# Типизация функций для подмножества языка РЕФАЛ

Дмитрий П. Сырбу

МГТУ имени Н. Э. Баумана

V совместное рабочее совещание ИПС имени А.К. Айламазяна РАН и МГТУ имени Н.Э.Баумана по функциональному языку Рефал

16 июня 2022

# Постановка задачи

- Целью данной работы является описания подмножества КС грамматик замкнутое относительно теоретико-множественных операций, в рамках которого выразимы описания типов функций подмножества языка РЕФАЛ и сопоставление типов с образцом предложения.
- Будет предложена процедура нормализации грамматики, а также описан процесс верификации типов.

# Нотация типов

- Формат грамматики типов

  <имя-типа> ::= <типовое-выражение>

  где *<имя-типа>* – переменная вида σ-, τ- или ξ-, а *<типовое-выражение>* – регулярное выражение в формате РБНФ.

- Формат описания типов функций

  <Func <типовое-выражение>> == <типовое-выражение>

```
τ.Tree ::= (Leaf σ.Bool) | (Branch τ.Tree τ.Tree)
σ.Bool ::= True | False
```

# Правила в грамматике типов

- Для σ-типов в правой части правила должен быть набор альтернатив, включающий только символьные константы и другие σ-типы

```
σ.Bool ::= True | False;
```

- У ξ-типов справа может быть записано произвольное регулярное выражение

```
ξ.SgnNumber ::= {‘+’| ‘-’}? σ.Number;
```

- Правило для τ-типов – перечисление термов, где *терм* — символ, произвольное выражение в скобках, σ- или τ-тип.

```
τ.Tree ::= (Leaf σ.Bool) | (Branch τ.Tree τ.Tree);
```

# Нормальная форма грамматики типов

- Исходная грамматика разбивается на правильные типовые тройки

$$\tau.\text{Имя} ::= \sigma.\text{Имя} \mid (\xi.\text{Имя})$$

- Типовое выражение для $\sigma$-типа может быть либо пустым (@), либо перечислением константных символов.
- Типовое выражение для $\xi$-типа представляет собой регулярные выражения над алфавитом $\tau$-типов.

# Типы в нормальной форме. Пример

```
σ.[Leaf]::= Leaf;
τ.[Leaf]::= σ.[Leaf] | (ξ.[Leaf]);
ξ.[Leaf]::= @;

σ.[Branch]::= Branch;
τ.[Branch]::= σ.[Branch] | (ξ.[Branch]);
ξ.[Branch]::= @;

σ.Tree ::= @;
τ.Tree ::= σ.Tree | (ξ.Tree);
ξ.Tree ::= τ.[Leaf] τ.Bool | τ.[Branch] τ.Tree τ.Tree;

σ.Bool ::= True | False;
τ.Bool ::= σ.Bool | (ξ.Bool);
ξ.Bool ::= @;
```

# Типы в нормальной форме. Пример

```
ξ.A ::= '+'? σ.CHAR+
```

```
τ.A ::= σ.A | ( ξ.A )

σ.A ::= @

ξ.A ::= {τ.['+'] | #} τ.CHAR  τ.CHAR*


τ.['+'] ::= σ.['+'] | ( ξ.['+'] )

σ.['+'] ::= '+'

ξ.['+'] ::= @
```

# Ортогональная нормальная форма грамматики

- Ортогональные типы описывают непересекающиеся множества значений.
- σ-типы — множества констант РЕФАЛа.
- ξ-типы — регулярные языки в алфавите ортогональных τ-типов
- τ-типы — объединение σ-типа и скобочного терма с ξ-типом внутри.

```
τ.⟨ortho⟩  ::= σ.⟨ortho⟩  | (ξ.⟨ortho⟩)
```

- Типы грамматики в нормальной форме выражаются как объединения соответствующих ортогональных типов.

# Построение ортогональных типов

- Типовые тройки в исходной грамматике нумеруются.
- Генерируется $2^N$ строк длины $N$, где $N$ — количество типовых троек в исходной грамматике, состоящих из последовательности `'0'` или `'1'`.
- Типы грамматики в нормальной форме выражаются как объединение типов соответствующего вида из с `'1'` в i-ой позиции, где `i` — номер типовой тройки

```
τ. ⟨i⟩  ::= … | τ. ⟨…⟩ 1 ⟨…⟩  | …
σ. ⟨i⟩  ::= … | σ. ⟨…⟩ 1 ⟨…⟩  | …
ξ. ⟨i⟩  ::= … | ξ. ⟨…⟩ 1 ⟨…⟩  | …
```

# Вычисление ортогональных типов

- Значение ортогональных типов вычисляется следующим образом

$$m.bitstr = \bigcap_{i=1}^{N} SET(m, i, bitstr)$$

где $m$ – вид типа (σ-, или ξ-), $bitstr$ – имя ортогонального типа, функция $SET$ определяется следующим образом

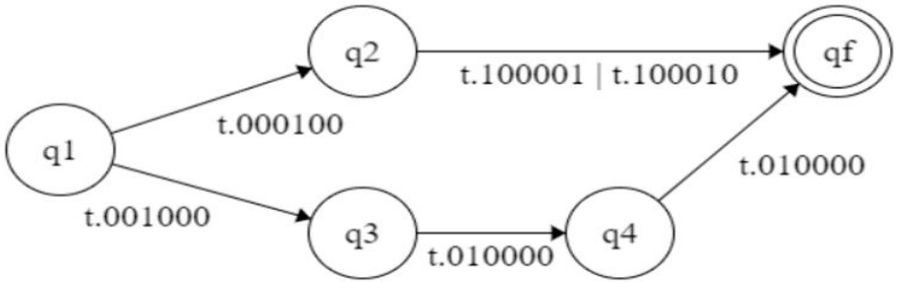
```
SET(mode, i, bitstr) =  m.i, bitstr[i] == '1'
                      | ~m.i, bitstr[i] == '0'
```

где ~`m.i` – дополнение множества `m.i`

- Типовая тройка с пустыми σ-, и ξ- компонентами может быть безопасно удалена из грамматики.

# ОНФ. Пример

```
τ.[Leaf] ::= τ.000100;
τ.[Branch] ::= τ.001000;
τ.Tree ::= τ.010000;
τ.Bool ::= τ.100001 | τ.100010;
τ.[True] ::= τ.100001;
τ.[False] ::= τ.100010;
ξ.Tree ::= t.000100 {t.100001 | t.100010} | t.001000 t.010000 t.010000
```



# Инкрементное построение ОНФ

1. Построение нормальной формы исходной грамматики.
2. Разбиение грамматики в НФ на подмножества типов (типовых троек)

$$U_1, U_2, \ldots, U_k$$

   таким образом, чтобы для $i<j$ $\xi$-типы из $U_i$ не зависели от $\tau$-типов из $U_j$.
3. Построение ОНФ для $U_1$.
4. Последовательное расширение аккумулируемой ОНФ грамматики с очищением от пустых типов.

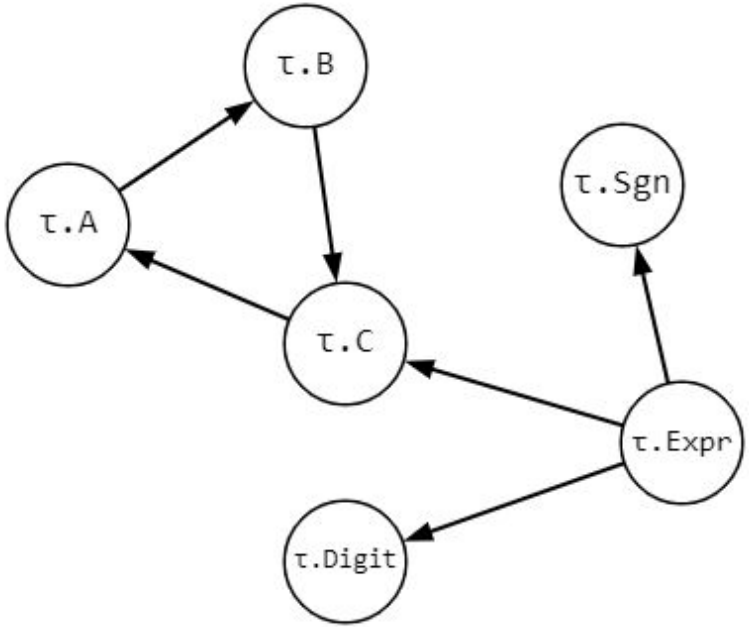
# Разбиение грамматики типов. Пример

- Рассмотрим следующую грамматику

```
τ.A ::= ‘a’ | (τ.B);

τ.B ::= ‘b’ | (τ.C);

τ.C ::= ‘c’ | (τ.A);

σ.Sgn ::= '+' | '-';

σ.Digit ::= '0' | '1';

ξ.Expr ::= σ.Sign? σ.Digit+ | t.C;
```

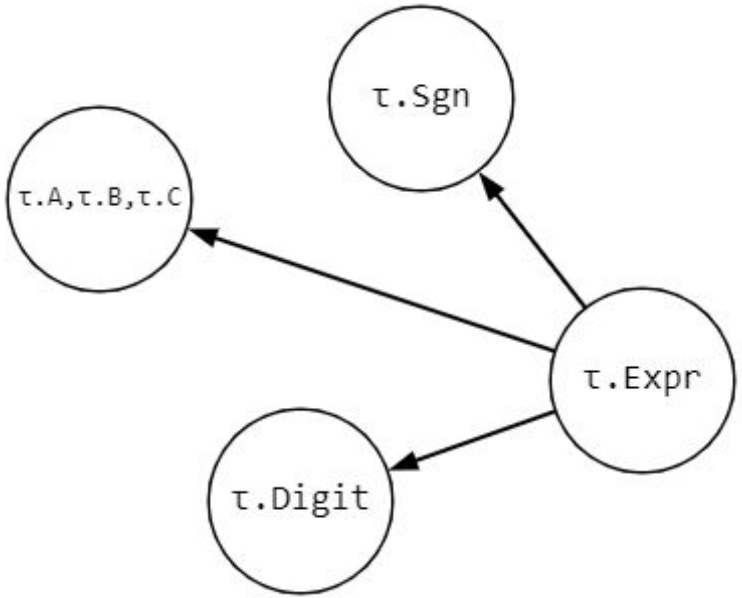
# Разбиение грамматики типов. Пример

- Граф типовых связей будет иметь вид



# Разбиение грамматики типов. Пример

- Конденсация графа типов



# Задача верификации типов функций

1. Построение ортогональной нормальной формы грамматики типов.
2. Описание типов функций приобретает вид

```
<Func ξ.Arg[Func]> == ξ.Res[Func]
```

3. Для каждого выражения-образца *Sentence* предложений функций происходит сопоставление с *ξ.Arg[Func]*.
4. Построение фактического типа правой части предложения.
5. Проверка на вложение фактического типа предложения в формальный тип *ξ.Res[Func]*.

# Сопоставление ξ-типа с образцом предложения

- Процедура сопоставления типа с образцом представляет собой решение системы уравнений вида

  $$RegLang : Pattern$$

  где $RegLang$ является конечным автоматом, допускающим тип, заданный регулярным языком, а $Pattern$ – *L-выражение* образца предложения.

- Система уравнений инициализируется единственным уравнением

  $$\xi.Arg[Func] : Sentence$$

- Результатом сопоставление отображения s- и t-переменных образца в ортогональные $\tau$-типы, а e-переменных в конечные автоматы над ортогональными $\tau$-типами.

# Тестирование. Верификация

```
σ.Bool = True | False;
<IsPalindrome ξ.Word > == σ.Bool


IsPalindrome {
  s.OneSymbol = True;
              = True;
  s.Equal e.Middle s.Equal = <IsPalindrome e.Middle>;
  e.Other = False;
  s.A s.B = '1';
 }
```

# Тестирование. Верификация

```
IsPalindrome { e.Word }  => { s.Bool }

[SUCCESS]: s.A = True;

[SUCCESS]: / ПУСТО / = True;

[SUCCESS]: s.A e.M s.A = <IsPal e.M>;

[SUCCESS]: e.Other = False;

[FAILURE]: s.A s.B = '1' doesn't satisfy type signature: e.Word  =>  s.Bool
```

# Спасибо за внимание