Санкт-Петербургский государственный университет

Кафедра системного программирования

Мухин Артем Михайлович

Синтез структур данных

Курсовая работа

Научный руководитель:
ст. преп. Кириленко Я. А.

Консультант:
программист JetBrains Мордвинов Д. А.

Санкт-Петербург
2018

Оглавление

Введение

Во многих программах используются специальные структуры данных, не содержащиеся явно в стандартных библиотеках языков программирования. Таковыми являются структуры, составленные из известных коллекций с помощью композиции или изменения некоторых свойств. Реализация таких структур — монотонная и сложная работа, в ходе которой зачастую возникают ошибки как на этапе подбора оптимальных структур (т.к для этого программисту необходимо знать асимптотики различных операций над структурами данных), так и на этапе реализации.

Существуют инструменты, позволяющие автоматически подбирать оптимальные реализации структур данных [2, 4, 9]. Часть из них использует синтез программного кода. Синтез программ — это задача автоматического поиска программы, написанной на заданном языке программирования и удовлетворяющей некоторым ограничениям, которые задает пользователь. Ограничения на требуемую программу могут быть спецификацией на некотором промежуточном языке, примерами ввода-вывода, описанием на естественном языке, частично написанными программами и так далее. Синтезатор структур данных — это инструмент, позволяющий получить реализацию (на заданном пользователем языке программирования) автоматически сконструированной структуры данных. Спецификации структур данных часто представляют собой описания на некотором декларативном языке, поскольку такой вид спецификации достаточно выразителен и удобен для пользователя.

В последние годы было предпринято несколько попыток создать синтезаторы структур данных [9]. Однако класс задач, решаемый ими, ограничен. Усовершенствование текущих подходов к синтезу структур данных является актуальной проблемой.

1. Постановка задачи

Целью данной работы является создание синтезатора структур данных, подходящего для более широкого класса задач, чем существующие. Для достижения этой цели были поставлены следующие задачи:

- исследовать существующие подходы к синтезу программ;
- спроектировать декларативный язык спецификации структур данных, достаточно мощный для описания нетривиальных задач;
- разработать алгоритм синтеза структур данных;
- разработать прототип ядра синтезатора.

2. Обзор

Синтез программ является одной из наиболее значимых проблем информатики и искусственного интеллекта. Успешное решение этой задачи требует преодоления двух основных проблем:

1. Необходимо предоставить пользователю достаточно простой и удобный способ спецификации желаемой программы, который не был бы слишком сложен для компьютера.

2. Синтез программ — сложнейшая комбинаторная проблема; во избежание взрыва количества программ приходится как можно сильнее сокращать пространство перебора.

Исторически реализация синтезаторов программ стала возможной после появления инструментов для автоматического доказательства теорем.

Рассмотрим некоторые подходы к синтезу программ.

2.1. Дедуктивный синтез

Первые подходы [5, 19] к синтезу программ были основаны на дедукции. Идея *дедуктивного синтеза* — автоматически доказать заданную пользователем спецификацию и затем использовать это доказательство для конструирования подходящей программы. Основной минус этого подхода заключается в том, что пользователю приходится составлять полную и формальную спецификацию, что во многих случаях не проще написания требуемой программы вручную.

2.2. Индуктивный синтез

Позже был придуман метод индуктивного синтеза, который позволяет вместо полных спецификаций программы использовать индуктивные ограничения, такие как примеры ввода-вывода и демонстрации работы программы. В 1970-е годы этот подход был реализован в нескольких инструментах для синтеза небольших программ на языке LISP [14].

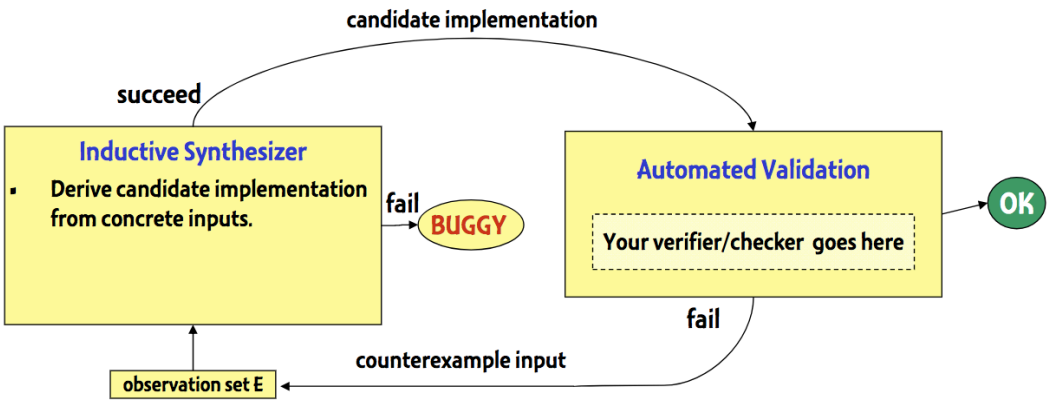
Сегодня для синтеза программ чаще применяют техники, сочетающие в себе и дедуктивный, и индуктивный подходы [12].

2.3. Синтез, управляемый оракулом

Одним из современных подходов к синтезу программ является синтез на основе *частично написанных программ* (программ с «дырками»). В качестве спецификации желаемой программы выступает некоторый каркас, содержащий «заглушки», реализацию которых должен найти синтезатор, а также ограничения в виде *утверждений* (assertions), которым должна удовлетворять итоговая программа. Первой известной реализацией этого подхода является инструмент Sketch [15], созданный в 2008 году.

Синтез программ, реализованный в Sketch, основывается на верификации программ-кандидатов, полученных генератором произвольных программ на фиксированном языке. Для каждого очередного кандидата требуется найти контрпример, который в дальнейшем используется как дополнительное ограничение на искомую программу. Такой подход получил название CEGIS (counterexample-guided inductive synthesis). Позже этот подход был обобщён до OGIS [6] (oracle-guided inductive synthesis). В OGIS синтезатор сначала ищет программу-кандидата с использованием упрощенной спецификации, а затем оракул проверяет правильность кандидата (например, в CEGIS оракул проверяет соответствие кандидата логической формуле).

Рис. 1: Архитектура CEGIS (изображение взято из [15])



2.4. Инструмент λ^2

Также стоит упомянуть инструмент для синтеза преобразований списков λ^2 [4]. Этот синтезатор принимает на вход несколько примеров преобразований списка или дерева и возвращает функцию, удовлетворяющую этим примерам. λ^2 обладает достаточной мощностью, чтобы синтезировать весьма нетривиальные операции, такие как декартово произведение списков и удаление дубликатов из списка. Данная работа интересна тем, что сочетает сразу несколько техник синтеза: индукцию для обобщения входных примеров; дедукцию для опровержения и генерации новых примеров; заполнение «дырок» в промежуточных программах-кандидатах.

2.5. Подход Syntax-Guided Synthesis

Было замечено, что многие задачи синтеза программ можно существенно упростить, если использовать спецификации двух типов: синтаксическую и семантическую. Синтаксическая спецификация представляет собой грамматику искомой программы, а семантическая спецификация — ограничения в терминах логики первого порядка. Подход SyGuS [18] реализует эту идею. Каждый год проводятся соревнования SyGuS-решателей [11, 16, 17].

2.6. Синтез структур данных

Синтез структур данных подразумевает собой автоматическую реализацию структуры данных, сконструированной согласно пользовательской спецификации. При этом синтезатор осуществляет перебор и выбор подходящего (и, в некоторых случаях, оптимального по асимптотическим или другим свойствам) варианта среди зафиксированного пространства структур данных.

В 1970-е годы задача выбора оптимальной реализации структуры данных была рассмотрена во время проектирования языка SETL [13]. Язык SETL предоставляет два основных агрегатных типа данных: неупо-

рядоченные множества и кортежи. Элементы множеств и кортежей могут быть любого произвольного типа (включая агрегатные типы). Также в этом языке есть словари, которые представляются в виде множества пар (т.е. кортежей из двух элементов). При этом неупорядоченное множество и словарь — это интерфейсы, у которых существует множество реализаций (на основе хэш-таблиц и различных деревьев). Для улучшения производительности программ, написанных на SETL, требовалось выбирать наиболее оптимальные для конкретных задач реализации этих интерфейсов во время исполнения программы.

Кроме того, были попытки синтезировать структуры данных с помощью реляционной алгебры. Так, авторы RelC [2] использовали реляционные спецификации для синтеза структур данных на основе реализаций из библиотек STL (C++ Standard Template Library) и Boost. RelC был протестирован на задачах над графами из некоторых проектов с открытым исходным кодом и показал прирост производительности по сравнению с оригинальными реализациями.

Авторы синтезатора структур данных Cozy [9] предложили подход к синтезу структур данных, основанный на CEGIS. С помощью Cozy работа программиста сводится к описанию требуемой ему структуры данных на языке спецификации высокого уровня, а поиск и реализацию подходящих структур данных берет на себя синтезатор. Cozy успешно решает некоторый узкий круг задач, а именно — фильтрацию элементов коллекции по предикатам, содержащим только имена переменных, полей и знаки сравнения. Рассмотрим язык спецификации Cozy подробнее. Его грамматика выглядит следующим образом:

⟨*predicate*⟩ ::= True | False
| ⟨*var*⟩ ⟨*comparison*⟩ ⟨*var*⟩
| ⟨*predicate*⟩ And ⟨*predicate*⟩
| ⟨*predicate*⟩ Or ⟨*predicate*⟩
| Not ⟨*predicate*⟩

⟨*var*⟩ ::= ⟨*field*⟩ | ⟨*query-var*⟩

⟨*comparison*⟩ ::= ‘==’ | ‘>’ | ‘>=’ | ‘<’ | ‘<=’ | ‘!=’

Этот язык имеет несколько проблем. Во-первых, в спецификациях возможны только запросы-фильтрации (поиск элементов коллекции, удовлетворяющих предикату), поэтому на этом языке нельзя описать выполнение каких-либо действий над элементами. Во-вторых, язык Cozy не поддерживает арифметику. Оракулы, используемые в синтезаторах, обычно основаны на SMT-решателях (программах, решающих задачу выполнимости формулы в теориях). Учитывая мощности современных SMT-решателей, добавление линейной арифметики выглядит возможным.

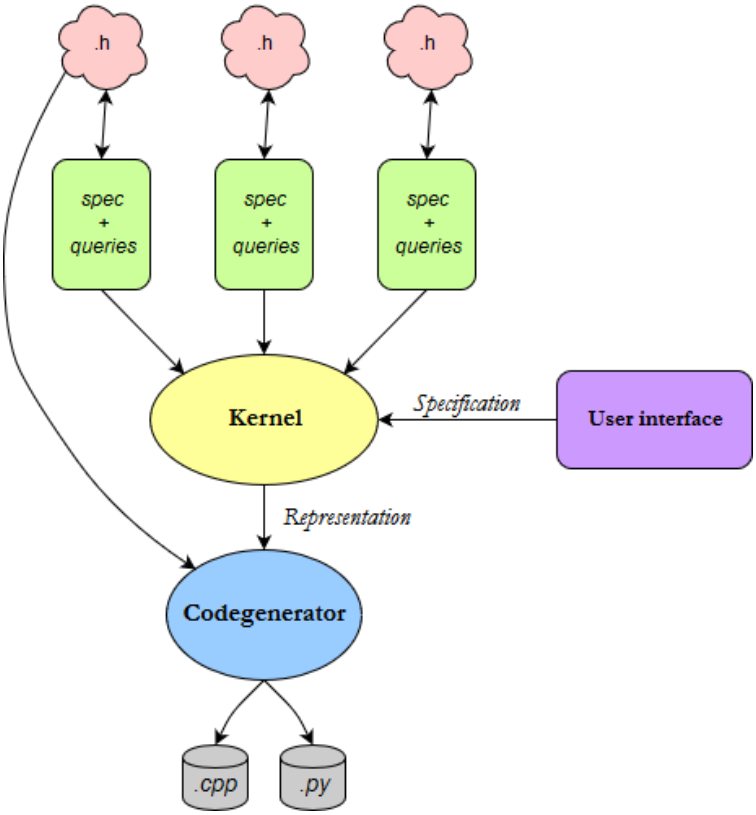
3. Решение

В данной главе описывается процесс проектирования и разработки прототипа ядра — части синтезатора, ответственной за поиск искомой структуры данных по пользовательской спецификации.

3.1. Архитектура синтезатора

Данная курсовая работа посвящена созданию ядра (*kernel*) синтезатора, а создание генератора кода и реализация структур данных остается за её рамками. На рисунке представлена архитектура будущего синтезатора.

Рис. 2: Архитектура синтезатора



3.2. Язык спецификации

Перед началом проектирования языка спецификации структур данных, была собрана некоторая коллекция задач, потенциально подходя-

щих для синтеза. Были выбраны несколько алгоритмически простых задач на нетривиальные структуры данных: дерево отрезков, декартово дерево, корневая декомпозиция отрезка, а также на более известные коллекции: хэш-таблица, приоритетная очередь, дерево поиска, ассоциативные контейнеры.

На основе этих задач необходимо было понять, как могли бы выглядеть спецификации (т.е. формальные описания задач) и, соответственно, какие возможности должен поддерживать язык спецификаций. Было замечено следующее:

- в задачах обычно требуются коллекции, элементами которых в общем случае бывают записи из нескольких элементов примитивных типов;
- действия над структурами данных удобно представлять в виде *запросов* — функций, которые принимают примитивные входные параметры и возвращают какие-то элементы структуры данных (например, все элементы, удовлетворяющие какому-то свойству) или примитивные значения (например, количество элементов);
- сами запросы удобно записывать в функциональном стиле с применением комбинаторов $filter$, map и $fold$.

Упрощенный вид грамматики спроектированного языка приведен ниже:

$$
\begin{aligned}
\mathit{Specification} &::= \mathit{RecordDecl}^* \;\; \mathit{CollectionDecl}^* \;\; \mathit{Query}^* \\
\mathit{RecordDecl} &::= \texttt{record} \;\; \mathit{Name} \;\; \texttt{\{} \; \mathit{Field}^* \; \texttt{\}} \\
\mathit{CollectionDecl} &::= \mathit{Name} \; \texttt{:} \; \texttt{Collection<}\mathit{Name}\texttt{>} \\
\mathit{Field} &::= \mathit{Name} \; \texttt{:} \; \mathit{Type} \\
\mathit{Query} &::= \texttt{query} \;\; \mathit{Name}\texttt{(} \; \mathit{Parameter}^* \; \texttt{)} \; \texttt{=} \; \mathit{Body} \\
\mathit{Body} &::= \mathit{Name} \mid \mathit{Func} \; \mathit{Body} \\
\mathit{Func} &::= \texttt{filter} \; f \mid \texttt{map} \; f \\
\mathit{Parameter} &::= \mathit{Name} : \mathit{Type} \\
\mathit{Type} &::= \texttt{Int} \mid \texttt{Float} \mid \texttt{String} \mid \texttt{Bool}
\end{aligned}
$$

Рис. 3: Грамматика языка спецификации

Спецификация состоит из объявлений типов записей (элементов коллекций) и типов коллекций, а также из определений запросов к коллекциям. Синтаксис определения запросов во многом подобен синтаксису языка Haskell. Пользователь может использовать комбинаторы *filter* и *map*, передавая в них функции, содержащие арифметические операторы, сравнение, использование значений полей записей и параметров запроса. Этим функциям соответствует нетерминал f.

Для демонстрации возможностей языка рассмотрим задачу поиска инцидентных данной вершине дуг в графе, спецификация которой приведена в Листинге 1.

Listing 1: Поиск инцидентных данной вершине дуг в графе

```
record Edge { src: Int, dst: Int }
graph: Collection<Edge>
query findEdges(node: Int) =
    filter (λedge → edge.src == node || edge.dst == node) graph
```

В этой спецификации вводится некоторая коллекция *graph*, состоящая из записей, соответствующих ребрам графа. Далее определяется запрос, результатом выполнения которого является набор рёбер, инцидентных данной вершине.

3.3. Поддержка структур данных

Синтезатор выполняет перебор различных структур данных, чтобы найти среди них нужную. Для ограничения пространства перебора необходимо зафиксировать множество структур данных, которые он потенциально будет поддерживать. Таковыми являются: список, хэш-таблица и другие стандартные коллекции, а также некоторые нестандартные: дерево отрезков [1], дерево Фенвика [3], корневая декомпозиция [7] и некоторые другие. Все эти структуры данных удобно описывать как совокупности функций над этими коллекциями. Типизация этих функций позволяет отсечь заведомо некорректные по типам программы. Пример такой типизации показан ниже (для структур данных словарь и дерево Фенвика).

```
* Map<K, V>
Map.build         :: (t → (K, V)) → [t] → Map<K, V>
Map.get           :: K → Map<K, V> → V
Map.add           :: K → V → Map<K, V> → Map<K, V>
Map.size          :: Map<K, V> → Int

* FenwickTree<T>
FenwickTree.build :: (T → T') → Array<T> → FenwickTree<T>
FenwickTree.eval  :: Int → Int → T'
FenwickTree.get   :: Int → T
```

Все коллекции в некотором приближении можно представлять как списки, а операции над коллекциями — как преобразования списков. Например, хэш-таблицу можно представлять как список пар $(key, value)$. Преобразования списков выражаются через комбинаторы $filter$, map и $fold$. Таким образом можно ввести «отношение эквивалентности» между функциями над коллекциями и преобразованиями списков. Примеры такой эквивалентности приведены ниже:

```
Map.build f ks ~ map (λk → (k, f k)) ks
Map.get key xs ~ map snd (filter (λ(k, v) → k = key) xs)
```

В данной работе рассматриваются только функции, выражаемые без использования $fold$. Поэтому многие нетривиальные операции над коллекциями остаются за рамками работы.

Композиции $filter$ и map можно выразить через $mapFilter$ по сле-

дующим правилам:

```
map f (filter p xs)           =  mapFilter p f xs
filter p (map f xs)           =  mapFilter (p ∘ f) f xs
filter p' (mapFilter p f xs)  =  mapFilter (λx → p x ∧ p' x) f xs
map f' (mapFilter p f xs)     =  mapFilter p (f' ∘ f) xs
mapFilter p f (filter p' xs)  =  mapFilter (λx → p x ∧ p' x) f xs
mapFilter p f (map f' xs)     =  mapFilter (p ∘ f') (f ∘ f') xs
```

Таким образом вводится редукция цепочки преобразований к одному *mapFilter*, которая упрощает будущую работу по верификации кандидатов.

Для наглядности рассмотрим пример редукции. Пусть синтезатор подобрал приведенное ниже решение задачи поиска ребер, выходящих из данной вершины (решением является определение запроса, в котором могут использоваться локальные связывания):

```
findEdges n = Map.get n table where
    table = Map.build (λe → (e.src, filter (λe' → e'.src = e.src) graph)) graph
```

Здесь вводится коллекция *table* типа `Map<Int, List<Edge>>` (тип выводится по Хиндли-Милнеру [10]) и определяется запрос к этой коллекции. Это решение можно редуцировать к *mapFilter* следующим образом:

```
findEdges n
= Map.get n table
~ Map.get n (Map.build (λe → (e.src, filter (λe' → e'.src = e.src) graph)) graph)
~ map snd (filter (λ(src, _) → src = n) (map (λe →
    (e.src, filter (λe' → e'.src = e.src) graph)) graph))
~ mapFilter (λe → e.src = n) (mapFilter (λe' → e'.src = e.src) id graph) graph
```

В данной цепочке преобразования происходят в следующем порядке:

1. В исходное определение подставляется локальное связывание (*table*).
2. Операции над коллекциями (*Map.get*, *Map.build*) приводятся в эквивалентные преобразования списков.
3. Преобразования списков редуцируются к *mapFilter*.

3.4. Перебор и верификация кандидатов

После получения спецификации и обработки синтезатор переходит к перебору и проверке возможных решений (*кандидатов*). Для пере-

бора (поиска) программ-кандидатов многие синтезаторы используют собственные генераторы (например, Cozy [9]). Генераторы используют различные техники поиска: поиск в глубину, поиск в ширину, поиск с чередованием (interleaving) и так далее. Однако вместо того чтобы решать задачу с нуля, было решено свести её к задаче SyGuS. Для этого был предложен способ построения спецификаций обоих типов. В качестве синтаксической спецификации строится грамматика, соответствующая функциям над внутренними коллекциями и их эквивалентной записи. Ниже показан небольшой фрагмент такой грамматики:

```
List<Edge>      ::=  graph | Map<K, List<Edge>>.get
Map<K,V>.get    ::=  map snd (filter(λ(k,v) → k=Obj<K>) Map<K,V>)
Map<K,V>        ::=  Map<K,V>.build
Map<K,V>.build ::=  map (λk → (k, Func<K,V> k)) List<K>
```

Нетерминалы в этой грамматике содержат типовые параметры. При этом SyGuS не поддерживает грамматику, снабженную типами, поэтому на следующем этапе данная грамматика специализируется и редуцируется. Допустим, искомый запрос возвращает значения типа List<Edge> (например, в задаче поиска инцидентных вершин). Тогда редукция происходит таким образом:

```
List<Edge> ∼
map snd (filter (λ(k,v) → k = Obj<K>) Map<K, List<Edge>>) ∼
map snd (filter (λ(k,v) → k = Obj<K>) (map (λk → (k, Func<K,List<Edge>> k)) List<K>)) ∼
mapFilter (λ(k,v) → k = Obj<K>) snd (map (λk → (k, Func<K,List<Edge>>)) List<K>) ∼
mapFilter (λ(k,v) → k = Obj<K>) Func<K,List<Edge>> List<K>
```

Для построения логических ограничений используется та же грамматика. Благодаря *mapFilter*-редукции в итоге всё сводится к проверке эквивалентности двух *mapFilter*: из спецификации и из программы-кандидата:

```
mapFilter (λe → e.src = n) id graph /* specification */
mapFilter (λ(k,v) → k = Obj<Int>) Func<Int,List<Edge>> graph /* candidate */
```

Эквивалетность двух *mapFilter* проверяется исходя из свойства

`mapFilter p f xs` $\sim$ `mapFilter p' f' xs` $\iff \forall x : p(x) \leftrightarrow p'(x) \wedge p(x) \rightarrow (f(x) = f'(x))$

В конце SyGuS приходит к решению в терминах mapFilter:

```
mapFilter (λe → e.src = n) (λe → filter (λe' → e'.src = e.src) graph) graph
```

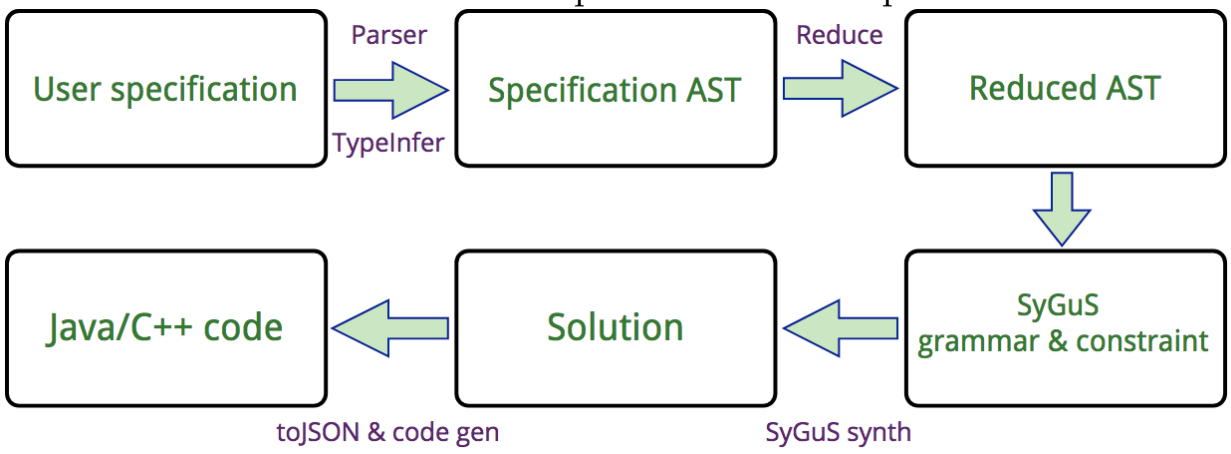
которое эквивалентно решению с использованием конкретных коллекций:

```
Map.get n (Map.build (λe → (e.src, filter (λe' → e'.src = e.src) graph) ) graph)
```

3.5. Реализация

Для реализации прототипа ядра синтезатора был выбран язык Haskell. Этот язык, благодаря мощной системе типов и механизму сопоставления с образцом (pattern-matching), отлично подходит для работы с синтаксическими деревьями и прототипированием языковых инструментов. Парсер языка спецификаций написан с помощью библиотеки монадических парсер-комбинаторов Parsec [8]. Помимо парсера, для языка спецификации были реализованы: pretty-printer, редукция к $mapFilter$, вывод типов (по Хиндли-Милнеру). Также был написан конвертер программ-решений в формат JSON для последующей обработки генератором кода.

Рис. 4: Схема работы синтезатора



На рисунке показана схема работы всего синтезатора. Сначала пользователь задает спецификацию требуемой структуры данных. По спецификации строится её дерево разбора, которое затем редуцируется к $mapFilter$. Далее по полученному представлению спецификации строится грамматика и логические ограничения для SyGuS. Полученные ограничения передаются SyGuS-решателю, который возвращает искомую структуру данных в некотором внутреннем представлении. Это

представление конвертируется в формат JSON и передается генератору кода. В результате пользователь получает реализацию требуемой структуры данных на заданном им языке программирования.

4. Заключение

В ходе данной работы получены следующие результаты:

1. Исследована активно развивающаяся предметная область — синтез программ.
2. Спроектирован язык спецификации структур данных.
3. Разработан алгоритм синтеза.
4. Разработан прототип ядра синтезатора.

Исходный код прототипа ядра доступен на GitHub [1].

Будущие исследования

В будущем планируется добавить поддержку комбинатора $fold$, так как без него невозможно выразить многие нетривиальные операции над коллекциями (например, сортировку). Добавление $fold$ позволит синтезатору использовать больше структур данных и операций над ними.

Также, поскольку данная работа скорее теоретическая и реализован лишь прототип ядра синтезатора, необходимо соединить воедино ядро с генератором кода и сравнить полученный синтезатор с существующими.

[1] `https://github.com/ortem/CollectionSynthesis`

Список литературы

[1] Bentley J. L. Solutions to Klee's rectangle problems // Unpublished manuscript, Dept of Comp Sci, Carnegie-Mellon University, Pittsburgh PA. — 1977.

[2] Data Representation Synthesis / Peter Hawkins, Alex Aiken, Kathleen Fisher et al. // Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '11. — New York, NY, USA : ACM, 2011. — P. 38–49. — URL: http://doi.acm.org/10.1145/1993498.1993504.

[3] Fenwick Peter M. A New Data Structure for Cumulative Frequency Tables // Softw. Pract. Exper. — 1994. — . — Vol. 24, no. 3. — P. 327–336. — URL: http://dx.doi.org/10.1002/spe.4380240306.

[4] Feser John K., Chaudhuri Swarat, Dillig Isil. Synthesizing Data Structure Transformations from Input-output // SIGPLAN Not. — 2015. — Vol. 50, no. 6. — P. 229–239. — URL: http://doi.acm.org/10.1145/2813885.2737977.

[5] Green C. Cordell. Application of theorem proving to problem solving // IJCAI. — 1969. — P. 219–240.

[6] Jha Susmit, Seshia Sanjit A. A Theory of Formal Synthesis via Inductive Learning // Acta Inf. — 2017. — . — Vol. 54, no. 7. — P. 693–726. — URL: https://doi.org/10.1007/s00236-017-0294-5.

[7] Kent Carmel, M Landau Gad, Ziv-Ukelson Michal. On the Complexity of Sparse Exon Assembly. — 2006. — 07. — Vol. 13. — P. 1013–27.

[8] Parsec, a fast combinator parser : Rep. : 35 / Department of Computer Science, University of Utrecht (RUU) ; Executor: Daan Leijen : 2001. — October.

[9] Loncaric Calvin, Torlak Emina, Ernst Michael D. Fast synthesis of fast collections // PLDI 2016: Proceedings of the ACM SIGPLAN 2016

Conference on Programming Language Design and Implementation. — 2016. — P. 355–368.

[10] Milner Robin. A Theory of Type Polymorphism in Programming // J. Comput. Syst. Sci. — 1978. — Vol. 17, no. 3. — P. 348–375. — URL: https://doi.org/10.1016/0022-0000(78)90014-4.

[11] Results and Analysis of SyGuS-Comp'15 / Rajeev Alur, Dana Fisman, Rishabh Singh, Armando Solar-Lezama // Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015. — 2015. — P. 3–26. — URL: https://doi.org/10.4204/EPTCS.202.3.

[12] S. Gulwani O. Polozov, Singh R. Program Synthesis // Foundations and Trends in Programming Languages. — 2017. — . — Vol. 4, no. 1-2. — P. 1–119.

[13] Schwartz J. T. Automatic Data Structure Choice in a Language of Very High Level // Proceedings of the 2Nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — POPL '75. — New York, NY, USA : ACM, 1975. — P. 36–40. — URL: http://doi.acm.org/10.1145/512976.512981.

[14] Shaw David E., Swartout William R., Green C. Cordell. Inferring LISP Programs from Examples // Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1. — IJCAI'75. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1975. — P. 260–267. — URL: http://dl.acm.org/citation.cfm?id=1624626.1624666.

[15] Solar-Lezama Armando. Program Synthesis by Sketching : Ph. D. thesis / Armando Solar-Lezama. — Berkeley, CA, USA : University of California at Berkeley, 2008. — AAI3353225.

[16] SyGuS-Comp 2016: Results and Analysis / Rajeev Alur, Dana Fisman, Rishabh Singh, Armando Solar-Lezama // Proceedings Fifth

Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016. — 2016. — P. 178–202. — URL: https://doi.org/10.4204/EPTCS.229.13.

[17] SyGuS-Comp 2017: Results and Analysis / Rajeev Alur, Dana Fisman, Rishabh Singh, Armando Solar-Lezama // Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017. — 2017. — P. 97–115. — URL: https://doi.org/10.4204/EPTCS.260.9.

[18] Syntax-guided synthesis / R. Alur, R. Bodik, G. Juniwal et al. // 2013 Formal Methods in Computer-Aided Design. — 2013. — Oct. — P. 1–8.

[19] Zohar Manna Richard J. Waldinger. Toward automatic program synthesis // Commun. ACM. — 1971. — P. 151–165.