

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	2
1.1	Haskell и Agda	2
1.2	TODO: Зависимые типы?	2
1.3	Экстракция кода	2
1.4	Применение экстракции	2
2	Постановка задачи	3
2.1	Цель	3
2.2	Существующие решения	4
2.2.1	Для Coq	4
2.2.2	Для Agda	4
2.3	Анализ MAlonzo	6
2.4	Задачи	6
3	Реализация	7
3.1	Архитектура	7
3.2	TODO: ???	7
3.3	TODO: PROFIT... Ha! See what I did there? No? I will go now... .	7
4	Заключение	8
4.1	Выводы	8
4.2	Дальнейшая разработка	8
5	Список литературы	9
A	Формальное определение трансформаций	10
B	Доказательство корректности	11

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 TODO: Зависимые типы?

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

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.

¹<http://haskell.org>

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

³<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 будет преобразован в: TODO: actually test with Coq

```
...  
data Fin = FinZero | FinSucc Fin  
elemAt :: List a → Fin → a  
...
```

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 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 2.1 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))

```

Как видно, это делает использование сгенерированного кода практически неприменимым.

Замечание об Agda 2.3.4 TODO: A note about Agda 2.3.4 which actually goes forward with supporting libraries

2.3 Анализ MAlonzo

TODO: Full description of MAlonzo internals.

2.4 Задачи

1. Реализовать
2. ???
3. PROFIT

TODO: Mention somewhere that the goal is to create a system TODO: that does not break invariants set up by Agda.

3 Реализация

3.1 Архитектура

Вместо изменения кодогенерации в MAlonzo было решено сгенерировать обертки, имеющие нужный интерфейс и вызывающие код MAlonzo. Это позволит менять меньше кода в MAlonzo, но это внесет проблемы с производительностью.

Решение является частью MAlonzo, код встроен на трех участках пути:

1. при начале обработки модуля вызывается обнуление контекста,
2. при обработке каждого определения верхнего уровня вызывается функция, проверяющая надо ли генерировать обертку для данного определения,
3. при окончании обработки модуля, если необходимо генерировать код, создается новый модуль, в который помещаются все обертки.

3.2 TODO: ???

3.3 TODO: PROFIT... Ha! See what I did there? No? I will go now...

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