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

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 пользоватсья, 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 программировнии 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:

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

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 контексте[1] рекурсивно. Если потомки составлены корректно, то получаем некие типы которые можем использовать в проверке некоторых равенств и возрате 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.

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

2.2. Индексы де Брейна

При реализации функциональных языков одной из самых сложных частей является написание подстановок. Большинство проблем и ошибок в реализации тоже связано с ней.

Одной из таких проблем является сравнение альфа-эквивалентных термов. Альфа-эквивалентными называются термы которые отличаются только в именовании связанных переменных. Например следующие три терма альфа-эквивалентны:

```
\x y -> y (x z)
\y x -> x (y z)
\a b -> b (a z)
```

Понятно что мы сталкиваемся с проблемами при использовании переменных в виде строк, например первый терм сверху выглядел бы как [Lam "x" (Lam "y" (App "y" (App "x" "z")))]. И проверка равенства этого терма терму [Lam "y" (Lam "x" (App "x" (App "y" "z")))] занятие склонное к ошибкам.

Другой проблемой такого представления термов является избегание захвата переменных при подстановке. Положим мы подставляем первый терм ниже в переменную "z" во втором.

```
\x -> y
\y -> z
\y -> \x -> y = \y x -> y
```

Очевидно что так делать нельзя, тк переменная "y" стала связанной, хотя не была таковой в первоначальном терме.

Ключевым замечанием является то, что переменные в функциональных языках являются указателями на место их связывания — этаким индексом в контекст и не несут никакой дополнительной информации.

Результат применения этого наблюдения называется индексами де Брейна. А именно: для каждой связанной переменной мы просто пишем расстояние от неё до ближайшего связывания.

Если переписать термы с альфа эквивалентностью выше то получим [\\->1 (2 z)] и проверка на альфа-эквивалентность превращается в проверку на равенство.

Также решается проблема избегания захвата переменных, а именно:

```
\ -> y
\ -> z
\ -> \ -> y = \\ -> y
```

Как видно "y" остался свободным.

Это представление значительно лучше удовлетворяет нашим требованиям разработчика языков. Мы перешли от [Lam "y" (Lam "x" (App "x" (App "y" "z")))] к [Lam

(Lam (App 1 (App 2 ”z”)))].

Однако общей проблемой обоих представлений является нетипизированность переменных — никто не контролирует построение термов вида [Lam (Lam (App 123 (App 23 ”z”)))]. Решение этой проблемы описано в секции 5.2.

3. Обзор аналогов

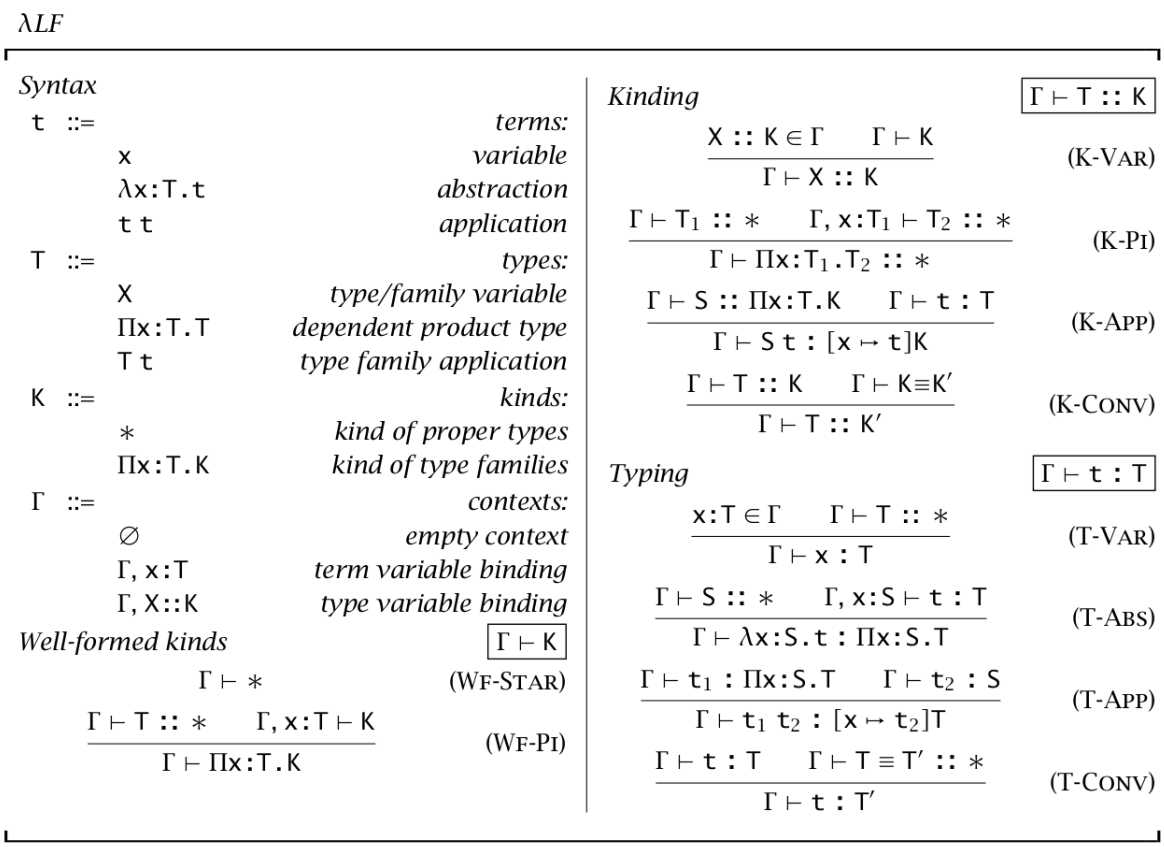

Рис. 1: Язык с лямбдой и П-типами

4. Определение языка спецификаций

Вдохновением данной работы послужила статьи [5] и [4]. Поэтому сам язык спецификации выглядит как язык описания алгебраических теорий[2].

Начнем с примера описания языка с зависимыми типами (рис.1) [6, 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 языка[3]: абстракция, пи-типы (стрелки в языках без зависимых типов) и аппликация. Легко заметить, что во всех яхыках 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 спецификации[4]:

DependentSorts :

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

[3]В дальнейшем мы называем из функциональными символами.

[4]Важно понимать что запись _ ⊢ не означает что контекст пуст, если слева ничего не написано это эквивалентно записи Γ ⊢.

```
  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 c-стабильность - можно помечать аксиомы типами, тогда аксиома применима только если все переменные входящие в терм являются представителями этих типов[5].

4.1. 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

[5]Если 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 аргуементами что и в forall (не больше).

4. Если в заключении аксиомы написан функциональный символ возвращающий сорт, он обязан также иметь тип (нельзя просто написать $\vdash f(\ldots)def$).

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 $x_1 \ldots x_k.T$, должна существовать предпосылка вида $x_1 : S_1 \ldots x_k : S_k \vdash T$. 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 B).

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 и (11) описано в Секции B.

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 '=>' должны встречаться и слева от него.

15. Подстановка запрещена слева от '=>'.

16. Все редукции всегда стабильны.

4.2. Проверки корректности спецификации языка

Все ограничения выше проверяются при обработке спецификации языка.

Также тривиальными проверками, осуществляемыми после парсинга языка, являются:

- Проверка того, что сорты 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.

5. Реализация

В данной секции описана реализация языка спецификации языков с зависимыми типами.

5.1. Парсер генераторы

В ходе всей работы использовались лексер и парсер генераторы alex[2] и happy[3].

Решение использовать именно парсер генераторы, а не парсер комбинаторы[?] или другие методы парсинга было обусловлено тем, что прогнозировались частые изменения грамматики вместе с эволюцией языка.

```
Axiom    :    Header '=' '\t' Forall '\t'
              Premise '|---' JudgementNoEq '/t' '/t'
                { Axiom (snd $1) (fst $1) $4 $6 $8 }
         |    Header '=' '\t'
              Premise '|---' JudgementNoEq '/t'
                { Axiom (snd $1) (fst $1) [] $4 $6 }
```

Listing 1: Часть спецификации парсера

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

5.3. Построение термов

Одной из проблем индексов де Брейна является их жесткая привязка к порядку переменных в контексте. Действительно чтобы переставить аргументы терма [Lam "y" (Lam "x" (App "x" (App "y" (App "y" "y"))))] мы всего-лишь меняем их местами в моменты их связывания и получаем [Lam "x" (Lam "y" (App "x" (App "y" (App "y" "y"))))]. Однако схожая операция для представления с использованием индексов де Брейна выливается в обход всего терма(!) [Lam (Lam (App 1 (App 2 (App 2 2))))] превращается в [Lam (Lam (App 2 (App 1 (App 1 1))))].

Но если уж пользователь так написал спецификацию, что мы имеем терм с другим порядком переменных или терм с большим их количеством, то мы должны поменять эти переменные местами и даже попытаться удалить лишние переменные.

Например чтобы привести "(x y z).T" к "(z x).T". Мы должны удалить "y" и переставить "x" и "z" местами.

Так же мы поступаем при возможном расширении контекста нашей метапеременной, например имеем "S" и хотим построить "Lam A x.S" — здесь нужна метапеременная "x.S", мы получаем её добавляя переменную в её контекст.

Решение предлагаемое в данной работе состоит из композиций операций swap_i'j, remove_i и add_i. Каждая операция выполняет traverse терма, который мы меняем. Примеры функций:

```
swap1'2 :: Var (Var  a) -> Identity (Var (Var  a))
swap1'2 (B ) = pure (F (B ))
swap1'2 (F (B )) = pure (B)
swap1'2 x = pure x

rem2 :: Var (Var a) -> TC (Var a)
rem2 B = pure B
rem2 (F B) = Left "There is var at 2"
rem2 (F (F x)) = pure (F x)

add2 :: Var a -> Identity (Var (Var a))
add2 B = pure $ B
add2 (F x) = pure $ F (F x)
```

Решение не является оптимальным, тк можно пройти по всему терму единожды и применить эти операции сразу, но возрастет сложность генерации/написания такого кода.

Для решения этой задачи написан модуль Solver[6].

По сути мы либо имеем больший контекст и из него получаем меньший, либо наоборот. Хотим делать меньше swap'ов.

Рассмотрим случай приведения большего контекста к меньшему, "[x, y, z]" к "[y, x]". Мы идем справа налево, тк наиболее близкая связанная переменная наиболее правая. Удаляем те переменные которых нет в контексте к которому мы хотим прийти, таким образом обеспечиваем меньше вызовов к разным функциям rem[7]. Затем просто применяем insertion на оставшихся контекстах. На количестве сгенерированных функций swap это не отразится.

5.4. Вывод типов и нормализация

Сам infer работает как описано в Секции 2.1. Функция nf пытается паттернматчиться, если это не выходит, то данная редукция неприменима[8].

[6]Стоит отметить что функции swap, rem и ad d должны быть сгенерированы и для этого ведется подсчёт в монаде кодогенерации путем записи максимального индекса. Следовательно функция swap дороже, тк мы генерируем C_2^i функций. Именно поэтому алгоритм пытается использовать как можно меньше разных функций.

[7]Мы не можем удалить переменную из контекста, если она присутствует в терме. Монада TC обеспечивает обработку ошибок

[8]Такой паттернматчинг невозможен с использованием библиотеки bound[8], поэтому был написан модуль SimpleBound с обычными индексами де Брейна.

```
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]
```

```
infer ctx (App v1 v2 v3)
  = do v4 <- infer ctx v2
       v5 <- pure (nf v4)
       v6 <- infer ctx v1
       checkEq (Pi v5 (toScope (fromScope v3))) v6
       checkT ctx TyDef v5
       checkT (consCtx v5 ctx) TyDef (fromScope v3)
       infer ctx v1
       infer ctx v2
       pure (instantiate v2 (toScope (fromScope v3)))
```

Listing 2: Пример правила вывода и части сгенерированной функции infer, соответствующей этому правилу

5.5. Генерация кода

Генерация кода происходит с использованием библиотеки haskell-src-exts[9], которая дает нам функции генерации и манипуляции ACT Haskell.

Тк большинство кода используемого для проверки не зависит от специфицированного языка, мы просто модифицируем написанный от руки модуль LangTemplate. В нем нужно определить функции приведения в нормальную форму и вывода типов. Также нужно определить тип данных термов и определить монадическое действие на типе данных термов.

Всё остальное либо генерируется с помощью Template Haskell[7] — instance Traversable, Functor, Eq, Show, Foldable[9], либо написано от руки с вызовами функций nf или infer.

```
emptyCtx :: (Show a, Eq a) => Ctx a
emptyCtx x = Left $ "Variable not in scope: " ++ show x

consCtx :: (Show a, Eq a) => Type a -> Ctx a -> Ctx (Var a)
consCtx ty ctx B = pure (F <$> ty)
```

[9]Foldable дает нам функцию toList, которая возвращает свободные переменные терма, Traversable позволяет применять функции swap, rem и add к переменным обходя весь терм.

```
consCtx ty ctx (F a)  = (F <$>) <$> ctx a

checkT :: (Show a, Eq a) => Ctx a -> Type a -> Term a -> TC ()
checkT ctx want t = do
  have <- infer ctx t
  when (nf have /= nf want) $ Left $
    "type mismatch, have: " ++ (show have) ++ " want: " ++ (show
      want)
```

Listing 3: Проверка типов и контексты

Заключение

В рамках данной работы достигнуты следуюшие результаты:

- Определен язык спецификаций зависимых языков с дальнейшей возможностью генерации тайпчекера.
- Реализована генерации структур данных представления языка с использованием индексов де Брюйна на уровне типов и функций манипуляции этими структурами со значительным использованием кодогенерации.
- Реализованы генерация функций приведения термов специфицированного языка в нормальную форму и проверки типов.

Существует несколько направлений развития данной работы:

- Можно реализовать поддержку определения функций над термами языка.
- Дать пользователю определять функции на уровне языка спецификации.
- Поддержать возможность композиции спецификации языков — тогда можно будет собирать языки из частей как предложено в [4].

Список литературы

[1] Agda programming language. — Access mode: http://wiki.portal.chalmers.se/agda/pmwiki.php (online; accessed: 25.05.2017).

[2] Alex: A lexical analyser generator for Haskell. — Access mode: https://www.haskell.org/alex/ (online; accessed: 25.05.2017).

[3] Happy, The Parser Generator for Haskell. — Access mode: https://www.haskell.org/happy/ (online; accessed: 25.05.2017).

[4] Isaev Valery. Algebraic Presentations of Dependent Type Theories. — arxiv : math.LO, cs.LO, math.CT/http://arxiv.org/abs/1602.08504v3.

[5] 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.

[6] Pierce Benjamin C. Advanced Topics in Types and Programming Languages. — The MIT Press, 2004. — ISBN: 0262162288.

[7] Template Haskell. — Access mode: https://wiki.haskell.org/Template_Haskell (online; accessed: 25.05.2017).

[8] bound: Making de Bruijn Succ Less. — Access mode: https://hackage.haskell.org/package/bound (online; accessed: 25.05.2017).

[9] haskell-src-exts: Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer. — Access mode: https://hackage.haskell.org/package/haskell-src-exts (online; accessed: 25.05.2017).

Приложения

A. Доказательство корректности функции sort

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

```
-- Определяемпредикатпринадлежностиэлементасписку .
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)

-- Определяемпредикат xs ⊠ ys, означающийсписок " xs
   являетсяподсписком 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 длялюбогосписка 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 ⊠ 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)
```

B. Проверка использования метапеременных в функциональном символе

В секции 4 описывался язык и ограничения налагаемые на спецификации.

Здесь описан алгоритм проверки использвания метапеременных в контекстах других метапеременных.