<sup>4</sup> (являющийся переписанным компилятором Alonzo[2]), который транслирует код на Agda в код на Haskell и затем компилирует его с помощью ghc<sup>5</sup> (де-факто стандарт Haskell), получая в результате исполняемый файл.

Из-за фокуса на генерацию исполняемых файлов, а не библиотек, генерируемый код создает имена вида буква+число, для функций не выписывается их тип и типы данных теряют типовые параметры. Пример из 2.1 преобразуется приблизительно в:

---

<sup>4</sup><http://thread.gmane.org/gmane.comp.lang.agda/62>

<sup>5</sup><http://www.haskell.org/ghc/>

```

name1 = "Main.Nat"
name2 = "Main.Nat.zero"
name3 = "Main.Nat.zero"
d1 = ()
data T1 a0 = C2 | C3 a0

name5 = "Main.List"
name7 = "Main.List.nil"
name8 = "Main.List.cons"
d5 a0 = ()
data T5 a0 a1 = C7 | C8 a0 a1

name10 = "Main.length"
d10 v0 C7 = unsafeCoerce C2
d10 v0 (C8 v1 v2) = unsafeCoerce (C3 (unsafeCoerce
    (d10 (unsafeCoerce v0) (unsafeCoerce v1))))

name12 = "Main.Fin"
name14 = "Main.Fin.finzero"
name16 = "Main.Fin.finsucc"
d12 a0 = ()
data T12 a0 a1 = C14 a0 | C16 a0 a1

name19 = "Main.elemAt"
d19 _ C7 _ = error "Impossible clause body"
d19 v0 (C8 v1 v2) (C14 v3) = unsafeCoerce v1
d19 v0 (C8 v1 v2) (C16 v3 v4) = unsafeCoerce (d19
    (unsafeCoerce v0) (unsafeCoerce v2) (unsafeCoerce v4))

```

Как видно, это делает использование сгенерированного кода практически неприменимым.



**Замечание об Agda 2.3.4** Работа начиналась на версии 2.3.2.2, в версии 2.3.4 появились 2 релевантные возможности:

- давать функциям пользовательские имена и заставлять MAlonzo генерировать для них более-менее осмысленные типы с помощью прагмы `{-# COMPILED_EXPORT AgdaName HaskellName #-}`
- флаг `--compile-no-main`, который не требует наличие функции *main*.

## 2.3 Анализ MAlonzo

TODO: Full description of MAlonzo internals.

## 2.4 Задачи

1. Придумать способ генерировать интерфейс на Haskell, не позволяющий нарушить инварианты, поддерживаемые Agda
2. Реализовать
3. Доказать корректность способа

## 3 Реализация

### 3.1 Генерирование ограниченного интерфейса

Как обсуждалось в 2.1, необходим способ ограничивать функционал генерируемого интерфейса, чтобы выставлять только те элементы, использование которых не может нарушить внутренние инварианты системы.

Еще необходимо уметь генерировать имена для получаемого интерфейса: правила именования в Agda и Haskell отличаются.

Поэтому было решено ввести прагму

```
{-# EXPORT AgdaName HaskellName #-},
```

которая пишется в том же модуле *AgdaModuleName*, где определяется *AgdaName*.

Если тип *AgdaName* не может быть сконвертирован в аналогичный тип на Haskell или *HaskellName* не соблюдает правила именования, то компиляция завершится с ошибкой.

На этапе компиляции вместе с `MAlonzo.Code.AgdaModuleName`, где хранится код генерируемый MAlonzo, будет сгенерирован `MAlonzo.Export.AgdaModule` в котором находится весь генерируемый интерфейс.

Решение создать отдельный модуль `MAlonzo.Export` вызвано желанием скрыть сгенерированный MAlonzo код от пользователя. В том числе это позволит при генерировании документации с помощью haddock<sup>6</sup> отображать только желаемый интерфейс.

### 3.2 Встраивание в MAlonzo

Вместо изменения кода, который генерирует MAlonzo было решено генерировать обертки, имеющие нужный интерфейс и вызывающие код, сгенерированный MAlonzo. Это позволяет менять меньше кода в MAlonzo, хотя может повлиять на производительность — оборачивание функций может быть не отброшено оптимизатором.

Кодогенерация вызывается на следующих участках:

1. При начале обработки модуля *AgdaModuleName* компилятором MAlonzo, контекст, содержащий сгенерированный код обнуляется.

---

<sup>6</sup><http://www.haskell.org/haddock/>

2. После обработки каждого определения *AgdaName*: функции или типа данных - проверяется наличие прагмы `{-# EXPORT AgdaName HaskellName #-}`. Если она не найдена, это определение пропускается; иначе - выполняется проверка на возможность сгенерировать интерфейс на Haskell. При неудаче выдается ошибка, иначе - в контекст добавляются сгенерированные обертки.
3. После обработки модуля, если контекст не пуст, создается модуль `MAlonzo.Ex` в который записывается код контекста.

### 3.3 Выполняемая кодогенерация

Формально задается в приложении А.

#### 3.3.1 Типы данных

Типы данных (**data**, **record**) экспортируются как **newtype**, а не **data**.

#### 3.3.2 Функции

Полиморфизм по-Черчу преобразуется в полиморфизм по-Карри.

## 4 Заключение

### 4.1 Выводы

### 4.2 Дальнейшая разработка

## 5 Список литературы

- [1] P. Letouzey. *A New Extraction for Coq*. TYPES2002, 2002.
- [2] M. Benke. *Alonzo — a compiler for Agda*. TYPES2007, 2007.

## А Формальное определение трансформаций

## В Доказательство корректности