

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Гарифуллин Шамиль Раифович

Генерация зависимых языков по спецификации пользователя

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
аспирант Исаев В. И.

Рецензент:
аспирант Подкопаев А. В.

Санкт-Петербург
2017

SAINT-PETERSBURG ACADEMIC UNIVERSITY
Higher education centre

Department of Mathematics and Information Technology

Shamil Garifullin

Specification based generation of languages with dependent types

Graduation Thesis

Admitted for defence.

Head of the chair:
professor Alexander Omelchenko

Scientific supervisor:
PhD student Valeriy Isaev

Reviewer:
PhD student Anton Podkopaev

Saint-Petersburg
2017

Оглавление

Введение	4
1. Постановка задачи	5
2. Зависимые языки	6
2.1. Проверка типов в зависимых языках	6
3. Определение языка спецификаций	7
3.1. Ограничения на спецификации, налагаемые языком	8
3.2. Проверки корректности спецификации языка	10
4. Реализация	11
4.1. Парсер генераторы	11
4.2. Индексы де Брейна и их проблемы(задачки с индексами)	11
4.3. Проверка типов	11
4.4. сама генерация кода - просто описать exts + структуру	11
4.5. Упорядочивание переменных в функциональном символе	11
Заключение	12
Список литературы	13
Приложения	14
Приложения	14
А. Доказательство корректности функции sort	14
В. Индексы де Брейна	14

Введение

Языки программирования с зависимыми типами могут быть использованы для доказательств свойств кода программы. Также возможно ввести типы аналогичные сущностям области математики в которой мы хотим доказывать теоремы и просто писать термы, таким образом предъявляя доказательства утверждений. Плюс данного подхода заключается в том, что проверка доказательств перекладывается на тайпчекер.

Однако сами языки программирования, являясь достаточно общими, часто содержат слишком много конструкций для интересующей нас области, и приходится ограничивать язык на котором мы пишем. Также может быть такая ситуация, что конструкции, которыми мы хотим пользоваться, не существуют в языке программирования. Поэтому если мы хотим переложить проверку наших высказываний на тайпчекер приходится писать свой язык программирования и уже в нем доказывать утверждения.

Решение описанной проблемы - генерация зависимых языков по спецификации конструкций, которые мы хотим от нашего языка является темой данной работы.

1. Постановка задачи

Целью данной работы является дизайн и имплементация языка для спецификации языков программирования с зависимыми типами. Ключевые задачи которые решает работа:

- Сужение множества возможных спецификаций зависимых языков, для того чтобы была возможна генерация тайпчекера
- Реализация генерации структур данных представления языка и функций манипуляции этими структурами.
- Реализация генерации функций приведения термов специфицированного языка в нормальную форму и проверки типов.

2. Зависимые языки

Языки с зависимыми типами позволяют типам зависеть от термов, то есть мы, например, можем иметь тип списков фиксированной длины. Что позволяет нам описывать ограничения налагаемые на использование функций, которые мы пишем.

Одной из наиболее частых ошибок при программировании на языке вида Haskell является взятие первого элемента списка.

```
head :: [a] -> a
head (x:_) = x
head [] = error "No head!"
```

Которая легко решается если мы можем иметь термы языка в типе.

```
head :: {n : N} -> Vec a (suc n) -> a
head (x:_) = x
```

Здесь тип явно специфицирует что функция не принимает термы типа 'Vec a 0'

Этот способ обобщается и можно доказывать корректность работы алгоритмов, например функции filter в Приложении А.

2.1. Проверка типов в зависимых языках

Рассмотрим пример:

$$\frac{\Gamma, x : S \vdash T \text{ type} \quad \Gamma, \vdash f : pi(S, T) \quad \Gamma \vdash t : S}{\Gamma \vdash app(f, t, T) : T[x := t]}$$

Если считать что заключение правила вывода, то проверка типов в любом языке происходит так: мы имеем некоторые аргументы внутри примитива, которые мы используем для составления узлов-потомков (предпосылок).

На этих узлах вызываем функцию вывода типов в возможно расширенном контексте¹ рекурсивно. Если потомки составлены корректно, то получаем некие типы которые можем использовать в проверке некоторых равенств и возврате типа примитива.

В зависимых языках все точно так же, однако проверка на равенство должна происходить после нормализации термов. Нормализацию мы применяем только после того как убедимся, что термы корректно составлены. То есть имеем факт того, что нормализация тесно связана с проверкой типов, а именно: проверка типов невозможна без нормализации термов.

Действительно, чтобы понять что $2 + 3 = 5$ мы должны провести вычисления и убедиться в этом. Действительно, чтобы понять что $2 + 3 = 5$ мы должны провести вычисления и убедиться в этом. Действительно, чтобы понять что $2 + 3 = 5$ мы должны провести вычисления и убедиться в этом.

¹Конечно мы должны для каждого расширения контекста проверять его корректность.

Syntax		Kinding	
$t ::=$		$\boxed{\Gamma \vdash T :: K}$	
x	terms:	$\frac{X :: K \in \Gamma \quad \Gamma \vdash K}{\Gamma \vdash X :: K}$	(K-VAR)
$\lambda x:T.t$	variable		
$t t$	abstraction	$\frac{\Gamma \vdash T_1 :: * \quad \Gamma, x:T_1 \vdash T_2 :: *}{\Gamma \vdash \Pi x:T_1.T_2 :: *}$	(K-PI)
$T ::=$	types:	$\frac{\Gamma \vdash S :: \Pi x:T.K \quad \Gamma \vdash t : T}{\Gamma \vdash S t : [x \mapsto t]K}$	(K-APP)
X	type/family variable	$\frac{\Gamma \vdash T :: K \quad \Gamma \vdash K \equiv K'}{\Gamma \vdash T :: K'}$	(K-CONV)
$\Pi x:T.T$	dependent product type		
$T t$	type family application		
$K ::=$	kinds:	$\boxed{\Gamma \vdash t : T}$	
$*$	kind of proper types	$\frac{x:T \in \Gamma \quad \Gamma \vdash T :: *}{\Gamma \vdash x : T}$	(T-VAR)
$\Pi x:T.K$	kind of type families	$\frac{\Gamma \vdash S :: * \quad \Gamma, x:S \vdash t : T}{\Gamma \vdash \lambda x:S.t : \Pi x:S.T}$	(T-ABS)
$\Gamma ::=$	contexts:	$\frac{\Gamma \vdash t_1 : \Pi x:S.T \quad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1 t_2 : [x \mapsto t_2]T}$	(T-APP)
\emptyset	empty context	$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T \equiv T' :: *}{\Gamma \vdash t : T'}$	(T-CONV)
$\Gamma, x:T$	term variable binding		
$\Gamma, X::K$	type variable binding		
Well-formed kinds		$\boxed{\Gamma \vdash K}$	
		$\frac{\Gamma \vdash *}{\Gamma \vdash \Pi x:T.K}$	(WF-STAR)
		$\frac{\Gamma \vdash T :: * \quad \Gamma, x:T \vdash K}{\Gamma \vdash \Pi x:T.K}$	(WF-PI)

Рис. 1: Язык с лямбдой и Π -типами

3. Определение языка спецификаций

Вдохновением данной работы послужила статьи [3] и [2]. Поэтому сам язык спецификации выглядит как язык описания алгебраических теорий².

Начнем с примера описания языка с зависимыми типами (рис.1) [4, Глава 2.1]

У нас явно выделяются три сорта(можно думать о сортах как о метатипах): кайнды, термы и типы(правила связанные с кайндами и само их описание опущены для простоты).

Также явно выделяются примитивы языка³: абстракция, пи-типы (стрелки в языках без зависимых типов) и аппликация. Легко заметить, что во всех языках присутствуют подстановка, контексты, символ ':' означающий что тип терма слева есть терм справа и связывание переменных.

Если принять во внимания все наблюдения выше то так этот язык будет выглядеть в нашем языке спецификации⁴:

DependentSorts :

²А именно: помимо правил вывода у нас есть сорта и функциональные символы.

³В дальнейшем мы называем из функциональными символами.

⁴Важно понимать что запись $_ \vdash$ не означает что контекст пуст, если слева ничего не написано это эквивалентно записи $\Gamma \vdash$.

```

tm, ty
FunctionalSymbols:
  lam: (ty, 0)*(tm, 1) -> tm
  app: (tm, 0)*(tm, 0)*(ty, 1) -> tm
  pi : (ty, 0)*(ty, 1) -> ty
Axioms:
  K-Pi =
    forall T1 : ty, x.T2 : ty
      x : T1 |- T2 def ---- |- pi(T1, x.T2) def

  TAbs =
    forall S : ty, x.T : ty, x.t : tm
      x : S |- t : T ---- |- lam(S, x.t) : pi(S, x.T)

  TApp =
    forall t1 : tm, t2 : tm, S : ty, x.T : ty
      |- t1 : pi(S, x.T),
      |- t2 : S,
      x : S |- T def
      -----
      |- app(t1, t2, x.T) : T[x:=t2]

```

Reductions:

```

Beta =
  forall x.b : tm, A : ty, a : tm, z.T : ty
    ---- |- app(lam(A, x.b), a, z.T) => b[x:=a] : T[z:=a]

```

Типизирование метапеременных позволяет проверять правильность применения функциональных символов и наличие нужных переменных в контексте. Именованные переменные служат для определения порядка переменных в контексте и не несут какой-то дополнительной информации.

Также в язык была добавлена проверка на с-стабильность - можно помечать аксиомы типами, тогда аксиома применима только если все переменные входящие в терм являются представителями этих типов⁵.

3.1. Ограничения на спецификации, налагаемые языком

1. Все используемые метапеременные должны иметь аннотацию (сорт), то есть присутствовать в секции forall аксиомы/редукции.
2. Запрещено равенство в заключении аксиом, для определенности каждого шага

⁵Если список типов пуст, то производится проверка на отсутствие свободных переменных

в проверке типов определяемого языка (если видим равенство не ясно в какую сторону идти при редуцировании)

3. Все аргументы в функциональный символ в заключении аксиомы должны быть метапеременными. Ещё и с теми же аргументами что и в forall (не больше).
4. Если в заключении аксиомы написан функциональный символ возвращающий сорт, он обязан также иметь тип (нельзя просто написать $\vdash f(\dots)def$).
5. Определения функциональных символов всегда одно, иначе появляется недетерминированность в проверке типов. Не играет особой роли, тк в данном случае можно сделать недетерминированность в проверке.
6. Подстановки разрешены только в метапеременные - в принципе это слабое ограничение, которое облегчает жизнь при реализации, не ограничивая пользователя.
7. В заключении контекст не должен быть расширен - это ограничение связано с тем, что иначе смысл аксиомы становится странным. А именно: функциональный символ применим только при введении перепенных в контекст.
8. Все метапеременные используемые в предпосылках должны либо присутствовать в метапеременных заключения или же должны быть типами какой-либо предпосылки.
9. Если в функциональном символе встречаются метапеременные с контекстами $x_1 \dots x_k.T$, должна существовать предпосылка вида $x_1 : S_1 \dots x_k : S_k \vdash T$. Это сделано для того чтобы не передавать типы контекстов метапеременных функционального символа явно.
10. Если метапеременная является типом предпосылки и не встречается в аргументах функционального символа, то она может использоваться только справа от двоеточия. Таким образом избегаются ситуации связанные с порядком проверки предпосылок языка. А именно: если у нас есть $x : S \vdash t : T, x : T \vdash r : S$. То нужно строить граф зависимостей для предпосылок и использовать порядок полученный в результате его топологической сортировки в генерации кода. (Аналогично с 4.5).
11. Все переменные контекстов метапеременных могут использовать только метапеременные левее внутри функционального символа в заключении - это связано с тем, что иначе могут возникнуть циклы в определениях метапеременных: S тип с аргументом типа R, R тип с аргументом типа S, S тип с аргументом типа R...

12. Из-за ослабления условия на метапеременные в пункте 8, порядок метапеременных неочевиден. Решение данной проблемы описано в секции 4.5.
13. Редукции не учитывают предпосылок при приведении в нормальную форму - предполагается что они не конфликтуют с аксиомами и проверки в аксиомах достаточно.
14. В редукциях все метапеременные справа от ' \Rightarrow ' должны встречаться и слева от него.
15. Подстановка запрещена слева от ' \Rightarrow '.
16. Все редукции всегда стабильны.

3.2. Проверки корректности спецификации языка

Все ограничения выше проверяются при обработке спецификации языка.

Также тривиальными проверками, осуществляемыми после парсинга языка, являются:

- Проверка того, что сорта используемых выражений совпадают с сортами аргументов функциональных символов.
- Подстановка осуществляется в переменные, которые есть в свободном виде в метапеременной.
- Контексты метапеременных содержат все их метапеременные.
- Все функциональные символы имеют правило ассоциированное вывода.

4. Реализация

4.1. Парсер генераторы

4.2. Индексы де Брейна и их проблемы(задачки с индексами)

4.3. Проверка типов

4.4. сама генерация кода - просто описать exts + структуру

4.5. Упорядочивание переменных в функциональном символе

Заключение

Список литературы

- [1] Agda programming language. — 2017. — Access mode: <http://wiki.portal.chalmers.se/agda/pmwiki.php> (online; accessed: 25.05.2017).
- [2] Isaev Valery. Algebraic Presentations of Dependent Type Theories. — arxiv : math.LO, cs.LO, math.CT/<http://arxiv.org/abs/1602.08504v3>.
- [3] Palmgren E., Vickers S.J. Partial Horn logic and cartesian categories // Annals of Pure and Applied Logic. — 2007. — Vol. 145, no. 3. — P. 314 – 353. — Access mode: <http://www.sciencedirect.com/science/article/pii/S0168007206001229>.
- [4] Pierce Benjamin C. Advanced Topics in Types and Programming Languages. — The MIT Press, 2004. — ISBN: 0262162288.

Приложения

А. Доказательство корректности функции sort

Ниже показан пример доказательства того, что функция `filter` выдает подсписок исходного списка. Код написан на Agda[1]

— Определяем предикат принадлежности элемента списку .

```
data ∈_ {A : Set} (a : A) : List A → Set where
  here : (xs : List A) → a ∈ (a ∷ xs)
  there : (x : A) (xs : List A) → a ∈ xs → a ∈ (x ∷ xs)
```

— Определяем предикат `xs ∷ ys`, означающий список ” `xs` является подсписком `ys` ”.

```
data ∷_ {A : Set} : List A → List A → Set where
  nil : [] ∷ []
  larger : {y : A} {xs ys : List A} → xs ∷ ys → xs ∷ (y ∷ ys)
  cons : {x : A} {xs ys : List A} → xs ∷ ys → (x ∷ xs) ∷ (x ∷
ys)
```

— Докажем, что `filter xs ∷ xs` для любого списка `xs`.

```
filter' : {A : Set} → (A → Bool) → List A → List A
filter' p [] = []
filter' p (x ∷ xs) = if p x then x ∷ filter' p xs else filter' p xs
```

```
filterLess : {A : Set} → (p : A → Bool) → (xs : List A) → filter' p
xs
```

```
filterLess p [] = nil
filterLess p (x ∷ xs) with p x
filterLess p (x ∷ xs) | false = larger (filterLess p xs)
filterLess p (x ∷ xs) | true = cons (filterLess p xs)
```

В. Индексы де Брейна