

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

TODO: ???, ???

Малаховски Я. М.

Санкт-Петербург
2014

Содержание

1	Введение	3
1.1	Haskell и Agda	3
1.2	Зависимые типы	3
1.3	Экстракция кода	3
1.4	Применение экстракции	3
2	Постановка задачи	5
2.1	Цель	5
2.2	Сохранение внутренних инвариантов	5
2.2.1	Скрытие зависимых типов	5
2.2.2	Скрытие конструкторов типов данных	7
2.2.3	Обработка простых типов	8
2.3	Существующие решения	8
2.3.1	Для Coq	8
2.3.2	Для Agda	10
2.4	Задачи	12
3	Реализация	14
3.1	Анализ MAlonzo	14
3.2	Генерирование ограниченного интерфейса	14
3.3	Встраивание в MAlonzo	14
3.4	Выполняемая кодогенерация	15
3.4.1	Типы данных	15
3.4.2	Типы функций	16
3.4.3	Обертки для функций	18
3.4.4	Проблема с генерированием типов данных	18
4	Заключение	23
4.1	Выводы	23
4.2	Дальнейшая разработка	23
5	Список литературы	24
A	Формальное определение трансформаций	25
B	Доказательство корректности	26

1 Введение

1.1 Haskell и Agda

Haskell¹ — функциональный язык программирования общего назначения.

Agda² — функциональный язык программирования с зависимыми типами и, одновременно, — система компьютерного доказательства теорем.

1.2 Зависимые типы

TODO: Basic definition of dependent types and its ability to be used in formal verification. TODO: And some slight intro to Agda syntax.

1.3 Экстракция кода

Термин «экстракция программ» пришел из языка/системы доказательства теорем Coq³, 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

¹<http://haskell.org>

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

³<http://coq.inria.fr>

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.

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 кюз 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:

data \perp **where**

data $_ \equiv _ \{A : Set\} (x : A) : A \rightarrow Set$ **where**

refl : $x \equiv x$

$_ \not\equiv _ : \{A : Set\} \rightarrow A \rightarrow A \rightarrow Set$

$x \not\equiv y = x \equiv y \rightarrow \perp$

data *IsEqual* ($A : Set$) : *Set* **where**

yes : $(x\ y : A) \rightarrow x \equiv y \rightarrow IsEqual\ A$

no : $(x\ y : A) \rightarrow x \not\equiv y \rightarrow IsEqual\ A$

isEqualNat : *Nat* \rightarrow *Nat* $\rightarrow IsEqual\ Nat$

isEqualNat = ...

someFunction : *IsEqual Nat* $\rightarrow SomeType$

someFunction = ...

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:

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

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

```
length :: List a → Nat
```

```
length Nil = Zero
```

```
length (Cons _ xs) = Succ (length xs)
```

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 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 $Nat : Set := | zero : Nat | succ : Nat \rightarrow Nat.$

Conjecture $succ_zero : \text{forall } n, succ\ n = zero \rightarrow False.$

Definition $succ_succ \{n1\} \{n2\} (e : succ\ n1 = succ\ n2) : n1 = n2 :=$
 $\text{match } e \text{ with } | eq_refl \Rightarrow eq_refl \text{ end}.$

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

$| nil : List\ A$

$| cons : A \rightarrow List\ A \rightarrow List\ A.$

Fixpoint $length \{A\} (xs : List\ A) \{struct\ xs\} : Nat :=$

$\text{match } xs \text{ with}$

$| nil \Rightarrow zero$

$| cons _ xs' \Rightarrow succ\ (length\ xs')$

$\text{end}.$

Inductive $Fin : Nat \rightarrow Set :=$

$| finzero : \text{forall } n : Nat, Fin\ (succ\ n)$

$| finsucc : \text{forall } n : Nat, Fin\ n \rightarrow Fin\ (succ\ n).$

Definition $finofzero \{n\} (f : Fin\ n) : n = zero \rightarrow False :=$

$\text{match } f \text{ with}$

$| finzero _ \Rightarrow \text{fun } e \Rightarrow succ_zero _ e$

$| finsucc _ _ \Rightarrow \text{fun } e \Rightarrow succ_zero _ e$

$\text{end}.$

Fixpoint $elemAt' \{A\} \{n\} (xs : List\ A) (i : Fin\ n) : n = length\ xs \rightarrow A :=$

$\text{match } xs, i \text{ with}$

$| nil, _ \Rightarrow \text{fun } (e : n = length\ (nil _)) \Rightarrow \text{match } finofzero\ i\ e \text{ with end}$

$| cons\ x _, finzero _ \Rightarrow \text{fun } _ \Rightarrow x$

$| cons _ xs', finsucc _ i' \Rightarrow \text{fun } e \Rightarrow elemAt' \ xs' \ i' \ (succ_succ\ e)$

$\text{end}.$

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

$elemAt' \ xs\ i\ eq_refl.$

будет преобразован примерно 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
```

```
elemAt' :: Nat → List a1 → Fin → a1
```

```
elemAt' _ Nil _ = error "absurd case"
```

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

```
elemAt' _ (Cons _ xs') (Finsucc n0 i') = elemAt' n0 xs' i'
```

```
elemAt :: (List a1) → Fin → a1
```

```
elemAt xs i = elemAt' (length xs) xs i
```

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

2.3.2 Для Agda

На Agda есть компилятор MAlonzo⁴ (являющийся переписанным компилятором Alonzo[2]), который транслирует код на Agda в код на Haskell и затем компилирует его с помощью ghc⁵ (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:

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

⁵<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:

```

data T12 where
  C14 :: a0 → T12
  C16 :: a0 → a1 → T12

```

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 правилам⁶:

$$\begin{aligned}
T\llbracket Set \rrbracket &= () \\
T\llbracket x \ As \rrbracket &= x \ T\llbracket As \rrbracket \\
T\llbracket \lambda(x : A) \rightarrow B \rrbracket &= \textit{undef} \\
T\llbracket (x : A) \rightarrow B \rrbracket &= \begin{cases} \textit{forall } x. T\llbracket A \rrbracket \rightarrow T\llbracket B \rrbracket & x \in fv \ B \\ T\llbracket A \rrbracket \rightarrow T\llbracket B \rrbracket & \text{иначе} \end{cases} \\
T\llbracket k \ As \rrbracket &= \begin{cases} T \ T\llbracket As \rrbracket & \text{COMPILED_TYPE } k \ T \\ () & \text{COMPILED } k \ E \\ \textit{undef} & \text{иначе} \end{cases}
\end{aligned}$$

2.4 Задачи

1. Разобраться со способом генерации кода MAlonzo.
2. Придумать способ генерировать интерфейс на Haskell к сгенерированному MAlonzo коду, не позволяющий нарушить инварианты, поддерживаемые Agda.

⁶<http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.ForeignFunctionInterface>

3. Доказать корректность генерируемого интерфейса.
4. Реализовать поддержку этого механизма в компиляторе MAlonzo и протестировать.

3 Реализация

3.1 Анализ MAlonzo

TODO: Full description of MAlonzo internals.

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

$$\{-\# \text{ EXPORT } AgdaName \text{ HaskellName } \#-\},$$

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`⁷ отображать только желаемый интерфейс.

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

⁷<http://www.haskell.org/haddock/>

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 добавляются 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:

data *AgdaName* ($A_0 : S_0$) \cdots ($A_m : S_m$) : $S_{m+1} \rightarrow \cdots \rightarrow S_n \rightarrow Set_0$ **where** \dots ,

где S_i - комбинация Set_0 ⁸ и \rightarrow , то аналогичным объявлением на Haskell будет:

newtype *HaskellName* ($a_0 :: K_0$) \cdots ($a_n :: K_n$) = \dots ,

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

⁸ Set_0 — консервативный выбор

$*$:

data $T :: *$

$K_0 \rightarrow \dots \rightarrow K_n \rightarrow *$:

data $T (a_0 :: K_0) \dots (a_n :: K_n) :: *$.

K_i представимы по индукции.

Теперь, MAlonzo по *AgdaName* сгенерирует

data $T_k a_0 \dots a_p = \dots$

и если сгенерировать

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

то для трансформации между термом из MAlonzo, имеющим тип $T_k \text{ args } \dots$ и термом, выставленным наружу, имеющим тип $HaskellName \text{ args } \dots$ можно использовать `unsafeCoerce`, так как **newtype** гарантирует⁹ идентичное внутреннее представление.

3.4.2 Типы функций

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

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

$X \text{ args } \dots$, args представимы и X — типовой параметр, тип из 3.4.1,

FFI-импортированный тип или встроенный тип

$(x : T_1) \rightarrow T_2$, $x \notin \text{freevars}(T_2)$

Тип зависимой функции с неиспользуемыми аргументами На Haskell будут иметь вид

$T_1 \rightarrow T_2$

то есть, абсолютно эквивалентен оригинальному.

⁹ <https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-740004.2.3>

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

1. $\forall a. () \rightarrow T$
2. $\forall (a :: K). T$, $K \leftarrow S$, 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 $Wrap^n \llbracket AgdaType \rrbracket (Term) = Term$. n задает уровень вложенности. Тогда генерируемая обертка для функции с типом из Agda $AgdaType$ и термом из MAlonzo $MAlonzoTerm$ будет выражаться $Wrap^0 \llbracket AgdaType \rrbracket (MAlonzoTerm)$.

$Wrap^n \llbracket (x : T_1) \rightarrow T_2 \rrbracket (t) = \lambda x. Wrap^n \llbracket T_2 \rrbracket (t (Wrap^{n+1} \llbracket T_1 \rrbracket (x)))$: x имеет уровень вложенности на 1 больше по определению.

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

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

$Wrap^n \llbracket X \text{ args } \dots \rrbracket (t) = \text{unsafeCoerce } t$: 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 : *List A*

cons : *A* → *List A* → *List A*

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:

data $T1\ a0\ a1 = C2 \mid C3\ a0\ a1$

$d4 = \text{unsafeCoerce } (\lambda v0\ v1 \rightarrow C2)$

$d5\ v0\ v1\ C2 = \text{unsafeCoerce } \textit{Nothing}$

$d5\ v0\ v1\ (C3\ v2\ v3) = \text{unsafeCoerce } (\textit{Just } (\text{unsafeCoerce } v2))$

data $T6\ a0\ a1\ a2 = C7\ a0\ a1\ a2$

data $T8\ a0 = C9\ a0 \mid C10\ a0$

$d11 = \text{unsafeCoerce}(\lambda v0 \rightarrow C3)$

$d12 = \text{unsafeCoerce}(\lambda v0\ v1 \rightarrow C3)$

Если потребовать экспортировать все написанные имена¹⁰ и потребовать, чтобы *Useless* и *Either* были экспортированы полностью, то можно получить примерно следующий код:

¹⁰ 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' = λ*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 () (λ*x* →
unsafeCoerce (unsafeCoerce *a0* (unsafeCoerce *x*))) (unsafeCoerce *a1*))

tUselessFromMA :: *T6 a0 a1 a2* → *Useless a*

tUselessFromMA (C7 *a0 a1 a2*) = unsafeCoerce (*Useless* (λ*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 = λ*x xs* → unsafeCoerce(d11 () (tUselessToMA *x*) (unsafeCoerce *xs*))

add2 :: *Either' a b* → *List (Either' a b)* → *List (Either' a b)*

add2 = λ*x xs* → unsafeCoerce(d12 ()₂₁) (tEitherToMA *x*) (unsafeCoerce *xs*))

А теперь 2 функции из внешнего кода:

```
test1 :: Maybe (Useless Int)
test1 = head' (add1 (Useless read "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' '\ETX')*. В случае с *Useless* проблема: он лежит в *Set₁*, а трансформация поддерживает только *Set₀*. 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*.

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