Содержание

1 Описание предприятия

Институт программных систем был создан в апреле 1984 года как Филиал Института проблем кибернетики АН СССР по решению Правительства СССР, направленному на развитие вычислительной техники и информатики в стране. Руководителем ФИПК АН СССР был назначен д.т.н., профессор Альфред Карлович Айламазян. В 1986 году Филиал Института проблем кибернетики был преобразован в Институт программных систем АН СССР, а в 2008 году институту было присвоено имя его первого директора профессора А.К. Айламазяна. С момента создания основными научными направлениями деятельности института являлись:

- высокопроизводительные вычисления;
- программные системы для параллельных архитектур;
- автоматизация программирования;
- телекоммуникационные системы и медицинская информатика.

Сегодня Институт программных систем имени А.К. Айламазяна РАН объединяет пять исследовательских центров и является динамично развивающимся коллективом, работающим в Отделении нанотехнологий и информационных технологий РАН.

Исследовательские центры ИПС РАН:

- Исследовательский центр мультипроцессорных систем (ИЦМС);
- Исследовательский центр медицинской информатики (ИЦМИ Интерин);
- Исследовательский центр искусственного интеллекта (ИЦИИ);
- Исследовательский центр процессов управления (ИЦПУ);
- Исследовательский центр системного анализа (ИЦСА).

2 Индивидуальное задание

Практическое задание заключается в реализации базового шага алгоритма рекомпрессии Артура Ежа для решения уравнений в словах [1]. Предполагается реализация двух программных модулей:

- `PairComp` — сжатие всех вхождений в уравнение данной пары констант с ветвлением исходного уравнения;
- `BlockComp` — сжатие всех вхождений данной константы в блоки максимальной длины, также с порождением новых ветвей.

Указанные модули являются частью интерактивного режима с визуализацией, который позволит удобно исследовать вопрос эффективности алгоритма Ежа с применением различных эвристик. В частности, для управления рабочим процессом интерактивного режима необходимы две вспомогательные функции:

- `Pick`, выбирающая одну из ветвей уравнения для дальнейшей работы;
- `SubstIndex` для подстановки переменных индексов в константы-блоки.

Формат и назначение всех указанных функций будут подробны описаны в разделах 5-8, а листинги — представлены в 10. Вся необходимая теоретическая база сосредоточена в разделе 4. Теории, вообще, уделено много внимания в настоящем отчёте: как и программная, теоретическая часть тоже выводилась и систематизировалась в процессе практики.

В качестве языка реализации был выбран Рефал-5, отлично подходящий для анализа и преобразования текстов вообще и математических структур в частности. Весь исходный код и различные полезные материалы, относящиеся к практике, доступны в репозитории github.com/TonitaN/Misc-Lectrues.

3 Цели

Условием успешного прохождения производственной практики я определил для себя осуществление следующих целей.

- *Качественная реализация программных модулей* `PairComp`, `BlockComp` *и функций* `Pick`, `SubstIndex` *для эффективной работы интерактивного режима.*

Под словом *качественная* понимается ясность и структурированность, быстродействие и результативность написанной программы.

- *Исследование и систематизация теоретической части практической применимости алгоритма Ежа.*

С момента своей публикации (2012 год [1]) техника повторного сжатия рассматривалась как хороший *теоретический*, но не *практический* метод решения уравнений в словах. Однако с применением даже некоторых эвристик — *разрезание уравнения* и *подсчёт баланса букв*, разработкой которых занимается мой научный руководитель А.Н. Непейвода — техника рекомпрессии может стать мощным прикладным инструментом в своей области.

- *Изучение языка Рефал-5, различных методов программирования, характерных для данного языка и функциональной парадигмы вообще, с применением хороших практик написания Рефал-кода.*

Специализация языка на работе со строковыми моделями влечёт совершенно отличный от многих современных ЯП способ *мышления* и, как следствие, написания программ — тем лично мне и интересен Рефал. Для его грамотного применения в работе стараюсь придерживаться рекомендаций А.В. Коновалова [4], [5] по оформлению кода.

4 Теоретическая основа

При описании структур данных используется формальная БНФ-грамматика. Все литералы имеют один из трёх префиксов: `s.`, `t.` или `e.`, соответствующие *символам*, *термам* и *объектным выражениям* языка Рефал.

Помимо явных символьных слов (`AreEqual`, `Var` и т.п.) используются следующие терминалы:

- `s.NUMBER` — любая макроцифра;
- `s.CHAR` — любой символ;
- `s.WORD` — любое символьное слово.
- `e.ANY` — любое объектное выражение.

В определении литерала заместо `::=` может использоваться инфиксный оператор `:` в случае, когда *лишь в данном контексте* выражение слева представляется как выражение справа. Символы `*` и `+` означают повторение литерала ноль и более и один и более раз соответственно.

4.1 Представление уравнения

Рассматриваются уравнения в словах с алфавитом переменных Ξ и алфавитом констант Σ. Применение алгоритма Ежа предполагает хранение дополнительной информации об уравнении: какие константы являются результатом сжатия блоков, какие переменные не могут быть пусты и т.д. Поэтому *уравнение* представляется структурой данных `t.Eq`:

```
t.Eq ::= ((AreEqual (e.LHS) (e.RHS)) (e.Constrs) (e.Conds)),
```

где `e.LHS: t.Elem*` и `e.RHS: t.Elem*` представляют его левую и правую части, `e.Constrs: t.Constr*` — ограничения на переменные, а `e.Conds: t.Cond*` содержит условия на константы.

Элемент `t.Elem` обобщённо представляет *константу* или *переменную*, соответствующие структурам данных `t.Const` и `t.Var`:

```
t.Elem ::= t.Const | t.Var,
t.Const ::= (s.CHAR s.NUMBER),
t.Var ::= (Var s.CHAR).
```

4.2 Ограничения на переменные

Ограничения на переменные представляются структурой `t.Constr` и описываются в конъюнктивной нормальной форме. Литералами дизъюнкций являются *рестрикции* — отрицательные условия на переменные типа `t.Restr`, подразделяющиеся на *краевые* `t.BoundRestr` и рестрикции *на пустоту* `t.EmptyRestr`:

```
t.Restr ::= t.BoundRestr | t.EmptyRestr.
```

Краевые рестрикции бывают *префиксными* `t.PrefixRestr` и *суффиксными* `t.SuffixRestr`. Они указывают, на какие константы не может начинаться или кончаться данная переменная.

```
t.BoundRestr ::= t.PrefixRestr | t.SuffixRestr,
t.PrefixRestr ::= (not t.Const starts t.Var),
t.SuffixRestr ::= (not t.Const ends t.Var).
```

Рестрикция на пустоту `t.EmptyRestr` сообщает, как следует, о невозможности обращения данной переменной в пустое слово ε:

```
t.EmptyRestr ::= (not empty t.Var).
```

В частности, будем называть переменную **непустой**, если для неё существует такая рестрикция.

Ограничения на переменные `t.Constr` могут содержать одну или две рестрикции, в соответствии с чем называются *тривиальными* `t.TrivialConstr` и *нетривиальными* `t.NonTrivialConstr`. В программе используется только четыре вида ограничений:

```
t.Constr ::= t.TrivialConstr | t.NonTrivialConstr
t.TrivialConstr ::= (OR t.Restr),
t.NonTrivialConstr ::= (OR t.SuffixRestr t.PrefixRestr).
```

Говоря далее *ограничения* мы, как правило, будем подразумевать именно тривиальные, если не оговорено противное, а также будем экстраполировать тип рестрикции на ограничение её содержащее: префиксное ограничение, ограничение на пустоту и т.д.

4.3 Условия на константы

Условие на константу `t.Cond` содержит сжимаемые блоки и соотвествующую этому сжатию константу. Вообще, *блок* `t.Block` обозначает степень константы и представляется в виде

```
t.Block ::= (t.Const t.Exp* (const s.NUMBER)),
```

где `t.Const` — сжимаемая константа, `(const s.NUMBER)` — обязательный константный показатель, а `t.Exp ::= (s.WORD s.NUMBER)` — переменный показатель степени. Таким образом, условие на константу представляется в виде

```
t.Cond ::= (t.Const is t.Block+).
```

В программе используется только два типа условий: *парное* `t.PairCond` и *условие на блок* `t.BlockCond`:

```
t.PairCond ::= (t.Const is (t.Const (const 1)) (t.Const (const 1))),
t.BlockCond ::= (t.Const is t.Block).
```

4.4 Краевые элементы

Мы хотим применять `Pair`- и `Block`-сжатие не только к обыкновенным константам, но и тем, что уже являются результатом сжатия в блоки. Для корректной обработки ограничений необходимо аккуратно отслеживать, на какие константы не может начинаться или кончаться та или иная переменная.

Пусть $\alpha, \gamma \in \Sigma$. Скажем, что $\alpha \in First(\gamma)$, если существует цепочка условий на константы такая, что

$$\gamma = \beta_1^{i_1} B_1, \quad \beta_1 = \beta_2^{i_2} B_2, \quad \ldots, \quad \beta_{n-1} = \alpha^{i_n} B_n,$$

где $\beta_i \in \Sigma,\ i = 1, 2, \ldots, n-1$, и $B_j \in \Sigma^*,\ j = 1, 2, \ldots, n$. Симметрично определяется множество $Last$-элементов константы (вообще, делая далее какое-либо утверждение о $First$-элементах константы будем считать, что оно симметрично выполняется и для её $Last$-множества).

Иногда удобно использовать обобщение введённых выше понятий: будем говорить, что константа α является **краевым** элементом константы γ, если $\alpha \in First(\gamma)$ или $\alpha \in Last(\gamma)$. Обозначение: $\alpha \in Bound(\gamma)$.

4.5 Слабые и сильные ограничения

В соответствии с введённым в разделе 4.4 понятием краевого элемента, для данной переменной X и константы γ мы можем определить *иерархические отношения* на множествах префиксных и суффиксных ограничений X с константами $First(\gamma)$ и $Last(\gamma)$ соответственно.

Пусть даны два, например, префиксных ограничения с константами α и β (для суффиксных — аналогично). Если $\alpha \in First(\beta)$, то α-ограничение является **более сильным**. В то же время β-ограничение есть **более слабое** по сравнению с первым. Имеет смысл хранить в уравнении лишь самые сильные ограничения.

4.6 Избыточные рестрикции и условия

В результате некоторых действий краевые рестрикции могут стать неактуальными (с рестрикциями на пустоту этого не происходит), вследствие чего их можно удалить из уравнения. Будем называть краевую рестрикцию **избыточной** в двух случаях:

- участвующей в ней переменной нет в уравнении;
- участвующей в ней константы нет в уравнении, а также в правых частях условий на константы.

Условие на константу также может стать **избыточным**. Это происходит в следующих ситуациях:

- константа в левой части условия не участвует в уравнении и не является $Bound$-элементом какой-либо другой константы;
- условие имеет парный тип, обеих констант его правой части нет в уравнении и на них нет каких-либо других условий.

4.7 Обработка нетривиальных ограничений

Нетривиальные ограничения на переменные также могут подвергаться обработке. Пусть уравнение содержит

```
t.NonTrivialConstr: (OR t.SuffixRestr t.PrefixRestr).
```

Если в уравнении найдётся равное или более сильное в сравнении с `(OR t.SuffixRestr)` или `(OR t.PrefixRestr)` краевое ограничение, то `t.NonTrivialConstr` **выполняется тривиально**, и его можно удалить. Это же происходит и в случае, когда избыточна хотя бы одна из рестрикций `t.SuffixRestr` и `t.PrefixRestr`.

Будем называть префиксную рестрикцию с константой $\alpha \in \Sigma$ **зависимой** относительно некоторого $\beta \in \Sigma$, если $\alpha \in First(\beta) \cup \{\beta\}$. В противном случае рестрикция называется **независимой**.

Иногда нам потребуется модифицировать нетривиальные ограничения. Например, мы хотим выполнить *подстановку*

$$X \to X\alpha, \quad X \in \Xi, \quad \alpha \in \Sigma$$

при наличии такого ограничения с зависимой относительно α рестрикцией. Мы **вынуждаем** выполнение второй рестрикции, превращая тем самым нетривиальное ограничение в тривиальное.

4.8 Подстановки

В алгоритме повсеместно требуется выполнять нерекурсивные подстановки выражений. Определим для этого специальный формат `t.Subst`:

```
t.Subst ::= (assign (e.Old) (e.New)),
```

где выражение `e.Old: e.ANY` — заменяемое, а `e.New: e.ANY` — новое.

Практически всегда подстановка заключается в извлечении у переменной слева или справа константы и обращении этой переменной в пустое слово. Например,

$$X \to X\alpha,\ Y \to \beta Y,\ Z \to \varepsilon, \quad X, Y, Z \in \Xi, \quad \alpha, \beta \in \Sigma.$$

Такие подстановки называются соответственно *суффиксными*, *префиксными* и *пустыми*.

В контексте конкретной задачи подстановки могут быть **элементарными** и **композитными**. Это означает, что для достижения некоторой цели могут потребоваться одна или несколько одновременно выполняющихся подстановок.

4.9 Нормальное уравнение

Назовём **сокращением** уравнения удаление всех совпадающих префиксных и суффиксных элементов его левой и правой частей. Два уравнения будем называть **эквивалентными**, если их сокращение производит уравнения с одинаковыми левыми и правыми частями.

Скажем, что уравнение **нормальное**, если оно несократимо, не содержит слабых и тривиально выполняющихся ограничений и избыточных рестрикций и условий. Также потребуем, чтобы в нормальном уравнении все ограничения были отсортированы, равно как и показатели степеней `t.Exp` условий на блок. В реализации используется быстрая сортировка, принимающая функцию-компаратор. Определив такие компараторы для ограничений и показателей степеней появляется возможность эффективно сортировать указанные элементы.

Нормальные уравнения представляют особый интерес для нас. Именно их возращают и ожидают получить на вход функции `Pick`, `SubstIndex`, `PairComp` и `BlockComp`.

5 Функция `Pick`

Скажем, что решение уравнения **неминимальное**, если его левая и правая части состоят из непустых переменных и только них.

Функция `Pick` принимает номер уравнения `s.Number: s.NUMBER` и сами уравнения `e.Eqs: t.Eq+`. Все переменные уравнения под номером `s.Number`, для которых возможна подстановка в пустое слово, обращает в ε и возвращает

- `Success`, если новое уравнение эквивалентно уравнению $\varepsilon = \varepsilon$;
- `NotMinimal`, если решение такого уравнения неминимальное;
- новое нормальное уравнение в остальных случаях.

`Pick` имеет довольно простую реализацию. Из множества всех переменных уравнения вычитаются непустые, для оставшихся же генерируются и выполняются подстановки в пустое слово. Если получившееся уравнение эквивалентно $\varepsilon = \varepsilon$, возвращаем `Success`. В противном случае получаем множество констант нового уравнения. Если это множество пусто, возвращаем `NotMinimal`, иначе — получившееся нормализованное уравнение.

6 Функция SubstIndex

Функция `SubstIndex` принимает индекс `s.Index: s.WORD`, показатели `e.Exps: t.Exp+`, уравнение `t.Eq` и выполянет подстановку показателей на место данного индекса.

Для всякого условия на блок с показателем `(s.Index s.NUMBER)` производится замена этого показателя на множество `e.Exps`, каждый элемент которого домножен на постоянную `s.NUMBER`. Обычно после этого шага порядок показателей нарушается — их нужно отсортировать.

В результате подстановки может получиться условие вида

```
t.Cond: (t.Const is (t.BlockConst (const 0))).
```

Такое условие *коллапсирует*, так как коллапсирует содержавшийся в нём блок. Нам полезно на месте генерировать подстановки констант из левых частей таких условий в ε: позже мы осуществим эти подстановки в левой и правой частях уравнения.

Также мы могли получить условия с разными левыми, но одинаковыми правыми частями:

```
t.Cond1: (t.Const1 is (t.BlockConst e.Exps (const s.NUMBER))),
t.Cond2: (t.Const2 is (t.BlockConst e.Exps (const s.NUMBER))).
```

В таком случае мы сохраняем лишь одно условие (в реализации — с лексикографически наименьшей константой), например, `t.Cond1`, и генерируем замену константы второго условия `t.Const2` на `t.Const1`.

Теперь выполняем в уравнении все сгенерированные подстановки (констант коллапсировавших и заменённых условий). Остаётся лишь нормализовать полученное уравнение.

7 Функция PairComp

7.1 Вхождения сжимаемой пары

Рассмотрим возможности появления пары $\alpha\beta \in \Sigma^+$ в уравнении. Пусть одна из его частей представляется как

$$\Phi\alpha\beta\Psi, \quad \Phi, \Psi \in (\Sigma \cup \Xi)^*. \tag{1}$$

Здесь получаем **явное** вхождение исходной пары. Если часть уравнения имеет вид

$$\Phi X\beta\Psi, \quad X \in \Xi, \quad \Phi, \Psi \in (\Sigma \cup \Xi)^*, \tag{2}$$

мы можем рассмотреть случай, когда решение X оканчивается на α, т.е. $X = X\alpha$. Выполнив подстановку $X \to X\alpha$ (если это возможно) получим новое явное вхождение пары $\alpha\beta$. Поэтому нам интересна ситуация (2). Будем называть такое вхождение пары **перекрёстным**.

Случай

$$\Phi\alpha Y\Psi, \quad Y \in \Xi, \quad \Phi, \Psi \in (\Sigma \cup \Xi)^* \tag{3}$$

симметричен предыдущему: имеем перекрёстное вхождение пары $\alpha\beta$ и можем получить явное, выполнив подстановку $Y \to \beta Y$.

Наконец, часть уравнения может представляться в виде

$$\Phi XY\Psi, \quad X, Y \in \Xi, \quad \Phi, \Psi \in (\Sigma \cup \Xi)^*. \tag{4}$$

В таком случае новое явное вхождение появляется (при отсутствии соответствующих ограничений) с выполненными одновременно подстановками $X \to X\alpha$ и $Y \to \beta Y$.

Пара $\alpha\beta$ также может входить в уравнение **неявно**, не являясь началом или концом решения, а находясь полностью внутри него. Такая ситуация в силу неминимальности нам неинтересна.

7.2 Существенные подстановки

Обратим внимание на случаи (2)-(4) раздела 7.1. Здесь краевые подстановки в своём контексте порождают новые явные вхождения в уравнение пары $\alpha\beta$, поэтому они называются **существенными**. Существенные элементарные и композитные подстановки включают одну и две подстановки соответственно. Например, подстановки в (2) и (3) являются существенными элементарными, а в (4) — существенной композитной.

В реализации нам потребуется понятие **специальной** подстановки: таковыми будем называть существенные элементарные подстановки, являющиеся частью какой-либо существенной композитной подстановки.

Описанные выше подстановки являются *непустыми*, но и *пустые* подстановки могут порождать новые явные пары, также называясь в этом случае существенными. Пусть часть уравнения представляется в виде

$$\Phi\alpha\Omega\beta\Psi, \quad \Omega \in \Xi^+, \quad \Phi, \Psi \in (\Sigma \cup \Xi)^*, \tag{5}$$

где хотя бы одна переменная в Ω может быть пуста. Пусть такой переменной будет $W \in \Xi$. Если максимальный блок W^i, $i \in \mathbb{N}$, следует непосредственно за α, обратив переменную в ε мы получим или новую явную пару $\alpha\beta$, или новую перекрёстную пару при соседстве α с новой переменной, следующей прямо за W^i. Получаем симметричную ситуацию, если W^i непосредственно предшествует β. Если же W^i находится внутри Ω, подстановка в пустое слово даст новое соседство двух переменных, следовательно — новое перекрёстное вхождение, где явная пара может быть порождена существенной композитной подстановкой.

Во всех случаях имеет смысл совершать подстановку $W \to \varepsilon$. Мы приходим к следующей **лемме**: всякое подмножество существенной композиции пустых подстановок существенно.

Аналогично (5) разбираются ситуации

$$\Phi X \Omega \beta \Psi, \quad X \in \Xi, \quad \Omega \in (\Xi/X)^+, \quad \Phi, \Psi \in (\Sigma \cup \Xi)^*, \tag{6}$$

$$\Phi \alpha \Omega Y \Psi, \quad Y \in \Xi, \quad \Omega \in (\Xi/Y)^+, \quad \Phi, \Psi \in (\Sigma \cup \Xi)^*, \tag{7}$$

$$\Phi X \Omega Y \Psi, \quad X, Y \in \Xi, \quad \Omega \in (\Xi/X/Y)^+, \quad \Phi, \Psi \in (\Sigma \cup \Xi)^*, \tag{8}$$

Важно, чтобы среди Ω была хотя бы одна переменная, для которой возможна пустая подстановка.

7.3 Опции `PairComp`

Для данного уравнения может обнаружиться множество пустых и непустых существенных подстановок. Вообще, не любая их комбинация ведёт к решению. Необходимо перебрать их всевозможные сочетания и уже среди них искать успешные.

Опция уравнения — легковесная абстракция для представления такого набора. В `PairComp` опция представляется типом `t.PairOption`:

```
t.PairOption ::= ((e.Substs) (e.Constrs)).
```

Здесь `e.Substs` хранит определённую комбинацию подстановок, а `e.Constrs` — накладываемые ограничения при этих подстановках. В совокупности ограничения могут быть противоречивы — такие опции отбрасываются сразу, не применяясь к уравнению. Остальные же *нормализуются* и производят новые уравнения. Нормализация `t.PairOption` означает удаление дубликатов, тривиально выполняющихся ограничений и конфликтующих рестрикций. Ограничения нормальной опции *модифицируются*: при префиксных подстановках удаляются независимые (зависимых на этом этапе быть не может — противоречие) префиксные и пустые ограничения, при суффиксных — симметрично. Наконец, к ограничениям опции прибавляются ограничения уравнения, чтобы в дальнейшем их уже не пришлось как-либо изменять.

7.4 Реализация `PairComp`

Функция принимает константу для замены `t.ReplConst`, элементы сжимаемой пары `t.Const1` и `t.Const2`, а также уравнение `t.Eq`. `PairComp` генерирует новую константу для замены и новые уравнения, выполняя различные комбинаций существенных пустых и непустых подстановок исходного уравнения `t.Eq`.

В начале рекурсивно генерируются ветви с выполняющимися и невыполняющимися существенными пустыми подстановками. Подстановки отбираются по одной, как описано в разделе 7.2. Для первой ветви сразу удаляются избыточные ограничения, возникшие с подстановкой переменной в ε, а к ограничениям второй ветви прибавляется непустота переменной.

Остановимся на какой-нибудь ветви. Как только на ней заканчиваются возможные существенные пустые подстановки, начинается поиск всех существенных непустых — элементарных и композитных. Они отбираются с учётом существующих краевых ограничений, поэтому на выходе получаем только возможные подстановки. Среди них удаляются дубликаты, а также определяются специальные подстановки.

Теперь можно генерировать первичные наборы опций, которые в дальнейшем будут декартово переменожаться. Говоря *подстановка выполняется* будем подразумевать осуществление данной подстановки без накладываемых ограничений. Говоря, что *подстановка не выполняется*, будем иметь в виду невыполнение подстановки с добавлением соответствующего ограничения.

Для всякой элементарной неспециальной подстановки генерируется **элементарный набор** из двух опций: в одной из них подстановка выполняется, а в другой — нет.

Для каждой композитной подстановки в зависимости от числа составляющих её специальных подстановок производится один из трёх **композитных наборов**.

1. *Обе* подстановки специальные. Набор содержит четыре опции: в первой выполняются обе подстановки, во второй выполняется одна и не выполняется другая, в третьей — наоборот, а в четвёртой не выполняется ни одна из подстановок.

2. *Одна* подстановка специальная, другая — нет. В наборе три опции: в первой выполняются обе подстановки, во второй выполняется специальная и не выполняется оставшаяся, в третьей просто не выполняется специальная.

3. *Ни одна* из подстановок не является специальной. Набор составляют две опции. В первой выполняются обе подстановки, а во второй не выполняется или одна, или другая. *Этот набор единственный во всей программе порождает нетривиальные ограничения.*

Производится декартово произведение полученных наборов. Пока есть два и более таких множества, берётся пара наборов, и их опции переменожаются. Результат — уже одно, а не два множества — возвращается в исходное семейство, и процедура повторяется.

Мы получили опции уравнения такими, какими мы определяли их в разделе 7.3. Нужно попытаться нормализовать эти опции: некоторые из них, возможно, будут удалены как противоречивые. Оставшиеся же дорабатываются как описывается в упомянутом разделе.

Наконец, опции применяются к уравнению. Если к этому моменту нет ни одной опции (изначально не было существенных непустых подстановок или все опции оказались противоречивы), искусственно применяется *пустая опция* вида `((/* no substs */) (/* no constrs */))`. Для каждой опции все её подстановки применяются к уравнению. Образовавшиеся явные пары сжимаются. Ограничения полученного уравнения подменяются ограничениями опции, и результат нормализуется.

8 Функция `BlockComp`

8.1 Сжатие переменной в блок

Предположим, что в уравнении нет тривиальных ограничений на его переменную $X \in \Xi$ (*нетривиальные ограничения не обрабатываются*). При сжатии X в блок $\alpha \in \Sigma$ необходимо рассмотреть два случая:

- переменная *коллапсирует* в блок:

$$X \to \alpha^i,\ i \in \mathbb{N} \cup \{0\}. \tag{9}$$

- происходит извлечение максимальных префиксных и суффиксных блоков у переменной:

$$X \to \alpha^i X \alpha^j,\ i, j \in \mathbb{N} \cup \{0\}. \tag{10}$$

  Теперь X не может начинаться или кончаться на α в силу максимальности извлечённых блоков, а также не может быть пустым (иначе переменная сжималась бы в блок)

$$X \neq \alpha X, \quad X \neq X\alpha, \quad X \neq \varepsilon. \tag{11}$$

Допустим, в уравнении есть префиксные ограничения на X, независимые относительно α (суффиксные ограничения здесь и далее обрабатываются симметричным образом). Случай (10) теперь ветвится:

- Префиксный блок пуст, извлекается только суффиксный:

$$X \to X\alpha^j,\ j \in \mathbb{N} \cup \{0\}.$$

  Исходные ограничения сохраняются с присоединением ограничений (11).

- Префиксный блок *не* пуст:

$$X \to \alpha^i X \alpha^j,\ i, j \in \mathbb{N} \cup \{0\}.$$

  Исходные ограничения удаляются, а (11) — добавляются.

Пусть вместе с присутствующими или отсутствующими независимыми префиксными ограничениями на X переменная также непуста. Это влияет только на сжатие в блок (9): подстановка заменяется на

$$X \to \alpha^{i+1},\ i \in \mathbb{N} \cup \{0\}, \tag{12}$$

а ограничения на переменную, если были в уравнении, удаляются.

Теперь рассмотрим случай, когда уравнение содержит только префиксные ограничения — зависимые и, возможно, независимые. Сжатие в блок (9) становится *невозможным*; оно заменяется на пустую подстановку $X \to \varepsilon$ с удалением

всех ограничений. Извлечение префиксного блока в (10) также незаконно, поэтому извлекается только суффиксный:

$$X \to X\alpha^j,\ j \in \mathbb{N} \cup \{0\}; \tag{13}$$

прежние ограничения сохраняются, а из новых (11) добавляются только $X \neq \varepsilon$ и $X \neq X\alpha$, так как $X \neq \alpha X$ равно или слабее по силе существующих зависимых префиксных ограничений.

Наконец, если X ещё и не пуст, коллапсирование в ε вообще блокируется, и остаётся только извлечение (13).

В Рефале сжатая в блок константа, например, `('A' 0)` при таких подстановках имеет вид

```
(('A' 0) Index (const 0)).
```

Здесь `Index` есть переменная пока что *безымянного* показателя степени. В будущем, как станут известны все такие безымянные блоки уравнения, на место `Index` встанет конкретный переменный показатель: например, `(i1 1)`.

8.2 Опции `BlockComp`

Опция `t.BlockOption`, как и `t.PairOption`, хранит комбинации подстановок и модифицируемые ограничения, с применением которых исходное уравнение порождает новые. Ограничения `block`-опции сразу будем делить на *добавляемые* к уравнению и *исключаемые* из него. Так делать, вообще, необязательно, но нынешняя реализация строится на этом подходе. Таким образом, `t.BlockOption` представляется в виде

```
((e.Substs) (e.AddedConstrs) (e.ExcludedConstrs)).
```

В отличие от набора `t.PairOption`, декартово произведение `block`-опций всегда совместимо, поскольку мы никак не обрабатываем нетривиальные ограничения (а если бы даже обрабатывали, недопустима ситуация с двумя зависимыми относительно сжимаемой константы рестрикциями, содержащимися в одном ограничении — т.е. мы всегда сможем выбрать *удовлетворяющую* нас рестрикцию). Тем не менее, в таких наборах всё так же нужно удалять тривиально выполняющиеся и избыточные в связи с коллапсированием переменной в блок или ε ограничения.

Наконец, к изменённым ограничениям опции прибавляются ограничения уравнения (для заменещения в дальнейшем).

8.3 Реализация `BlockComp`

Функция принимает константу для сжатия `t.BlockConst`, константу и индекс `t.NewConst` и `s.NewIndex` соответственно для введения условий и уравнение `t.Eq`.

Для каждой переменной уравнения генерируется первичный набор опций по правилам, описанным в разделах 8.1 и 8.2. Число опций в наборе варьируется

от одной пустой (если зависимые и префиксные, и суффиксные ограничения, а переменная непуста) до целых пяти в случае, когда префиксные и суффиксные ограничения независимы (здесь одна опция — коллапсирование переменной, другие четыре есть результат декартова произведения 2×2 опций с извлечением префиксных и суффиксных блоков).

Производится декартово произведение таких первичных наборов. Полученные опции нормализуются и применяются к уравнению.

После применения подстановок опции части уравнения содержат безымянные блоки, описанные в конце раздела 8.1. Такие стоящие рядом друг с другом или сжимаемой константой блоки *объединяются*, образуя новую степень `t.BlockConst`. В частности, две стоящие рядом исходные константы преобразуются в единый блок-квадрат.

После объединения всех указанных блоков безымянные индексы `Index` в лексикографическом порядке заменяются на конкретные переменные показатели степени. Полученные блоки обозначаются и переносятся в условия на константы.

Наконец, функция возвращает неиспользованные константу и индекс, а также нормализованное получившееся уравнение.

9 Тестирование

Реализация корректно отрабатывает на многих наборах входных данных. В разделе приведены некоторые unit-тесты, написанные при разработке каждой из четырёх функций.

9.1 Функция `Pick`

На вход подаются слудующие уравнения `e.Eqs`:

```
(
  (
    AreEqual
    ((Var 'X') ('A' 0))
    ((Var 'Y') ('B' 0) (Var 'Y'))
  )
  (/* no constrs */)
  (/* no conds */)
)
(
  (
    AreEqual
    (('A' 0) (Var 'X') ('A' 0))
    ((Var 'Y') ('A' 0) ('A' 0) (Var 'Y'))
  )
  (
    (OR (not empty (Var 'X')))
    (OR (not ('A' 0) starts (Var 'X')) (not ('B' 0) ends (Var 'X')))
  )
  (/* no conds */)
)
(
  (
    AreEqual
    ((Var 'X'))
    ((Var 'Y'))
  )
  (
    (OR (not empty (Var 'X')))
  )
  (/* no conds */)
)
(
  (
    AreEqual
    ((Var 'X') ('A' 0))
    (('A' 0) (Var 'Y'))
  )
```

```
  (
    (OR (not ('A' 0) ends (Var 'X')))
  )
  (/* no conds */)
)
```

В результате вызовов <Pick 1 e.Eqs> и <Pick 2 e.Eqs> получаем преобразованные уравнения, при <Pick 3 e.Eqs> — NotMinimal, а <Pick 4 e.Eqs> возвращает Success.

9.2 Функция SubstIndex

Замена индекса i1 на ((i2 2) (const 2)) в уравнении

```
(
  (
    AreEqual
    (('A' 2))
    (('A' 1) (Var 'X') ('A' 1))
  )
  (
    (OR (not empty (Var 'X')))
  )
  (
    (('A' 1) is (('A' 0) (i1 1) (const 0)))
    (('A' 2) is (('A' 0) (i2 2) (const 2)))
  )
)
```

производит уравнение

```
(
  (
    AreEqual
    (/* empty */)
    ((Var 'X') ('A' 1))
  )
  (
    (OR (not empty (Var 'X')))
  )
  (
    (('A' 1) is (('A' 0) (i2 2) (const 2)))
  )
)
```

А, например, композиция функций <SubstIndex i4 ((i1 3) (const 1)) t.Eq> и <SubstIndex i1 ((i2 2) (i3 1) (const 0)) t.Eq>, применённых к такому уравнению t.Eq

```
(
  (
    AreEqual
    (('A' 3) ('B' 0) (Var 'X') ('A' 3) (Var 'Y'))
    (('A' 2) ('B' 0) (Var 'X') (Var 'Z') ('A' 2))
  )
  (/* no constrs */)
  (
    (('A' 1) is (('A' 0) (i1 3) (const 1)))
    (('A' 2) is (('A' 0) (i2 6) (i3 3) (const 1)))
    (('A' 3) is (('A' 0) (i4 1) (const 0)))
  )
)
```

возвращает

```
(
  (
    AreEqual
    (('A' 2) (Var 'Y'))
    ((Var 'Z') ('A' 2))
  )
  (/* no constrs */)
  (
    (('A' 2) is (('A' 0) (i2 6) (i3 3) (const 1)))
  )
)
```

9.3 Функция `PairComp`

Сжатие пары `('A' 1) ('B' 0)` новой константой `('B' 1)` в уравнении

```
(
  (
    AreEqual
    (('A' 1) ('B' 0) ('C' 0) (Var 'X') ('B' 0))
    ((Var 'Y') (Var 'Z') ('B' 0) ('A' 1) (Var 'X'))
  )
  (
    (OR (not empty (Var 'Z')))
    (OR (not ('C' 0) ends (Var 'Y')))
    (OR (not ('A' 1) ends (Var 'X')) (not ('B' 0) starts (Var 'X')))
  )
  ((('A' 1) is (('B' 0) (const 1)) (('C' 0) (const 1))))
)
```

возвращает новую константу `('C' 1)` и производит 6 уравнений:

```
(
  /* X = A1 X, Z = Z A1, X != B0 X */
  (
    AreEqual
    (('B' 1) ('C' 0) (Var 'X') ('B' 1))
    ((Var 'Y') (Var 'Z') ('B' 1) ('A' 1) (Var 'X') ('A' 1))
  )
  (
    (OR (not ('C' 0) ends (Var 'Y')))
    (OR (not ('B' 0) starts (Var 'X')))
  )
  (
    (('A' 1) is (('B' 0) (const 1)) (('C' 0) (const 1)))
    (('B' 1) is (('A' 1) (const 1)) (('B' 0) (const 1)))
  )
)
(
  /* X = A1 X, Z != Z A1, X != B0 X */
  (
    AreEqual
    (('B' 1) ('C' 0) (Var 'X') ('B' 1))
    ((Var 'Y') (Var 'Z') ('B' 0) ('A' 1) (Var 'X') ('A' 1))
  )
  (
    (OR (not empty (Var 'Z')))
    (OR (not ('C' 0) ends (Var 'Y')))
    (OR (not ('A' 1) ends (Var 'Z')))
    (OR (not ('B' 0) starts (Var 'X')))
  )
  (
    (('A' 1) is (('B' 0) (const 1)) (('C' 0) (const 1)))
    (('B' 1) is (('A' 1) (const 1)) (('B' 0) (const 1)))
  )
)
(
  /* X != A1 X, Z = Z A1, X = B0 X */
  (
    AreEqual
    (('B' 1) ('C' 0) ('B' 0) (Var 'X') ('B' 0))
    ((Var 'Y') (Var 'Z') ('B' 1) ('B' 1) (Var 'X'))
  )
  (
    (OR (not ('A' 1) ends (Var 'X')))
    (OR (not ('C' 0) ends (Var 'Y')))
  )
  (
    (('A' 1) is (('B' 0) (const 1)) (('C' 0) (const 1)))
```

```
    (('B' 1) is (('A' 1) (const 1)) (('B' 0) (const 1)))
  )
)
(
  /* X != A1 X, Z = Z A1, X != B0 X */
  (
    AreEqual
    (('B' 1) ('C' 0) (Var 'X') ('B' 0))
    ((Var 'Y') (Var 'Z') ('B' 1) ('A' 1) (Var 'X'))
  )
  (
    (OR (not ('A' 1) ends (Var 'X')))
    (OR (not ('C' 0) ends (Var 'Y')))
    (OR (not ('B' 0) starts (Var 'X')))
  )
  (
    (('A' 1) is (('B' 0) (const 1)) (('C' 0) (const 1)))
    (('B' 1) is (('A' 1) (const 1)) (('B' 0) (const 1)))
  )
)
(
  /* X != A1 X, Z != Z A1, X = B0 X */
  (
    AreEqual
    (('B' 1) ('C' 0) ('B' 0) (Var 'X') ('B' 0))
    ((Var 'Y') (Var 'Z') ('B' 0) ('B' 1) (Var 'X'))
  )
  (
    (OR (not empty (Var 'Z')))
    (OR (not ('A' 1) ends (Var 'X')))
    (OR (not ('C' 0) ends (Var 'Y')))
    (OR (not ('A' 1) ends (Var 'Z')))
  )
  (
    (('A' 1) is (('B' 0) (const 1)) (('C' 0) (const 1)))
    (('B' 1) is (('A' 1) (const 1)) (('B' 0) (const 1)))
  )
)
(
  /* X != A1 X, Z != Z A1, X != B0 X */
  (
    AreEqual
    (('B' 1) ('C' 0) (Var 'X') ('B' 0))
    ((Var 'Y') (Var 'Z') ('B' 0) ('A' 1) (Var 'X'))
  )
  (
    (OR (not empty (Var 'Z')))
```

```
    (OR (not ('A' 1) ends (Var 'X')))
    (OR (not ('C' 0) ends (Var 'Y')))
    (OR (not ('A' 1) ends (Var 'Z')))
    (OR (not ('B' 0) starts (Var 'X')))
  )
  (
    (('A' 1) is (('B' 0) (const 1)) (('C' 0) (const 1)))
    (('B' 1) is (('A' 1) (const 1)) (('B' 0) (const 1)))
  )
)
```

9.4 Функция BlockComp

Сжатие в блок составной константы ('A' 1) уравнения

```
(
  (
    AreEqual
    (('A' 1) (Var 'X') (Var 'Y') ('B' 0))
    ((Var 'X') ('C' 0) ('A' 1) (Var 'Z'))
  )
  (
    (OR (not empty (Var 'Y')))
    (OR (not empty (Var 'Z')))
    (OR (not ('C' 0) ends (Var 'X')))
    (OR (not ('B' 0) ends (Var 'Y')))
    (OR (not ('A' 1) starts (Var 'X')))
  )
  ((('A' 1) is (('A' 0) (const 1)) (('B' 0) (const 1))))
)
```

с введением константы ('B' 1) и индекса i1 производит 6 новых уравнений с новыми константами и индексами:

```
/* X = empty, Y = A1^i Y, Z = A1^(j+1) */
(
  ('D' 1) i3
  (
    (
      AreEqual
      (('B' 1) (Var 'Y') ('B' 0))
      (('C' 0) ('C' 1))
    )
    (
      (OR (not empty (Var 'Y')))
      (OR (not ('B' 0) ends (Var 'Y')))
      (OR (not ('A' 1) starts (Var 'Y')))
    )
```

```
    (
      (('A' 1) is (('A' 0) (const 1)) (('B' 0) (const 1)))
      (('B' 1) is (('A' 1) (i1 1) (const 1)))
      (('C' 1) is (('A' 1) (i2 1) (const 2)))
    )
  )
)
/* X = empty, Y = A1^i Y, Z = A1^j Z A1^k */
(
  ('E' 1) i4
  (
    (
      AreEqual
      (('B' 1) (Var 'Y') ('B' 0))
      (('C' 0) ('C' 1) (Var 'Z') ('D' 1))
    )
    (
      (OR (not empty (Var 'Y')))
      (OR (not empty (Var 'Z')))
      (OR (not ('B' 0) ends (Var 'Y')))
      (OR (not ('A' 1) ends (Var 'Z')))
      (OR (not ('A' 1) starts (Var 'Y')))
      (OR (not ('A' 1) starts (Var 'Z')))
    )
    (
      (('A' 1) is (('A' 0) (const 1)) (('B' 0) (const 1)))
      (('B' 1) is (('A' 1) (i1 1) (const 1)))
      (('C' 1) is (('A' 1) (i2 1) (const 1)))
      (('D' 1) is (('A' 1) (i3 1) (const 0)))
    )
  )
)
/* X = X, Y = A1^i Y, Z = A1^(j+1) */
(
  ('D' 1) i3
  (
    (
      AreEqual
      (('A' 1) (Var 'X') ('B' 1) (Var 'Y') ('B' 0))
      ((Var 'X') ('C' 0) ('C' 1))
    )
    (
      (OR (not empty (Var 'X')))
      (OR (not empty (Var 'Y')))
      (OR (not ('C' 0) ends (Var 'X')))
      (OR (not ('A' 1) ends (Var 'X')))
      (OR (not ('B' 0) ends (Var 'Y')))
```

```
      (OR (not ('A' 1) starts (Var 'X')))
      (OR (not ('A' 1) starts (Var 'Y')))
    )
    (
      (('A' 1) is (('A' 0) (const 1)) (('B' 0) (const 1)))
      (('B' 1) is (('A' 1) (i1 1) (const 0)))
      (('C' 1) is (('A' 1) (i2 1) (const 2)))
    )
  )
)
/* X = X, Y = A1^i Y, Z = A1^j Z A1^k) */
(
  ('E' 1) i4
  (
    (
      AreEqual
      (('A' 1) (Var 'X') ('B' 1) (Var 'Y') ('B' 0))
      ((Var 'X') ('C' 0) ('C' 1) (Var 'Z') ('D' 1))
    )
    (
      (OR (not empty (Var 'X')))
      (OR (not empty (Var 'Y')))
      (OR (not empty (Var 'Z')))
      (OR (not ('C' 0) ends (Var 'X')))
      (OR (not ('A' 1) ends (Var 'X')))
      (OR (not ('B' 0) ends (Var 'Y')))
      (OR (not ('A' 1) ends (Var 'Z')))
      (OR (not ('A' 1) starts (Var 'X')))
      (OR (not ('A' 1) starts (Var 'Y')))
      (OR (not ('A' 1) starts (Var 'Z')))
    )
    (
      (('A' 1) is (('A' 0) (const 1)) (('B' 0) (const 1)))
      (('B' 1) is (('A' 1) (i1 1) (const 0)))
      (('C' 1) is (('A' 1) (i2 1) (const 1)))
      (('D' 1) is (('A' 1) (i3 1) (const 0)))
    )
  )
)
/* X = X A1^i, Y = A1^j Y, Z = A1^(k+1) */
(
  ('E' 1) i5
  (
    (
      AreEqual
      (('A' 1) (Var 'X') ('B' 1) (Var 'Y') ('B' 0))
      ((Var 'X') ('C' 1) ('C' 0) ('D' 1))
```

```
    )
    (
      (OR (not empty (Var 'X')))
      (OR (not empty (Var 'Y')))
      (OR (not ('A' 1) ends (Var 'X')))
      (OR (not ('B' 0) ends (Var 'Y')))
      (OR (not ('A' 1) starts (Var 'X')))
      (OR (not ('A' 1) starts (Var 'Y')))
    )
    (
      (('A' 1) is (('A' 0) (const 1)) (('B' 0) (const 1)))
      (('B' 1) is (('A' 1) (i1 1) (i2 1) (const 0)))
      (('C' 1) is (('A' 1) (i3 1) (const 0)))
      (('D' 1) is (('A' 1) (i4 1) (const 2)))
    )
  )
)
/* X = X A1^i, Y = A1^j Y, Z = A1^k Z A1^l */
(
  ('F' 1) i6
  (
    (
      AreEqual
      (('A' 1) (Var 'X') ('B' 1) (Var 'Y') ('B' 0))
      ((Var 'X') ('C' 1) ('C' 0) ('D' 1) (Var 'Z') ('E' 1))
    )
    (
      (OR (not empty (Var 'X')))
      (OR (not empty (Var 'Y')))
      (OR (not empty (Var 'Z')))
      (OR (not ('A' 1) ends (Var 'X')))
      (OR (not ('B' 0) ends (Var 'Y')))
      (OR (not ('A' 1) ends (Var 'Z')))
      (OR (not ('A' 1) starts (Var 'X')))
      (OR (not ('A' 1) starts (Var 'Y')))
      (OR (not ('A' 1) starts (Var 'Z')))
    )
    (
      (('A' 1) is (('A' 0) (const 1)) (('B' 0) (const 1)))
      (('B' 1) is (('A' 1) (i1 1) (i2 1) (const 0)))
      (('C' 1) is (('A' 1) (i3 1) (const 0)))
      (('D' 1) is (('A' 1) (i4 1) (const 1)))
      (('E' 1) is (('A' 1) (i5 1) (const 0)))
    )
  )
)
```

10 Листинги

В силу большого объёма кода программы (около 2500 строк) в раздел включена лишь часть реализации основных четырёх функций. Тем не менее, формат и назначение невключённых, кажется, угадывается по названию и контексту использования.

10.1 Функция `Pick`

```
/*
  <Pick s.NUMBER e.Eqs>
    == Success
    == NotMinimal
    == t.Eq
*/
Pick {
  s.Number t.Eq e.RestEqs
    , s.Number : {
      1, t.Eq : ((AreEqual (e.LHS) (e.RHS)) (e.Constrs) (e.Conds))
        , <MapCompose
            ((GetVars (e.LHS)) (GetVars (e.RHS))) (/* empty */)
          > : (e.Vars)
        , <SubtractSets
            (e.Vars) (<GetNonEmptyVars e.Constrs>)
          > : (e.EmptyVars)
        , <MapCall
            Revert
            (GenSubst (/* empty */))
            <MapCall Plain Wrap e.EmptyVars>
          > : e.Substs
        , <MapCall
            Curry
            (Subst (e.Substs))
            (e.LHS) (e.RHS)
          > : {
          (e.Elems) (e.Elems) = Success;

          (e.NewLHS) (e.NewRHS)
            , <MapCompose
                ((GetConsts (e.NewLHS)) (GetConsts (e.NewRHS)))
                (/* empty */)
              > : {
              (/* no consts */) = NotMinimal;

              (e.SomeConsts)
                , <MapCompose
                    (
```

```
                        <MapCall
                          Curry
                          (Wrap CleanUpConstrs)
                          e.Substs
                        >
                      )
                      (e.Constrs)
                    > : (e.NewConstrs)
                  = (
                      (AreEqual (e.NewLHS) (e.NewRHS))
                      (e.NewConstrs) (e.Conds)
                    );
              };
          };

        s.Other = <Pick <Sub s.Number 1> e.RestEqs>;
    };
}
```

10.2 Функция SubstIndex

```
/*
  <SubstIndex s.Index (e.Exps) t.Eq> == t.NewEq
*/
SubstIndex {
  s.Index (e.Exps) ((AreEqual (e.LHS) (e.RHS)) (e.Constrs) (e.Conds))
    , <SubstExpsToConds
        s.Index (e.Exps) (e.Conds) (/* empty */) (/* empty */)
      > : (e.ReplacedConds) (e.CollapsedConstsSubsts)
    , <ReplaceRepeatedConds
        (e.ReplacedConds) (/* empty */)
      > : (e.UniqueConds) (e.RepeatedConstsSubsts)
    , <MapCall
        Curry
        (Subst (e.CollapsedConstsSubsts e.RepeatedConstsSubsts))
        (e.LHS) (e.RHS)
      > : (e.ReplacedLHS) (e.ReplacedRHS)
    = <NormalizeEq
        (
          (AreEqual (e.ReplacedLHS) (e.ReplacedRHS))
          (e.Constrs) (e.UniqueConds)
        )
      >;
}

/*
  <SubstExpsToConds
```

```
27      s.Index (e.Exps) (e.Conds) (e.ProcessedConds) (e.Substs)
28    >
29      == (e.ProcessedConds) (e.Substs)
30  */
31  SubstExpsToConds {
32    s.Index (e.Exps) (e.Conds) (e.ProcessedConds) (e.Substs)
33      , e.Conds : {
34        e.L (t.Const is t.Block) e.R
35          , <SubstExpsToBlock s.Index (e.Exps) t.Block> : {
36            /* block collapsed */
37              = <SubstExpsToConds
38                  s.Index (e.Exps) (e.R) (e.ProcessedConds e.L)
39                  (e.Substs <GenSubst (t.Const) (/* empty */)>)
40                >;
41
42            t.NewBlock
43              = <SubstExpsToConds
44                  s.Index (e.Exps) (e.R)
45                  (e.ProcessedConds e.L (t.Const is t.NewBlock)) (e.Substs)
46                >;
47          };
48
49        e.Other = (e.ProcessedConds e.Conds) (e.Substs);
50      };
51  }
52
53  /*
54    <SubstExpsToBlock s.Index (e.Exps) t.Block>
55      == t.NewBlock
56      == empty
57  */
58  SubstExpsToBlock {
59    s.Index (e.Exps) (t.BlockConst e.OldExps)
60      , e.OldExps : {
61        e.L (s.Index s.Multiplier) e.R
62          , <SumUpExps
63              e.L <MapCall Revert (MulExp s.Multiplier) e.Exps> e.R
64            > : {
65            (const 0) = /* empty */;
66
67            e.ReplacedExps (const s.Number)
68              , <QuickSort IsLess-Exp e.ReplacedExps> : e.SortedExps
69              = (t.BlockConst e.SortedExps (const s.Number));
70          };
71
72        e.Other = (t.BlockConst e.OldExps);
73      };
```

```
}

/*
  <ReplaceRepeatedConds (e.Conds) (e.Substs)>
    == (e.UniqueConds) (e.Substs)
*/
ReplaceRepeatedConds {
  (e.Conds) (e.Substs)
    , e.Conds : {
      e.L (t.Const1 is t.Block) e.M (t.Const2 is t.Block) e.R
        = <ReplaceRepeatedConds
            (e.L (t.Const1 is t.Block) e.M e.R)
            (e.Substs <GenSubst (t.Const2) (t.Const1)>)
          >;

      e.Other = (e.Conds) (e.Substs);
    };
}
```

10.3 Функция PairComp

```
/*
  <PairComp t.Const t.Const1 t.Const2 t.Eq>
    == t.NewConst (e.Eqs)
*/
PairComp {
  t.Const t.Const1 t.Const2 t.Eq
    = <GetNewConst t.Const>
      (<HandleEmptySubsts t.Const t.Const1 t.Const2 t.Eq>);
}

/*
  <HandleEmptySubst t.Const t.Const1 t.Const2 t.Eq>
    == e.Eqs
*/
HandleEmptySubsts {
  t.Const t.Const1 t.Const2 t.Eq
    , t.Eq : ((AreEqual (e.LHS) (e.RHS)) (e.Constrs) (e.Conds))
    , <MapCallTill
        (/* empty */)
        Curry
        (GetEmptySubst t.Const1 t.Const2 (e.Constrs))
        (e.LHS) (e.RHS)
      > : {
      t.Subst
        = <HandleEmptySubsts
            t.Const t.Const1 t.Const2
```

```
27              (
28                (
29                  AreEqual
30                  <MapCall Curry (Subst (t.Subst)) (e.LHS) (e.RHS)>
31                )
32                <CleanUpConstrs t.Subst (e.Constrs)> (e.Conds)
33              )
34            >
35            <HandleEmptySubsts
36              t.Const t.Const1 t.Const2
37              (
38                (AreEqual (e.LHS) (e.RHS))
39                (e.Constrs (OR <GenSubstDenial t.Subst>)) (e.Conds)
40              )
41            >;
42
43        /* empty */
44          = <HandleNonEmptySubsts t.Const t.Const1 t.Const2 t.Eq>;
45    };
46  }
47
48  /*
49    <HandleNonEmptySubsts t.Const t.Const1 t.Const2 t.Eq>
50      == e.Eqs
51  */
52  HandleNonEmptySubsts {
53    t.Const t.Const1 t.Const2 t.Eq
54      , t.Eq : ((AreEqual (e.LHS) (e.RHS)) (e.Constrs) (e.Conds))
55      , <MapCompose
56          (
57            <MapCall
58              Curry
59              (Wrap
60                GetNonEmptySubsts t.Const1 t.Const2 (e.Constrs) (e.Conds)
61              )
62              (e.LHS) (e.RHS)
63            >
64          )
65          (/* empty */) (/* empty */)
66        > : (e.ElementarySubsts) (e.CompositeSubsts)
67      , <CartesianProductOfOptionSets
68          <GenOptionSets (e.ElementarySubsts) (e.CompositeSubsts)>
69        > : (e.MultipliedOptions)
70      , <MapCall
71          Curry
72          (NormalizePairOption (e.Conds) (e.Constrs))
73          e.MultipliedOptions
```

```
      > : {
      /* empty */
        = <ApplyPairOption
            t.Const t.Const1 t.Const2 t.Eq ((/* empty */) (e.Constrs))
          >;

      e.NormalizedOptions
        = <MapCall
            Curry
            (ApplyPairOption t.Const t.Const1 t.Const2 t.Eq)
            e.NormalizedOptions
          >;
    };
}

/*
  <ApplyPairOption t.Const t.Const1 t.Const2 t.Eq t.PairOption>
    == t.NewEq
*/
ApplyPairOption {
  t.Const t.Const1 t.Const2 t.Eq ((e.Substs) (e.Constrs))
    , t.Eq : ((AreEqual (e.LHS) (e.RHS)) (e.IrrelevantConstrs) (e.Conds))
    , (t.Const is (t.Const1 (const 1)) (t.Const2 (const 1))) : t.NewCond
    , <MapCall
        Curry
        (Subst (e.Substs <GenSubst (t.Const1 t.Const2) (t.Const)>))
        (e.LHS) (e.RHS)
      > : (e.ReplLHS) (e.ReplRHS)
    = <NormalizeEq
        (
          (AreEqual (e.ReplLHS) (e.ReplRHS))
          (e.Constrs) (e.Conds t.NewCond)
        )
      >;
}
```

10.4 Функция BlockComp

```
/*
  <BlockComp t.BlockConst t.Const s.Index t.Eq>
    == (t.NewConst t.NewIndex t.NewEq)+
*/
BlockComp {
  t.BlockConst t.Const s.Index t.Eq
    , t.Eq : ((AreEqual (e.LHS) (e.RHS)) (e.Constrs) (e.Conds))
    , <MapCompose
        ((GetVars (e.LHS)) (GetVars (e.RHS))) (/* empty */ )
```

```
      > : (e.EqVars)
    , <MapCall
        Curry
        (GenBlockOptionSets (e.Constrs) (e.Conds) t.BlockConst)
        e.EqVars
      > : e.OptionSets
    , <CartesianProductOfOptionSets e.OptionSets> : (e.MultipliedOptions)
    , <MapCall
        Curry
        (NormalizeBlockOption (e.Conds) (e.Constrs))
        e.MultipliedOptions
      > : e.NormOptions
    = <MapCall
        Curry
        (ApplyBlockOption t.BlockConst t.Const s.Index t.Eq)
        e.NormOptions
      >;
}

/*
  <ApplyBlockOption t.BlockConst t.Const s.Index t.Eq t.Option>
    == (t.NewConst t.NewIndex t.NewEq)
*/
ApplyBlockOption {
  t.BlockConst t.Const s.Index t.Eq ((e.Substs) (e.Constrs))
    , t.Eq : ((AreEqual (e.LHS) (e.RHS)) (e.IrrelevantConstrs) (e.Conds))
    , <MapCall
        Curry
        (Subst (e.Substs))
        (e.LHS) (e.RHS)
      > : (e.ReplLHS) (e.ReplRHS)
    , <MapCall
        Curry
        (JoinBlocks t.BlockConst)
        (e.ReplLHS) (e.ReplRHS)
      > : (e.JoinedLHS) (e.JoinedRHS)
    , <MapCallCompose
        (NameBlocks t.BlockConst (/* empty */))
        (t.Const s.Index (e.Conds))
        (e.JoinedLHS) (e.JoinedRHS)
      > : (e.NewLHS) (e.NewRHS) t.NewConst s.NewIndex (e.NewConds)
    = (
        t.NewConst s.NewIndex
        <NormalizeEq
          ((AreEqual (e.NewLHS) (e.NewRHS)) (e.Constrs) (e.NewConds))
        >
      );
```

```
}

/*
  <JoinBlocks t.BlockConst (e.Elems)> == (e.NewElems)
*/
JoinBlocks {
  t.BlockConst (e.Elems)
    , e.Elems : {
      e.L t.BlockConst (t.BlockConst e.Indices (const s.Num)) e.R
        = <JoinBlocks
            t.BlockConst
            (e.L (t.BlockConst e.Indices (const <Add s.Num 1>)) e.R)
          >;

      e.L (t.BlockConst e.Indices (const s.Num)) t.BlockConst e.R
        = <JoinBlocks
            t.BlockConst
            (e.L (t.BlockConst e.Indices (const <Add s.Num 1>)) e.R)
          >;

      e.L (t.BlockConst e.Indices1 (const s.Num1))
      (t.BlockConst e.Indices2 (const s.Num2)) e.R
        = <JoinBlocks
            t.BlockConst
            (
              e.L
              (
                t.BlockConst e.Indices1 e.Indices2
                (const <Add s.Num1 s.Num2>)
              )
              e.R
            )
          >;

      e.L t.BlockConst t.BlockConst e.R
        = <JoinBlocks t.BlockConst (e.L (t.BlockConst (const 2)) e.R)>;

      e.Other = (e.Elems);
    };
}

/*
  <NameBlocks
    t.BlockConst (e.ProcessedElems) t.Const s.Index (e.Conds) (e.Elems)
  >
    == t.NewConst t.NewIndex (e.NewConds) t.NewElem
*/
```

```
NameBlocks {
  t.BlockConst (e.ProcessedElems) t.Const s.Index (e.Conds) (e.Elems)
    , e.Elems : {
      e.L (t.BlockConst e.Indices t.Summand) e.R
        , <NameIndices
            s.Index (e.Indices) (/* empty */)
          > : s.NewIndex (e.NamedIndices)
        , <MapCall Revert (FormPair 1) e.NamedIndices> : e.Exps
        = <NameBlocks
            t.BlockConst (e.ProcessedElems e.L t.Const)
            <GetNewConst t.Const> s.NewIndex
            (e.Conds (t.Const is (t.BlockConst e.Exps t.Summand)))
            (e.R)
          >;

      e.Other = t.Const s.Index (e.Conds) (e.ProcessedElems e.Elems);
    };
}

/*
  <NameIndices s.Index (e.NamelessIndices) (e.NamedIndices)>
    == s.NewIndex (e.NamedIndices)
*/
NameIndices {
  s.Index (e.NamelessIndices) (e.NamedIndices)
    , e.NamelessIndices : {
      Index e.RestIndices
        = <NameIndices
            <GetNewIndex s.Index> (e.RestIndices)
            (e.NamedIndices s.Index)
          >;

      /* empty */ = s.Index (e.NamedIndices);
    };
}
```

11 Заключение

Настоящая практика была самым интересным и увлекательным учебным опытом, который когда-либо у меня был. Во многом это обусловлено простором для исследования и творческой свободой. Не менее мотивировала возможность познакомиться с задачами современной науки и внести свой небольшой вклад в их решение. Я особенно благодарен моему научному руководителю — А.Н. Непейводе — за энтузиазм и самоотдачу, с которыми она подходила к делу. Очень вдохновляет.

Поставленные цели практики во многом были достигнуты. Изучение руководства Ф.В. Турчина [2], лекций А.П. Немытых [3] и упоминавшихся рекомендаций А.В. Коновалова [4], [5] помогло освоиться с Рефалом-5 и его методами программирования. Приложение алгоритма Ежа также было формализовано — мне кажется, удачно. Но программной части необходима доработка: сейчас многие фрагменты кода нуждаются в рефакторинге, функции на некоторых входных данных возвращают некорректный результат. Тем не менее, многие классы уравнений обрабатываются корректно (как демонстрируют unit-тесты).

Есть желание продолжать исследовать данный вопрос. Тем более, представленная сфера очень актуальна для современной информатики.

Список литературы

[1] Artur Jeż. *Recompression: a simple and powerful technique for word equations* (https://arxiv.org/abs/1203.3705).

[2] Ф.В. Турчин. *Рефал-5 — руководство по программированию и справочник* (http://refal.ru/rf5_frm.htm).

[3] А.П. Немытых. *Лекции по языку программирования Рефал* (http://www.botik.ru/PSI/RCMS/publications/publ-texts-2014/refal-17.pdf).

[4] А.В. Коновалов. *Форматы функций и типы данных в Рефале-5λ* (https://bmstu-iu9.github.io/refal-5-lambda/A-formats-and-types.html).

[5] А.В. Коновалов. *Рекомендации по стилю оформления кода* (https://github.com/bmstu-iu9/refal-5-lambda/blob/master/doc/style-guide.md).