

# Экстракция кода из Agda в Haskell

Шабалин Александр Леонидович

научный руководитель

к. ф.-м. н. Москвин Денис Николаевич

Академический университет

2014 г.

# Формальная верификация

- ▶ Необходимо уметь убеждаться, что написанная программа решает поставленную задачу.
- ▶ Тестирование не может показать, что программа верна для всех случаев (если, конечно, нельзя сделать полный перебор).
- ▶ Формальная верификация позволяет сравнить программу с формальной математической моделью и доказать их эквивалентность на всех входных данных.

# Agda

- ▶ Один из способов формально верифицировать — строить формулы достаточно мощной логики над элементами программы и проверять их на этапе компиляции.
- ▶ Agda — функциональный язык программирования, система типов которого позволяет строить формулы на языке предикативной конструктивной логики.

# Использование верифицированного кода

Написание верифицированного алгоритма недостаточно — необходимо еще использовать этот код из «реальных» приложений. Подходы:

1. Использовать Agda для написания приложений целиком.
  - + Можно верифицировать больше кода.
  - Не Тьюринг-полный язык.
2. По коду на Agda генерировать код на другом языке
  - + Удобнее писать «реальный» код.
  - Необходимо поддерживать корректность кода при трансляции.

Второй пункт называется «экстракция программ» и используется в системе Coq.

## Задача

По коду на Agda получить код на Haskell, который можно использовать из программы на Haskell, не нарушая внутренние инварианты, поддерживаемые Agda.

Целевым языком выбран Haskell по нескольким причинам:

- ▶ Уже есть транслятор: MAlonzo компилирует Agda через трансляцию в Haskell
- ▶ Языки синтаксически похожи
- ▶ Типы в Haskell — подмножество типов в Agda

## Ограничение выставляемого интерфейса

Поскольку на Agda можно потребовать от аргументов функций свойств, не представимых в Haskell, то необходимо уметь запрещать давать прямой доступ к ним. Иначе, можно будет передать неправильный (с точки зрения Agda) аргумент, который пройдет проверку типов на стороне Haskell, и получить падение программы на этапе исполнения.

## Существующие решения

**Coq 8.4pl3** Экстракция программ<sup>1</sup>. Генерируется код, из которого стираются все доказательства. Но это значит, что некоторые функции, требовавшие инварианты на этапе компиляции, теперь будут их требовать на этапе исполнения.

**Agda 2.3.2.2** Компилятор MAlonzo<sup>2</sup>. Фокусируется на генерировании исполняемых файлов через трансляцию в Haskell. Генерирует имена вида буква+число, теряет всю информацию о типах (кроме арности функций).

**Agda 2.3.4** Появилась возможность давать пользовательские имена функциям и генерировать для них разумные типы.

---

<sup>1</sup>P. Letouzey. *A New Extraction for Coq*. 2002

<sup>2</sup><http://thread.gmane.org/gmane.comp.lang.agda/62>

# Цели и задачи

## Цель работы

Разработать механизм для MAlonzo, генерирующий интерфейс на Haskell к коду на Agda, использование которого не позволит нарушить инварианты, поддерживаемые Agda.

## Задачи:

1. Разобраться с принципом кодогенерации в MAlonzo.
2. Разработать способ генерировать интерфейс из выделенных функций и типов данных и на этапе компиляции проверять, что он не нарушает инварианты.
3. Доказать корректность выполняемых трансформаций.
4. Реализовать и протестировать.



## Обзор реализации

- ▶ Выставляемый интерфейс представляет собой обертки над кодом, сгенерированным `MAlonzo`, которые имеют типы, поддерживающие те же инварианты, что требует код на `Agda`.
- ▶ Язык `Agda` расширен прагмой `{-# EXPORT AgdaName HaskellName #-}`, которой передается имя из `Agda` и желаемое имя в `Haskell`. Если сущность `AgdaName` представима в `Haskell` и `HaskellName` — разрешенное имя для этой сущности, то во время компиляции генерируется соответствующая обертка.
- ▶ Для модуля `AgdaModuleName` код интерфейса помещается в модуль `MAlonzo.Export.AgdaModuleName`, чтобы отделить код, сгенерированный `MAlonzo` (находящийся в `MAlonzo.Code.AgdaModuleName`) от безопасного интерфейса.

# Подробности реализации

Тип данных и все его конструкторы выразимы

Можно полностью задать этот тип на Haskell и использовать его конструкторы для создания экземпляра и для сопоставления с образцом (pattern matching).

Для реализации нужно

- ▶ либо подменить этим типом тип, генерируемый MAonzo,
- ▶ либо ввести биекцию между этими двумя типами, которая будет использоваться при генерации оберток функций.

## Подробности реализации

Тип данных выразим, но хотя бы один конструктор не выразим

Тип необходимо сделать абстрактным для внешнего кода. Ограничиваться генерированием только представимых конструкторов нельзя — множество термов, имеющих данный тип будет отличаться между Agda и Haskell.

Для реализации было принято решение делать `newtype` обертку над типом, генерируемым `MAlonzo`, что позволяет использовать `unsafeCoerce` для трансформации между ними.

# Подробности реализации

## Функции

Способ реализации параметрического полиморфизма отличается в Agda и в Haskell: Agda требует передавать параметр типа как аргумент функции, Haskell выводит его автоматически. MAlonzo при генерировании кода оставляет этот дополнительный аргумент, который всегда будет иметь значение *Unit*.

Если оставлять этот аргумент в генерируемом интерфейсе, то пользователю придется вручную передавать и пропускать *Unit*. Поэтому, генерируются обертки, которые делают это автоматически.

# Выводы

Таким образом:

1. Экстракция кода из Agda в Haskell, сохраняющая семантику, возможна для широкого класса типов данных и функций Agda.
2. Разработан способ генерировать безопасный интерфейс на Haskell к коду на Agda.
3. Доказана его корректность.

## Agda 2.3.4

- ▶ Прагма `{-# COMPILED_EXPORT AgdaName HaskellName #-}`.
- ▶ Подменяется имя функций (переименовывание типов данных не поддерживается), генерируемых MAlonzo.
- ▶ Функции требуют *Unit* на место параметров типов.

## Что дальше

- ▶ Необходимо что-то сделать для экспортирования типов данных целиком: подменять код `MAlonzo` или генерировать биекции.
- ▶ В `Agda` есть способ симулировать классы типов из `Haskell` с помощью `record` — можно попробовать генерировать соответствующие классы типов и их реализации.
- ▶ Существует множество расширений системы типов `Haskell`, которые позволяют в той или иной степени реализовывать зависимые типы — их использование позволит выразить больше типов `Agda` в `Haskell`.

## data in newtype type parameter

**newtype** *List a*, **data**  $T = \text{In Int} \mid \text{Ch Char}$

$\text{trTtoT1} :: T \rightarrow T1$ ,  $\text{trT1toT} :: T1 \rightarrow T$ ,  $\text{empty} :: \text{List } a$

$\text{add} :: T \rightarrow \text{List } T \rightarrow \text{List } T$

$\text{add} = \lambda x \text{ xs. unsafeCoerce } (d7 (\text{trTtoT1 } x) (\text{unsafeCoerce } \text{xs}))$

$\text{head} :: \text{List } a \rightarrow a$

$\text{head} = \lambda \text{xs. unsafeCoerce } (d8 (\text{unsafeCoerce } \text{xs}))$

$\text{test} = \text{head } (\text{add } (\text{In } 3) \text{ empty})$

Для корректной работы *head* обязан был вызвать *trT1toT* вместо *unsafeCoerce*.