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

1 Введение **4**
1.1 Haskell и Agda 4
1.2 Зависимые типы 4
1.2.1 Определение зависимых типов 4
1.2.2 Связь с логикой 5
1.3 Экстракция кода 5
1.4 Применение экстракции 5

2 Постановка задачи **7**
2.1 Цель 7
2.2 Сохранение внутренних инвариантов 7
2.2.1 Скрытие зависимых типов 7
2.2.2 Скрытие конструкторов типов данных 9
2.2.3 Обработка простых типов 10
2.3 Существующие решения 10
2.3.1 Экстракция для Coq 10
2.3.2 Экстракция для Agda 12
2.4 Задачи 14

3 Реализация **16**
3.1 Анализ MAlonzo 16
3.2 Генерирование ограниченного интерфейса 17
3.3 Встраивание в MAlonzo 18
3.4 Выполняемая кодогенерация 18
3.4.1 Типы данных 18
3.4.2 Типы функций 19
3.4.3 Обертки для функций 21
3.4.4 Проблема с генерированием типов данных 21

4 Заключение **28**
4.1 Выводы 28
4.2 Дальнейшая разработка 28

5 Список литературы **29**

A Формальное определение трансформаций **30**
A.1 Кайнды 30
A.2 Объявление типов 30
A.3 Встроенные типы 31
A.4 Импортированные типы 32
A.5 Преобразование типов 32
A.6 Преобразование термов 33
A.7 Объявление функций 33

B Доказательство корректности **35**
B.1 Кайнды 35
B.2 Объявление типов 36
B.3 Встроенные типы 36
B.4 Импортированные типы 36
B.5 Функции 37

1 Введение

1.1 Haskell и Agda

Haskell[1] — функциональный язык программирования общего назначения.

Agda[2] — функциональный язык программирования с зависимыми типами и, одновременно, — система компьютерного доказательства теорем.

1.2 Зависимые типы

1.2.1 Определение зависимых типов

Зависимый тип — тип, зависящий от значения. К примеру,

$$vecSum : \{n : Nat\} \to Vec\ Nat\ n \to Vec\ Nat\ n \to Vec\ Nat\ n.$$

Эта функция берет 2 вектора и делает попарную сумму. Вектор — это односвязный список, в типе которого записана его длина. Таким образом, функция принимает 3 аргумента[3]: число n, 2 вектора длины n — и возвращает вектор длины n. Тип этой функции зависимый, потому что если на первый аргумент передать 0, то ее тип будет $Nat \to Vec\ Nat\ 0 \to Vec\ Nat\ 0 \to Vec\ Nat\ 0$, а если передать 3, то — $Nat \to Vec\ Nat\ 3 \to Vec\ Nat\ 3 \to Vec\ Nat\ 3$.

Рассмотрим еще один пример:

$$vecFromListNat : (xs : List\ Nat) \to Vec\ Nat\ (length\ xs).$$

Эта функция строит из обычного односвязного списка вектор. В ее типе встроен вызов функции $length$. И если вызвать ее от пустого списка, то тип будет $List\ Nat \to Vec\ Nat\ 0$, а если от списка из 5 элементов, то — $List\ Nat \to Vec\ Nat\ 5$.

Возможность смешивать типы и термы приводит к тому, что эти 2 множества неразличимы (в отличие, к примеру, от System F). Таким образом, необходима возможность давать «типы» для типов. Эту роль исполняет Set_0. К примеру, верно, что $List\ Nat : Set_0$. В свою очередь, $Set_0 : Set_1$ и так далее.

[1] `http://haskell.org`

[2] `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.HomePage`

[3] Первый аргумент в фигурных скобках — синтаксический сахар, позволяющий программисту не писать этот аргумент при вызове функции, так как система может его вывести самостоятельно.

1.2.2 Связь с логикой

Соответствие Карри-Говарда — связь между программами и интуиционистской логикой[4]. В частности, утверждается, что типы соответствуют логическим утверждениям, а термы — доказательствам, в том смысле, что если существует терм с каким-то типом, то логическое утверждение, соответствующее этому типу будет тавтологией.

Зависимым типам соответствует логика первого порядка:

$A \to B$ соотвествует импликации,

$(x : T_1) \to T_2$ соответствует квантору всеобщности,

$T_1 : T_2 \to Set$ соответствует предикату (например, вместо T_1 можно подставить $Vec\ Nat$, а вместо T_2 — Nat)

$\bot$ соответствует `FALSE` ($\bot$ — тип данных без единого конструктора).

1.3 Экстракция кода

Термин «экстракция программ» пришел из языка/системы доказательства теорем Coq[5], похожего на Agda, и означает генерацию функционального кода из доказательств [1].

1.4 Применение экстракции

Можно выделить 2 основных причины для реализации механизма экстракции:

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

Генерирование 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

[4] Еще известна как конструктивная логика

[5] `http://coq.inria.fr`

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. В этой работе фокус ставится на этот пункт.

2 Постановка задачи

2.1 Цель

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

2.2 Сохранение внутренних инвариантов

2.2.1 Скрытие зависимых типов

Рассмотрим пример:

```
module Main where

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 3 аргумента: тип элементов в списке, список и число, меньшее длины списка. Таким образом, есть гарантия, что эта функция всегда будет вызвана с индексом внутри списка. 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 извне.

2.2.2 Скрытие конструкторов типов данных

Рассмотрим пример:

$$
\begin{aligned}
&\textbf{data } \bot \textbf{ where}\\
&\\
&\textbf{data } \_ \equiv \_ \ \{A : Set\}\ (x : A) : A \to Set \textbf{ where}\\
&\quad refl : x \equiv x\\
&\\
&\_ \not\equiv \_ : \{A : Set\} \to A \to A \to Set\\
&x \not\equiv y = x \equiv y \to \bot\\
&\\
&\textbf{data } IsEqual\ (A : Set) : Set \textbf{ where}\\
&\quad yes : (x\ y : A) \to x \equiv y \to IsEqual\ A\\
&\quad no : (x\ y : A) \to x \not\equiv y \to IsEqual\ A\\
&\\
&isEqualNat : Nat \to Nat \to IsEqual\ Nat\\
&isEqualNat = \cdots\\
&\\
&someFunction : IsEqual\ Nat \to SomeType\\
&someFunction = \cdots
\end{aligned}
$$

В примере задаются 4 типа: пустой тип, равенство, неравенство (как отрицание равенства) и тип проверки на равенство. У последнего в конструкторах содержатся соответствующие доказательства. Из-за этого невозможно полностью этот тип представить в Haskell. Но если сделать этот тип абстрактным (скрыть конструкторы), то его станет можно использовать из Haskell: построение с помощью $isEqualNat$ и использование с помощью $someFunction$.

2.2.3 Обработка простых типов

С другой стороны, такие простые типы как Nat, $List$ и функция $length$ могут быть напрямую представлены в Haskell:

$$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

$$\begin{aligned} &length :: List\ a \to Nat \\ &length\ Nil = Zero \\ &length\ (Cons\ \_\ xs) = Succ\ (length\ xs) \end{aligned}$$

Несмотря на это, типы Nat и $List$ все равно будут выставлены как абстрактные типы. Причина раскрывается в 3.4.4.

2.3 Существующие решения

2.3.1 Экстракция 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 2.2.1:

Inductive $Nat : Set :=$ | $zero : Nat$ | $succ : Nat \to Nat$.
Conjecture $succ\_zero$: **forall** $n,\ succ\ n = zero \to False$.
Definition $succ\_succ\ \{n1\}\ \{n2\}\ (e : succ\ n1 = succ\ n2) : n1 = n2 :=$
  match e **with** | $eq\_refl \Rightarrow eq\_refl$ **end**.

Inductive $List\ (A : Type) : Type :=$
| $nil : List\ A$
| $cons : A \to List\ A \to List\ A$.
Fixpoint $length\ \{A\}\ (xs : List\ A)\ \{struct\ xs\} : Nat :=$
  match xs **with**
  | $nil \Rightarrow zero$
  | $cons\ \_\ xs' \Rightarrow succ\ (length\ xs')$
  end.

Inductive $Fin : Nat \to Set :=$
| $finzero$: **forall** $n : Nat,\ Fin\ (succ\ n)$
| $finsucc$: **forall** $n : Nat,\ Fin\ n \to Fin\ (succ\ n)$.
Definition $finofzero\ \{n\}\ (f : Fin\ n) : n = zero \to False :=$
  match f **with**
  | $finzero\ \_ \Rightarrow$ **fun** $e \Rightarrow succ\_zero\ \_\ e$
  | $finsucc\ \_\ \_ \Rightarrow$ **fun** $e \Rightarrow succ\_zero\ \_\ e$
  end.

Fixpoint $elemAt'\ \{A\}\ \{n\}\ (xs : List\ A)\ (i : Fin\ n) : n = length\ xs \to A :=$
  match $xs,\ i$ **with**
  | $nil,\ \_ \Rightarrow$ **fun** $(e : n = length\ (nil\ \_)) \Rightarrow$ **match** $finofzero\ i\ e$ **with end**
  | $cons\ x\ \_,\ finzero\ \_ \Rightarrow$ **fun** $\_ \Rightarrow x$
  | $cons\ \_\ xs',\ finsucc\ \_\ i' \Rightarrow$ **fun** $e \Rightarrow elemAt'\ xs'\ i'\ (succ\_succ\ e)$
  end.

Fixpoint $elemAt\ \{A\}\ (xs : List\ A)\ (i : Fin\ (length\ xs)) : A :=$
  $elemAt'\ xs\ i\ eq\_refl$.

будет преобразован примерно в:

$$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

$$length :: List\ a1 \to Nat$$
$$length\ Nil = Zero$$
$$length\ (Cons\ a\ xs') = Succ\ (length\ xs')$$

$$\textbf{data}\ Fin = Finzero\ Nat \mid Finsucc\ Nat\ Fin$$

$$elemAt' :: Nat \to List\ a1 \to Fin \to a1$$
$$elemAt'\ \_\ Nil\ \_ = \texttt{error}\ "absurd\ case"$$
$$elemAt'\ \_\ (Cons\ x\ \_)\ (Finzero\ \_) = x$$
$$elemAt'\ \_\ (Cons\ \_\ xs')\ (Finsucc\ n0\ i') = elemAt'\ n0\ xs'\ i'$$

$$elemAt :: (List\ a1) \to Fin \to a1$$
$$elemAt\ xs\ i = elemAt'\ (length\ xs)\ xs\ i$$

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

2.3.2 Экстракция для Agda

На Agda есть компилятор MAlonzo[6] (являющийся переписанным компилятором Alonzo[2]), который транслирует код на Agda в код на Haskell и затем компилирует его с помощью ghc[7] (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 2.2.1 преобразуется приблизительно в:

[6] `http://thread.gmane.org/gmane.comp.lang.agda/62`

[7] `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))
```

Параметры для типов данных здесь используются только, чтобы объявить их для использования в конструкторах: $T12$, к примеру, мог бы аналогично (с точки зрения сохранения информации о типах) объявляться как:

$$\begin{aligned}&\textbf{data } T12 \textbf{ where}\\ &\quad C14 :: a0 \to T12\\ &\quad C16 :: a0 \to a1 \to T12\end{aligned}$$

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 осмысленные типы с помощью прагмы
  `{-# COMPILED_EXPORT` $AgdaName\ HaskellName$ `#-}`
- флаг `--compile-no-main`, который позволяет компилировать код в библиотеку, а не исполняемый файл.

Генерируемый тип для функций выполняется по следующим правилам[4]:

$$\begin{aligned}T[\![Set]\!] &= ()\\ T[\![x\ As]\!] &= x\ T[\![As]\!]\\ T[\![\lambda(x : A) \to B]\!] &= undef\\ T[\![(x : A) \to B]\!] &= \begin{cases}\textbf{forall}\ x.\ T[\![A]\!] \to T[\![B]\!] & x \in freevars(B)\\ T[\![A]\!] \to T[\![B]\!] & \text{иначе}\end{cases}\\ T[\![k\ As]\!] &= \begin{cases}T\ T[\![As]\!] & \text{задан } \texttt{\{-\# COMPILED\_TYPE } k\ T \texttt{ \#-\}}\\ () & \text{задан } \texttt{\{-\# COMPILED } k\ E \texttt{ \#-\}}\\ undef & \text{иначе}\end{cases}\end{aligned}$$

2.4 Задачи

1. Провести анализ методов трансляции и принципов генерации кода в компиляторе MAlonzo.
2. Разработать механизм генерации интерфейса на Haskell к сгенерированному MAlonzo коду, не позволяющий нарушить инварианты, поддерживаемые Agda.

3. Доказать корректность генерируемого интерфейса.

4. Реализовать поддержку механизма экстракции в компиляторе MAlonzo и провести тестирование этой реализации.

3 Реализация

3.1 Анализ MAlonzo

Ниже приводится принцип трансляции компилятора Alonzo[2]. В самой работе используется MAlonzo — переписанный с нуля Alonzo, но сохраняющий правила кодогенерации.

Для трансляции из Agda в Haskell необходимо рассмотреть трансляцию типов данных в типы данных, типов данных в значения и термов в термы.

Тип данных в тип данных Несмотря на то, что преобразование теряет информацию о типах, генерировать типы данных необходимо, потому что по термам с такими типами выполняется сопоставление с образцом на этапе исполнения. Соответственно, каждый конструктор типа данных должен транслироваться в конструктор с такой же арностью, чтобы не терять информацию, которая может быть использована при исполнении. Хотя типы данных из Agda отвечают GADT из Haskell, генерируются обычные типы данных, теряющие типовые параметры. Это гарантирует, что компилятор Haskell на этапе оптимизации не произведет какую-нибудь трансформацию, легальную с точки зрения Haskell, но меняющую семантику с точки зрения Agda.

Тип данных в функцию Поскольку в Agda типы и термы могут быть использованы в одном контексте, а в Haskell они раздельны, то необходимо для типа сгенерировать функцию, сохраняющую арность типа. То есть, число аргументов генерируемой функции должно быть равно числу параметров типа. А поскольку на типах нельзя проводить сопоставление с образцом, эта функция может просто возвращать ().

Термы в термы Термы на Agda и на Haskell используют одинаковый синтаксис и одинаковую семантику для всех случаев, кроме одного: сопоставление с образцом. Поскольку Agda — язык с зависимыми типами, то сопоставление первого аргумента, определяет тип второго аргумента. Таким образом, можно записать определение функции, в которой ведется сопоставление по двум аргументам одновременно, но конструкторы второго аргумента относятся к разным типам. В Haskell такое невозможно, поэтому необходимо разбить определение из нескольких клозов на несколько определений из одного клоза, вызывающих друг

друга по цепочке, и использовать `unsafeCoerce` для второго аргумента. Еще одно отличие, вызванное сопоставлением с образцом — наличие невозможного образца в Agda. Поскольку тип второго аргумента может меняться, в каком-то клозе он может стать типом, у которого нет конструктоов и в Agda есть специальный синтаксис, чтобы пометить этот аргумент в таком клозе и не писать реализацию этого клоза. В Haskell такого не существует и поэтому код этого клоза заменяется на вызов функции `error`.

3.2 Генерирование ограниченного интерфейса

Как обсуждалось в 2.2.1, необходим способ ограничивать функционал генерируемого интерфейса, чтобы выставлять только те элементы, использование которых не может нарушить внутренние инварианты системы.

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

Поэтому, было решено ввести прагму

$$\texttt{\{-\# EXPORT } \mathit{AgdaName}\ \mathit{HaskellName}\ \texttt{\#-\}},$$

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

Если сущность $AgdaName$ не может быть сконвертирована в аналогичную 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.`$AgdaModuleName$, 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[8] отображать только желаемый интерфейс.

[8] `http://www.haskell.org/haddock/`

3.3 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.

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 добавляется сгенерированные обертки.

3. После обработки модуля, если контекст не пуст, создается модуль `MAlonzo.Export.`$AgdaModuleName$, в который записывается код из контекста.

3.4 Выполняемая кодогенерация

Формально задается в приложении А. Корректность доказывается в приложении В.

3.4.1 Типы данных

В 2.2.2 говорилось, что некоторые типы данных (вводимые в Agda с помощью **data** и **record**) можно представить в Haskell, только если сделать их абстрактными.

Конкретнее, если объявление типа имеет вид:

$$\textbf{data}\ AgdaName\ (A_0 : S_0) \cdots (A_m : S_m) : S_{m+1} \to \cdots \to S_n \to Set_0\ \textbf{where}\ \ldots,$$

где S_i - комбинация Set_0 и $\to$, то аналогичным объявлением на Haskell будет:

$$\textbf{newtype}\ HaskellName\ (a_0 :: K_0) \cdots (a_n :: K_n) = \ldots,$$

где K_i — S_i, в котором Set_0 заменяется на $*$. Введенное ограничение на S_i гарантирует, что такой тип представим в Haskell: структурная индукция по K

$*$**:**

$$\textbf{data}\ T :: *$$

$K_0 \to \cdots \to K_n \to *$**:**

$$\textbf{data}\ T\ (a_0 :: K_0) \ldots (a_n :: K_n) :: *.$$

K_i представимы по индукции.

Set_0 — консервативный выбор. Существуют типы из Set_1, представимые в Haskell с помощью экзистенциальных типов. Но было принято решение ограничиться Set_0, потому что в Agda верно $Set_0 : Set_1$, а в Haskell $* :: *$ — неверно.

Теперь, MAlonzo по $AgdaName$ сгенерирует

$$\textbf{data}\ T_k\ a_0 \cdots a_p = \ldots$$

и если сгенерировать

$$\textbf{newtype}\ HaskellName\ (a_0 :: K_0) \cdots (a_n :: K_n) =$$
$$HaskellName\ (\textbf{forall}\ b_0 \cdots b_p.\ T_k\ b_0 \cdots b_p),$$

то для трансформации между термом из MAlonzo, имеющим тип $T_k\ args \ldots$ и термом, выставленным наружу, имеющим тип $HaskellName\ args \ldots$ можно использовать `unsafeCoerce`, так как **newtype** гарантирует[5] идентичное внутреннее представление.

3.4.2 Типы функций

Типы функций T, представимые в Haskell, имеют следующий вид:

$(A : S) \to T$, $\quad S$ состоит из Set_0 и $\to$

$X\ args \ldots$, $\quad args$ представимы и X — типовой параметр, тип из 3.4.1, FFI-импортированный тип или встроенный тип

$(x : T_1) \to T_2$, $\quad x \notin freevars(T_2)$

Тип зависимой функции с неиспользуемыми аргументами На Haskell будут иметь вид

$$T_1 \to T_2$$

то есть, абсолютно эквивалентен оригинальному.

Объявление типового параметра Для $(A : S) \to T$ рассмотрим 2 способа:

1. $\forall a.\ () \to T$
2. $\forall (a :: K).\ T, \quad K - S$, где Set_0 заменяется на $*$

Первый используется в Agda 2.3.4, как описано в 2.3.2. Второй — в данной работе.

Добавление аргумента () делает терм более похожим на Agda, так как в ней требуется передача типового аргумента терму и этот способ используется в MAlonzo при генерации кода. С другой стороны, второй способ является более естественным при использовании из Haskell, но требует написания оберток над кодом, сгенерированным MAlonzo, которые будут выполнять преобразование между 2-мя типами.

Типовой атом Разбивается на 4 случая:

1. **Типовой параметр**

   Заменяется на переменную из соответствующего объявления

2. **«Экспортированный» тип**

   Заменяется на тип, в который экспортировали

3. **«Импортированный» тип**

   Заменяется на тип, из которого импортировали с помощью `COMPILED_DATA` или `COMPILED_TYPE`

4. **Встроенный тип**

   Встроенные типы, получаемые из постулатов: `INTEGER`, `FLOAT`, `CHAR`, `STRING`, `IO` - заменяются соответственно на Int, $Float$, $Char$, $String$, IO, так как MAlonzo использует ровно эти типы вместо постулатов.

Встроенные типы вроде `LIST` в MAlonzo просто получают функции по преобразованию между [] и типом, сгенерированным по обычным правилам и таким образом эквивалентны генерированию экспорту типов вместе с конструкторами.

3.4.3 Обертки для функций

Необходимо по типу Agda и по терму из MAlonzo сгенерировать терм Haskell с типом из 3.4.2, использующий терм из MAlonzo.

Рассмотрим преобразование $Wrap^n[\![AgdaType]\!](Term) = Term$. n задает уровень вложенности. Тогда генерируемая обертка для функции с типом из Agda $AgdaType$ и термом из MAlonzo $MAlonzoTerm$ будет выражаться $Wrap^0[\![AgdaType]\!](MAlonzoTerm)$.

$Wrap^n[\![(x : T_1) \to T_2]\!](t) = \lambda x.\ Wrap^n[\![T_2]\!](t\ (Wrap^{n+1}[\![T_1]\!](x)))$**:** x имеет уровень вложенности на 1 больше по определению.

$Wrap^{2k}[\![(A : S) \to T]\!](t) = Wrap^{2k}[\![T]\!](t\ ())$**:** Если уровень вложенности четный, то t — терм из MAlonzo и необходимо вместо параметра типа подставить ()

$Wrap^{2k+1}[\![(A : S) \to T]\!](t) = Wrap^{2k+1}[\![T]\!](\lambda\_.\ t)$**:** Если уровень вложенности нечетный, то t — терм из внешнего кода и необходимо пропустить параметр типа, который MAlonzo попытается туда подставить.

$Wrap^n[\![X\ args\ldots]\!](t) =$ `unsafeCoerce` t**:** На текущий момент X либо типовой параметр, либо экспортированный тип(то есть представляется, как **newtype**-обертка над типом из MAlonzo), либо импортированный тип или встроенный тип(то есть MAlonzo использует его вместо генерации нового). Все случаи позволяют использовать `unsafeCoerce`.

Из последнего пункта вытекает проблема с использованием встроенных типов `LIST` и `BOOL`: MAlonzo генерирует свои типы для них, а также биекции между ними и [], $Bool$ соответственно. Таким образом, использовать $unsafeCoerce$ нельзя: конкретный пример в 3.4.4.

3.4.4 Проблема с генерированием типов данных

Рассмотрим код на Agda:

```
open import Data.Maybe

data List {l} (A : Set l) : Set l where
  nil : ListA
  cons : A → ListA → ListA

empty : ∀{l} {A : Set l} → List A
empty = nil

head : ∀{l} {A : Set l} → List A → Maybe A
head nil = nothing
head (cons x _) = just x

data Useless (A : Set) : Set₁ where
  useless : {B : Set} → (B → A) → B → Useless A

data Either(A B : Set) : Set where
  left : A → Either A B
  right : B → Either A B

add1 : ∀{A} → Useless A → List (Useless A) → List (Useless A)
add1 = cons

add2 : ∀{AB} → Either A B → List (Either A B) → List (Either A B)
add2 = cons
```

По нему получится примерно следующий код на Haskell, сгенерируемый MAlonzo:

$$\textbf{data}\ T1\ a0\ a1 = C2 \mid C3\ a0\ a1$$

$$d4 = \texttt{unsafeCoerce}\ (\lambda v0\ v1 \to C2)$$

$$d5\ v0\ v1\ C2 = \texttt{unsafeCoerce}\ Nothing$$
$$d5\ v0\ v1\ (C3\ v2\ v3) = \texttt{unsafeCoerce}\ (Just\ (\texttt{unsafeCoerce}\ v2))$$

$$\textbf{data}\ T6\ a0\ a1\ a2 = C7\ a0\ a1\ a2$$

$$\textbf{data}\ T8\ a0 = C9\ a0 \mid C10\ a0$$

$$d11 = \texttt{unsafeCoerce}(\lambda v0 \to C3)$$

$$d12 = \texttt{unsafeCoerce}(\lambda v0\ v1 \to C3)$$

Если потребовать экспортировать все написанные имена[9] и потребовать, чтобы $Useless$ и $Either$ были экспортированы полностью, то можно получить примерно следующий код:

[9] Нужно представить, что есть поддержка $Set\ l$(нужна только для $Useless$)

```
newtype List a = List (forall b0 b1. T1 b0 b1)

empty :: List a
empty = unsafeCoerce (d4 () ())

head' :: List a -> Maybe a
head' = \xs -> unsafeCoerce (d13 () () (unsafeCoerce xs))

data Useless a where
  Useless :: (b -> a) -> b -> Useless a
tUselessToMA :: Useless a -> T6 a0 a1 a2
tUselessToMA (Useless a0 a1) = unsafeCoerce (C7 () (\x ->
  unsafeCoerce (unsafeCoerce a0 (unsafeCoerce x))) (unsafeCoerce a1))
tUselessFromMA :: T6 a0 a1 a2 -> Useless a
tUselessFromMA (C7 a0 a1 a2) = unsafeCoerce (Useless (\x ->
  unsafeCoerce (unsafeCoerce a1 (unsafeCoerce x))) (unsafeCoerce a2))

data Either' a b where
  Right' :: b -> Either' a b
  Left' :: a -> Either' a b
tEitherToMA :: Either a b -> T8 a0
tEitherToMA (Left' a0) = unsafeCoerce (C9 (unsafeCoerce a0))
tEitherToMA (Right' a0) = unsafeCoerce (C10 (unsafeCoerce a0))
tEitherFromMA :: T8 a0 -> Either a b
tEitherFromMA (C9 a0) = unsafeCoerce (Left' (unsafeCoerce a0))
tEitherFromMA (C10 a0) = unsafeCoerce (Right' (unsafeCoerce a0))

add1 :: Useless a -> List (Useless a) -> List (Useless a)
add1 = \x xs -> unsafeCoerce(d11 () (tUselessToMA x) (unsafeCoerce xs))

add2 :: Either' a b -> List (Either' a b) -> List (Either' a b)
add2 = \x xs -> unsafeCoerce(d12 ()() (tEitherToMA x) (unsafeCoerce xs))
```

А теперь 2 функции из внешнего кода:

$$test1 :: Maybe\ (Useless\ Int)$$
$$test1 = head'\ (add1\ (Useless\ read\ \text{``3''})\ empty)$$

$$test2 :: Maybe\ (Either'\ Int\ Char)$$
$$test2 = head'\ (add2\ (Left'\ 3)\ empty)$$

В обоих случаях $add1$ и $add2$ используют соответственно $tUselessToMA$ и $tEitherToMA$ для конвертирования в тип MAlonzo, но $head'$ использует `unsafeCoerce` вместо $tUselessFromMA$ и $tEitherFromMA$ для конвертирования обратно. Можно заметить, что $Either'$ задан с конструкторами в другом порядке и поэтому тип $Either'$ гарантировано отличается от $T8$ и результат $test2$: $Just\ (Right'\ \text{`}\backslash ETX'\text{`})$. В случае с $Useless$ проблема: он лежит в Set_1, а трансформация поддерживает только Set_0. Поэтому код интерфейса был написан вручную, как если бы была возможность его сгенерировать. Здесь `unsafeCoerce` не применим, так как в конструкторе есть параметр типа в качестве аргумента, которого не будет в интерфейсе.

Таким образом, необходимо поддерживать следующий инвариант: тип данных в интерфейсе должен иметь внутреннее представление идентичное типу, генерируемому MAlonzo. Значит, необходимо либо делать `newtype`-обертки, либо менять тип, генерируемый MAlonzo.

Неприменимость проблемы для функций Выше утверждалось, что «тип данных в интерфейсе должен иметь внутренее представление, идентичное типу, генерируемому MAlonzo». Тип функций тоже в некотором роде тип данных (например, тип пары: $(a \to b \to c) \to c$) и если функция полиморфна, то этот тип будет иметь разные представления в интерфейсе и внутри MAlonzo. Рассматривая пример выше, напишем еще одну функцию:

Agda:

$add3 : (\forall\{A\} \to A \to A) \to List\ (\forall\{A\} \to A \to A) \to List\ (\forall\{A\} \to A \to A)$

$add3 = cons$

MAlonzo:

$d13 = \texttt{unsafeCoerce}\ C3$

Интерфейс:

$add3 :: (\textbf{forall}\ a.\ a \to a) \to List\ (\textbf{forall}\ a.\ a \to a) \to List\ (\textbf{forall}\ a.\ a \to a)$

$add3 = \lambda f\ fs \to \texttt{unsafeCoerce}\ (d13$

$(\lambda\_\ x \to \texttt{unsafeCoerce}\ (f\ (\texttt{unsafeCoerce}\ x)))\ (\texttt{unsafeCoerce}\ fs))$

Тест:

$test3 :: Int$

$test3 =$

$\textbf{case}\ head'\ (add3\ id\ empty)\ \textbf{of}$

$Nothing \to \texttt{error}$ “Impossible”

$Just\ f \to f\ 3$

Ошибка должна быть та же самая: один раз функцию упаковали в список, разложив ее на аргументы, достали из него с помощью `unsafeCoerce`. Но такой код нескомпилируется: он требует расширения языка `ImpredicativeTypes`, без которого у типа данных типовые параметры обязаны быть мономорфными[3]. А поскольку только полиморфные функции отличаются по структуре между интерфейсом и MAlonzo, то проблема решается автоматически.

Что если теперь вместо специально построенного *List* использовать чистое λ-исчисление для эмуляции типов данных. Сразу рассмотрим типы функций в интерфейсе:

$$lnil :: a \to b$$
$$lcons :: a \to ((a \to b \to c) \to d \to b) \to (a \to b \to c) \to d \to c$$
$$lhead :: ((c \to b \to c) \to (d \to e \to e) \to a) \to a$$
$$ladd :: (\textbf{forall}\ a.\ a \to a) \to (((\textbf{forall}\ a.\ a \to a) \to b \to c) \to d \to b) \to$$
$$((\textbf{forall}\ a.\ a \to a) \to b \to c) \to d \to c$$

$$test = lhead\ (ladd\ id\ lnil)$$

Получится та же самая ошибка компиляции: требуется импредикативность. На самом деле, ошибка с импредикативностью при использовании $List$ появляется из-за аналогичных причин.

4 Заключение

4.1 Выводы

В данной работе был рассмотрен механизм транслирования кода на Agda в код на Haskell компилятором MAlonzo, разработан способ генерирования безопасного интерфейса к нему и доказана его корректность. В результате программист на Haskell получает возможность использовать типы данных и функции из Agda, обладающие формально доказанными свойствами.

4.2 Дальнейшая разработка

Потенциальные направления развития:

- Возможность полностью экспортировать простые типы.
- Экстракция классов типов и их инстансов из **record**-эмуляций.
- Использовать современные расширения системы типов Haskell для эмуляции широкого класса зависимых типов.

5 Список литературы

[1] P. Letouzey. *A New Extraction for Coq.* TYPES2002, 2002.

[2] M. Benke. *Alonzo — a compiler for Agda.* TYPES2007, 2007.

[3] S. P. Jones, D. Vytiniotis, S. Weirich, M. Shields. *Practical type inference for arbitrary-rank types.* 2011.

[4] *Agda Foreign Function Interface.*
`http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.ForeignFunctionInterface`

[5] *The Haskell 2010 Language — Datatype Renamings.*
`https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-740004.2.3`

A Формальное определение трансформаций

Все преобразования заданы в виде

$$TransformName[\![Pattern_1]\!](Args\dots) = \begin{cases} Result_1 & Condition_{1,1} \\ \vdots & \\ Result_{k_1} & Condition_{1,k_1} \end{cases}$$

$$\vdots$$

$$TransformName[\![Pattern_n]\!](Args\dots) = \begin{cases} Result_m & Condition_{n,1} \\ \vdots & \\ Result_{m+k_n-1} & Condition_{n,k_n} \end{cases}$$

По $Pattern_i$ производится сопоставление с шаблоном в порядке сверху вниз. При совпадении, производится проверка соответствующих $Condition_{i,j}$ сверху вниз, пустой $Condition_{i,j}$ считается успешным. При успешном выполнении $Condition_{i,j}$ проверяется определен ли соответствующий $Result_l$. Если определен, то он возвращается как результат иначе идет переход к $Condition_{i,j+1}$. Если все $Condition_{i,j}$ для данного $Pattern_i$ проверены, то идет переход к $Pattern_{i+1}$. Если все $Pattern_i$ проверены, то результат становится неопределенным.

$Result_l$ определен, если все вызываемые внутри него трансформации определены.

A.1 Кайнды

$$KT[\![AgdaType]\!] = HaskellKind$$

$$KT[\![Set_0]\!] = *$$
$$KT[\![Kind_1 \to Kind_2]\!] = KT[\![Kind_1]\!] \to KT[\![Kind_2]\!]$$

A.2 Объявление типов

$$DTMA[\![AgdaTypeName]\!] = HaskellTypeName$$
$$DT[\![AgdaTypeName]\!] = HaskellTypeName$$
$$DTD[\![AgdaTypeName]\!] \doteq HaskellTypeDeclaration,$$

где

$DTMA$ — имя типа, генерируемого MAlonzo,

DT — имя запрошенное пользователем с помощью
`{-# EXPORT` $AgdaTypeName$ $HaskellTypeName$ `#-}`,

DTD — генерируемое объявление типа с именем DT.

Когда прагма `EXPORT` для имени $AgdaTypeName$ не определена, то DT и DTD становятся неопределенными.

Используя определение:

$$\textbf{data}\ AgdaDataType\ (A_1 : S_1) \cdots (A_m : S_m) : S_{m+1} \to \cdots \to S_n \to Set_0\ \textbf{where} \ldots$$

$$\begin{aligned} &DTD[\![AgdaDataType]\!] \doteq \\ &\quad \textbf{newtype}\ DT[\![AgdaDataType]\!]\ (a_1 :: KT[\![S_1]\!]) \cdots (a_n :: KT[\![S_n]\!]) \\ &\qquad = DT[\![AgdaDataType]\!]\ (\forall b_1 \cdots b_k.\ DTMA[\![AgdaDataType]\!]\ b_1 \cdots b_k), \end{aligned}$$

где k — арность конструктора типов $DTMA[\![AgdaDataType]\!]$.

Если $AgdaDataType$ — **record**, то $m = n$ и DTD определяется точно так же.

Если $AgdaDataType$ находится в параметризованном модуле или в **record**, то параметры автоматически приписываются как дополнительные $A_i : S_i$.

A.3 Встроенные типы

$$\begin{aligned} &BTMA[\![BuiltinType]\!](AgdaType) = HaskellType \\ &Builtins = \{\texttt{INTEGER}, \texttt{FLOAT}, \texttt{CHAR}, \texttt{STRING}, \texttt{IO}\}, \end{aligned}$$

где

$Builtins$ — подмножество встроенных типов, поддерживаемых MAlonzo, которые привязываются к постулатам,

$BTMA$ — определенная в MAlonzo функция преобразования встроенного типа к типу на Haskell.

Когда `{-# BUILTIN` $BuiltinType$ $AgdaType$ `#-}` не определена, то $BTMA$ становится неопределенным.

$$
\begin{aligned}
BTMA[\![\texttt{INTEGER}]\!](t) &= Int \\
BTMA[\![\texttt{FLOAT}]\!](t) &= Float \\
BTMA[\![\texttt{CHAR}]\!](t) &= Char \\
BTMA[\![\texttt{STRING}]\!](t) &= String \\
BTMA[\![\texttt{IO}]\!](t) &= IO
\end{aligned}
$$

A.4 Импортированные типы

$$CTMA[\![AgdaTypeName]\!] = HaskellTypeName$$

Когда `{-# COMPILED_TYPE` $AgdaTypeName$ $HaskellTypeName$ `#-}` или `{-# COMPILED_DATA` $AgdaTypeName$ $HaskellTypeName$ $HaskellConstructor \ldots$ `#-}` заданы, то $CTMA$ определяется как

$$CTMA[\![AgdaTypeName]\!] = HaskellTypeName.$$

A.5 Преобразование типов

$$
\begin{aligned}
&TT[\![AgdaType]\!](Context) = HaskellType \\
&Context = \{AgdaTypeVarName \mapsto HaskellTypeVarName\}
\end{aligned}
$$

$$
TT[\![T\ args \ldots]\!](\Gamma) = \begin{cases}
a\ TT[\![args \ldots]\!](\Gamma) & (T \mapsto a) \in \Gamma \\
CTMA[\![T]\!]\ TT[\![args \ldots]\!](\Gamma) & \\
BTMA[\![B]\!](T)\ TT[\![args \ldots]\!](\Gamma) & B \in Builtins \\
DT[\![T]\!]\ TT[\![args \ldots]\!](\Gamma) &
\end{cases}
$$

$$TT[\![(A : S) \to T]\!](\Gamma) = \forall (a :: KT[\![S]\!]).\ TT[\![T]\!](\Gamma \cup \{A \mapsto a\})$$

$$TT[\![(x : T_1) \to T_2]\!](\Gamma) = TT[\![T_1]\!](\Gamma) \to TT[\![T_2]\!](\Gamma) \quad x \notin freevars(T_2)$$

A.6 Преобразование термов

$$Wrap^{2k}[\![AgdaType]\!](MAlonzoTerm) = InterfaceTerm$$
$$Wrap^{2k+1}[\![AgdaType]\!](InterfaceTerm) = MAlonzoTerm$$

$Wrap^0[\![AgdaType]\!](term)$ задан только, когда $TT[\![AgdaType]\!](\varnothing)$ определен.

$$\begin{aligned}
Wrap^n[\![A\ args\ldots]\!](term) &= \texttt{unsafeCoerce}\ term \\
Wrap^{2k}[\![(A : S) \to T]\!](term) &= Wrap^{2k}[\![T]\!](term\ ()) \\
Wrap^{2k+1}[\![(A : S) \to T]\!](term) &= Wrap^{2k+1}[\![T]\!](\lambda_.\ term) \\
Wrap^n[\![(x : T_1) \to T_2]\!](term) &= \lambda x.\ Wrap^n[\![T_2]\!](term\ (Wrap^{n+1}[\![T_1]\!](x)))
\end{aligned}$$

A.7 Объявление функций

$$\begin{aligned}
VTMA[\![AgdaName]\!] &= HaskellName \\
VT[\![AgdaName]\!] &= HaskellName \\
VTD[\![AgdaName]\!] &\doteq HaskellDeclaration,
\end{aligned}$$

где

$VTMA$ — имя функции, генерируемой MAlonzo,

VT — имя запрошенное пользователем с помощью `{-# EXPORT` $AgdaName$ $HaskellName$ `#-}`,

VTD — генерируемое объявление функции с именем VT.

Когда прагма `EXPORT` для имени $AgdaName$ не определена, то VT и VTD становятся неопределенными.

Используя определение:

$$AgdaName : AgdaType$$
$$AgdaName = \ldots$$

$$
\begin{aligned}
&VTD[\![AgdaName]\!] \doteq \\
&\quad VT[\![AgdaName]\!] :: TT[\![AgdaType]\!](\varnothing) \\
&\quad VT[\![AgdaName]\!] = Wrap^0[\![AgdaType]\!](VTMA[\![AgdaName]\!])
\end{aligned}
$$

Если $AgdaName$ — конструктор какого-то типа данных, то VTD можно использовать, чтобы выставить его как обычную функцию.

Если $AgdaName$ находится в параметризованном модуле или в **record**, то параметры автоматически приписываются в $AgdaType$.

B Доказательство корректности

B.1 Кайнды

Утверждение 1. *Если*

$$KT[\![AgdaType]\!] = HaskellKind$$

определена, то существует такой тип $HaskellType$ в Haskell, что

$$HaskellType :: HaskellKind.$$

Доказательство. Структурная индукция по $AgdaType$:

Set_0**:** Тогда $HaskellKind = *$ и

$$\textbf{data}\ HaskellType :: *.$$

$S_1 \to S_2$**:** Пусть

$$K_1 = KT[\![S_1]\!],$$
$$K_2 = KT[\![S_2]\!].$$

Если K_1 или K_2 не определены, то $KT[\![AgdaType]\!]$ не определена. Иначе, $HaskellKind = K_1 \to K_2$ и существуют

$$\textbf{data}\ HaskellType_1 :: K_1 \text{ и}$$
$$\textbf{data}\ HaskellType_2 :: K_2.$$

Мы всегда можем добавить к типу данных еще один параметр. Добавим один к $HaskellType_2$:

$$\textbf{data}\ HaskellType\ (a :: K_1) :: K_2.$$

Это тоже валидный тип на Haskell, поскольку на место a можно подставить $HaskellKind_1$, и его кайнд — $K_1 \to K_2 = HaskellKind$.

иначе**:** $KT[\![AgdaType]\!]$ не определена.

□

B.2 Объявление типов

Утверждение 2. *Если определено* $DT[\![AgdaTypeName]\!] = HaskellTypeName$, *то можно использовать* `unsafeCoerce` *для преобразования между* $HaskellTypeName$ *и внутренним представлением* $AgdaTypeName$ *в MAlonzo.*

Доказательство. Если $DT[\![AgdaTypeName]\!] = ExternalHaskellTypeName$ определено, то определено $DTD[\![AgdaTypeName]\!]$ как `newtype`-обертка над $DTMA[\![AgdaTypeName]\!] = InternalHaskellTypeName$. Тогда гарантируется[10], что термы с типами $ExternalHaskellTypeName$ и $InternalHaskellTypeName$ будут иметь одинаковое внутреннее представление. Значит, можно использовать `unsafeCoerce`. □

B.3 Встроенные типы

Утверждение 3. *Если определена* $BTMA[\![B]\!](AgdaType) = HaskellType$ *для некоторого* $B \in Builtins$, *то можно использовать* `unsafeCoerce` *для преобразования между* $HaskellType$ *и внутренним представлением* $AgdaType$ *в MAlonzo.*

Доказательство. Если определена `{-# BUILTIN B` $AgdaType$ `#-}` для $B \in Builtins$, то MAlonzo при генерации кода на месте $AgdaType$ будет использовать $BTMA[\![B]\!](AgdaType)$. Таким образом, `unsafeCoerce` использовать безопасно, так как тип один и тот же. □

B.4 Импортированные типы

Утверждение 4. *Если определена* $CTMA[\![AgdaTypeName]\!] = HaskellTypeName$, *то можно использовать* `unsafeCoerce` *для преобразования между* $HaskellTypeName$ *и внутренним представлением* $AgdaTypeName$ *в MAlonzo.*

Доказательство. Если определена `{-# COMPILED_TYPE` $AgdaTypeName$ $HaskellTypeName$ `#-}` или `{-# COMPILED_DATA` $AgdaTypeName$ $HaskellTypeName$ $HaskellConstructor\ldots$ `#-}`, то MAlonzo при генерации кода на месте $AgdaTypeName$ будет использовать $HaskellTypeName$. Таким образом, `unsafeCoerce` использовать безопасно, так как тип один и тот же. □

[10] `https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-740004.2.3`

B.5 Функции

Утверждение 5. *Если определены* $VT[\![AgdaName]\!] = HaskellName$ *и* $VTD[\![AgdaName]\!]$*, то вызов* $HaskellName$ *будет эквивалентен вызову* $AgdaName$ *с соответствующими аргументами.*

Это утверждение, в том числе, означает, что все инварианты, поддерживаемые Agda, сохраняются. Поскольку набор аргументов, нарушающий инварианты, запрещен для вызова в Agda, он так же должен быть запрещен для вызова в Haskell.

Утверждение верно, если MAlonzo корректен: то есть генерируемый им код сохраняет семантику кода Agda.

Доказательство. Пусть функция $AgdaName$ определена с типом $AgdaType$. Структурная индукция по $AgdaType$:

$T\ args\ldots,$ T — экспортированный, встроенный и импортированный тип**:** По доказанным выше утверждениям: 2, 3 и 4 — использование `unsafeCoerce` между соответствующими типами MAlonzo и интерфейса легально.

Логический смысл типа данных — это утверждение с ограниченным набором доказательств (конструкторов). В случае GADT набор разрешенных конструкторов зависит от параметров $args\ldots$. Таким образом, нужно показать, что разрешенные конструкторы для $T\ args\ldots$ для стороны Agda и интерфейса одни и те же. Терм с типом $T\ \widehat{args}\ldots$ мог быть сконструирован на стороне интерфейса вызовом конструктора либо на стороне MAlonzo вызовом функции-обертки, когда $args\ldots$ — специализация $\widehat{args}\ldots$.

вызван конструктор на стороне интерфейса: Работает для `COMPILED_DATA` типов. MAlonzo гарантирует, что тип в Agda будет таким же, что и тип импортированный из Haskell.

вызвана функция-обертка: Специализация типов выполняется в Agda и в Haskell одинаково: оба используют предикативные типы. Таким образом, если подстановка верна на Haskell, то она была бы и верна на Agda. А корректный набор конструкторов гарантируется Agda в оборачиваемой функции.

$T\ args\ldots,$ T — типовой параметр**:** Если T будет специализирована на тип данных, то это случай из предыдущего пункта и `unsafeCoerce` легален.

Если T будет специализирована на функцию, то список $args\ldots$ пуст. При специализации вместо T может быть подставлена только мономорфная функция, так как иначе требуются импредикативные типы[3]. А мономорфные функции имеют аналогичное представление между MAlonzo и интерфейсом.

Для переменных логический инвариант: если при вызове T специализируется в $SpecType$, то на всех остальных местах T специализируется в $SpecType$. Это поддерживается на Agda и на Haskell.

$(A : S) \to T$**:** При преобразовании терма с соответствующим типом из MAlonzo в интерфейс, () подставляется в терм из MAlonzo первым аргументом. При обратном преобразовании, терм $InterfaceTerm$ оборачивается в $\lambda\_ \to InterfaceTerm$. Для T утверждение выполняется по индукции.

Логический смысл — объявление утверждения, параметризуемого согласно S. Как было показано в 1, если существует тип в Agda, удовлетворяющий S, значит существует тип в Haskell, удовлетворяющий соответствующему кайнду.

$(x : T_1) \to T_2, \quad x \notin freevars(T_2)$**:** Для T_1 и T_2 утверждение выполняется по индукции. При трансляции терма:

MAlonzo $\to$ интерфейс: терм с типом T_1 придет из интерфейса, терм с типом T_2 должен быть в MAlonzo;

интерфейс $\to$ MAlonzo: терм с типом T_1 придет из MAlonzo, терм с типом T_2 должен быть в интерфейсе.

Что и происходит при увеличении уровня вложенности в $Wrap$.

Логический смысл — импликация для Agda, и для Haskell.

иначе**:** тип $\forall\Gamma.\ TT[\![AgdaType]\!](\Gamma)$ не определен, а значит $VTD[\![AgdaName]\!]$ тоже не определено.

□