Архитектура упрощённой системы автоматического вывода на основе подхода А. С. Подколзина

А. П. Анненков*

Решение формализуемых математических задач с помощью ЭВМ остаётся актуальной проблемой, для которой существует несколько принципиально различных подходов. Значительный вклад в развитие данного направления внесла монография А. С. Подколзина, в которой предложена оригинальная архитектура решателя задач, основанная на логическом выводе с помощью правил преобразования термов. В данной статье описывается архитектура упрощённой системы, реализующей ключевые идеи этого подхода: механизм переключения внимания через уровни, организацию базы правил и представление контекста с метаданными.

Ключевые слова: решатель математических задач, автоматический вывод, логические процессы, логический язык, логическая формализация задач.

1. Введение

Автоматизация решения формализуемых математических задач остаётся актуальной проблемой на стыке математической логики, теоретической информатики и компьютерной алгебры. Существующие подходы варьируются от систем символьных вычислений и систем автоматических доказательств теорем до алгоритмов, основанных на предобученных языковых моделях, демонстрирующих значительные успехи в обработке математических текстов. Однако создание прозрачного, расширяемого решателя, основанного на принципах формального вывода и понятной стратегии поиска решения, сохраняет свою востребованность как для теоретических исследований, так и для практических приложений.

Значительный вклад в это направление был сделан А. С. Подколзиным, предложившим оригинальную архитектуру решателя, основанную на

* *Анненков Александр Петрович* — аспирант каф. математической теории интеллектуальных систем мех.-мат. ф-та МГУ, e-mail: `just.std@yandex.ru`, ORCID: 0009-0002-9332-1569.

Annenkov Alexander Petrovich — graduate student, Lomonosov Moscow State University, Faculty of Mechanics and Mathematics, Chair of Mathematical Theory of Intellectual Systems.

логическом выводе с использованием правил преобразования термов. Его подход, изложенный в монографии «Компьютерное моделирование логических процессов», отличается строгой формализацией, механизмом управляемого переключения внимания и идеей ранжирования правил по уровням, что позволяет эффективно управлять пространством поиска решений. Не умаляя теоретической глубины и полноты предложенного решения, стоит отметить, что оригинальная система представляет собой сложный комплекс, что создаёт трудности для её изучения, модификации и интеграции с другими средствами.

В связи с этим целью данной работы является создание и описание упрощённой архитектуры, построенной на тех же ключевых принципах, но реализованной в виде компактной экспериментальной системы. В статье рассматриваются основные проектные решения: структуры данных, алгоритм основного цикла работы и механизмы управления выводом. Особое внимание уделяется созданию целостной минимально достаточной системы, способной решать базовые алгебраические задачи. Предлагаемая реализация может служить основой для учебных моделей, прототипов и дальнейших исследований в области автоматизированного логического вывода.

Структура статьи: в разделе 2 вводятся основные понятия и формализация задач; раздел 3 описывает механизм переключения внимания; раздел 4 посвящён архитектуре системы, включая представление правил, термов и контекста; раздел 5 детализирует цикл работы алгоритма; раздел 6 кратко описывает языки системы; в приложении приводится пошаговая демонстрация работы системы на примере решения линейного уравнения, а также псевдокод основных алгоритмов системы.

2. Основные понятия

Начнём описание системы с формализации некоторых понятий и описания класса решаемых задач. В основе принципа работы лежит получение новых термов с помощью применения правил к полученным ранее термам (либо условиям задачи). *Терм*, в контексте данной работы, будем понимать в классическом индуктивном определении:

1. Переменные и константы (0-арные функциональные символы) являются термами.
2. Если f — n-арный функциональный символ, а $t_1, t_2, \ldots, t_n$ — термы, то $f(t_1, t_2, \ldots, t_n)$ — также терм.

2.1. Формализация задач

Задачей в данной работе будем называть пару, состоящую из *целевой установки* и *списка условий*.

Целевая установка — определяет тип задачи и терм, который является предметом поиска или преобразования. В системе целевая установка также представляется термом.

Система поддерживает три типа целевых установок:

1. **Find**(x) — описать значение неизвестного терма x через известные термы.
2. **Prove**(s) — вывести терм s из условий задачи (доказать истинность s в случае истинности условий).
3. **Transform**(s) — преобразовать терм к требуемому виду или упростить терм.

Список условий представляет собой конечный набор термов, считающихся истинными в контексте решаемой задачи.

Пример 1. Рассмотрим задачу решения уравнения $4x + 7 = 2x + 1$. Требуется найти значение x, удовлетворяющее равенству. Её формализация:

- целевая установка: **Find**(x);
- список условий: $\{4x + 7 = 2x + 1\}$.

2.2. Правила преобразования

Правило, помимо классического вида ***шаблон*** $\Rightarrow$ ***вывод*** содержит список термов, ограничивающих условия его применения. Последние будем называть *ограничениями*. Переменные, входящие в шаблон правила, будем называть *параметрами*.

Применение правила к терму происходит по следующему алгоритму:

1. Найти подстановку параметров, которая обращает шаблон правила в подтерм обрабатываемого терма.
2. Проверить истинность ограничений после применения такой подстановки. В случае неудачи попробовать снова с другой подстановкой.
3. Заменить подтерм, соответствующий шаблону правила, на терм, получаемый применением подстановки к выводу правила.

Следует отметить, что результат такого алгоритма неоднозначен. Для одного набора значений параметров возможно несколько различных вхождений шаблона в обрабатываемый терм. В то же время возможно, что подходящий набор таких значений может быть не единственным. Таким образом, результатом применения правила к терму считается множество (возможно пустое) термов.

Пример 2. Рассмотрим правило переноса $a = b \Rightarrow a - b = 0$ с ограничением, что b не является синтаксически нулём. Применим его к терму $4x + 7 = 2x + 1$ из примера 1:

1. Подстановка: $\{a \mapsto 4x + 7, b \mapsto 2x + 1\}$.
2. Ограничение проверяется тривиальным сравнением термов.
3. Новый терм: $(4x + 7) - (2x + 1) = 0$.

2.3. Процесс обучения

Система представляет собой непосредственно алгоритм, который будем называть *ядром*, и набор правил, записанных на специальном языке. *Обучением* системы мы будем называть добавление в систему новых правил, либо внесение изменений в существующие для расширения класса решаемых задач, а также уменьшения времени, затрачиваемого системой на поиск ответа.

3. Переключение внимания

Важным вопросом при реализации ядра системы является порядок обхода термов. Классические подходы, такие как обход в глубину или обход в ширину, в данной задаче приводят к зацикливанию алгоритма на отдельных термах задачи, в то время как важные для получения ответа термы не будут рассмотрены вовсе.

Одно решение для этой задачи было предложено А. С. Подколзиным. Его суть состоит в ранжировании правил по уровням в зависимости от их *общеупотребимости* — доли задач, в которых срабатывание правила существенно ускоряет решение, либо является обязательным для получения ответа.

Описываемая система наследует решение, описанное А. С. Подколзиным: на этапе обучения каждому правилу приписывается целое неотрицательное число, называемое *уровнем правила* — L_r. При решении задачи каждому терму приписывается целое неотрицательное число, называемое

уровнем терма L_t. Вновь сгенерированному терму приписывается нулевой уровень. В процессе решения система пытается применять к терму уровня L_t правила уровня $L_r = L_t$. Если применение всех подходящих правил этого уровня не дало новых термов, уровень терма увеличивается на единицу. Заметим, что в системе А. С. Подколзина допускается, чтобы правило срабатывало на нескольких уровнях. Однако эта возможность используется там нечасто, поэтому в рассматриваемой здесь системе в целях простоты правило привязывается к конкретному уровню, и применение правила на нескольких уровнях при необходимости можно смоделировать созданием нескольких копий одного правила для всех уровней, на которых его предполагается пытаться применить.

В начале каждой итерации выбирается один из термов с наименьшим уровнем. Если таких термов несколько, выбирается тот, что был получен раньше. В случае, если все термы имеют больший уровень, чем максимальный существующий уровень правила, алгоритм завершается. Если к моменту завершения не был получен ответ, результатом считается специальный флаг, сигнализирующий о том, что решение не найдено.

4. Архитектура системы

В данном разделе описываются ключевые компоненты экспериментальной системы, их организация и взаимодействие. Основное внимание уделяется структурам данных, используемым для представления информации: правила, термы, контекст задачи и механизмы их обработки.

4.1. Система подзадач

Часто решение задачи требует промежуточных шагов, выходящих за рамки целевой установки самой задачи. Например, требуется упрощение терма или доказательство вспомогательного утверждения в рамках задачи на поиск значения неизвестной. В таких случаях используется механизм подзадач. *Подзадача* — это вспомогательная задача, которая решается независимо, и результат которой используется в основной задаче.

Подзадачи используют те же целевые установки, что были описаны ранее:

1. **Transform** — преобразование терма используется для упрощения всех новых термов в контексте задачи.
2. **Prove** — доказательство утверждения может быть использовано для проверки истинности ограничений в процессе применения правила.

3. **Find** — поиск значения переменной может быть использован для рассмотрения случаев с последующим объединением ответов.

Механизм подзадач также позволяет изолировать контекст решения задачи от попадания в него вспомогательных термов, полученных в ходе решения подзадачи. А также — значительно упрощает процесс отладки, позволяя быстрее находить места, в которых у системы возникли затруднения.

Подзадачи могут создавать свои подзадачи, образуя иерархию с ограничением глубины вложенности. Результаты решения подзадач кэшируются, чтобы избежать повторного решения одинаковых подзадач. Добавление подзадачи в кэш производится до начала её решения, при этом специальным образом отмечается, что решение задачи ещё не получено. При попытке рекурсивно решить такую подзадачу вместо ответа будет получен флаг, сигнализирующий о том, что решение не найдено.

4.2. Представление правил

Полный перебор всех правил для каждого терма приводит к большому количеству вычислений. В связи с этим актуальной становится задача разработки эффективных алгоритмов и структур данных, позволяющих на ранних этапах идентифицировать и исключать заведомо безуспешные попытки применения правил, что существенно повышает производительность системы в целом.

4.2.1. Привязка правил к функциональным символам

Наблюдения показывают, что значительная часть правил оказывается отсеяна на этапе сопоставления терма с шаблоном. Это наводит на мысль о поиске некоторого инварианта, который позволит сократить количество перебираемых правил на как можно более раннем этапе.

Таким инвариантом выступает требование присутствия всех функциональных символов шаблона в обрабатываемом терме как необходимое условие успешной унификации. Опираясь на этот факт, А. С. Подколзин предлагает привязывать правила к символу из шаблона. При обработке терма система выбирает только те правила, ключевые функциональные символы которых присутствуют в данном терме, что существенно сокращает пространство перебора. Выбор символа осуществляется пользователем при добавлении правила.

В описываемой системе дополнительно производится проверка полного вхождения символов шаблона в терм до попытки сопоставления.

4.2.2. Разбиение правил по уровням

Как было описано в разделе 3, система применяет к терму лишь те правила, чей уровень соответствует уровню терма $L_r = L_t$. При поиске подходящих правил разумно ограничиться лишь множеством правил этого уровня.

4.2.3. База правил

Для эффективного поиска подходящих правил система организует их в специальную структуру — *базу правил*. Учитывая оптимизации, описанные в подразделах 4.2.1 и 4.2.2, база правил строится как двухуровневая структура. На первом уровне правила распределяются по уровням срабатывания L_r, соответствующим уровням термов. На втором уровне для каждого значения L_r создаётся отображение, связывающее каждый функциональный символ со списком правил данного уровня, привязанных к данному символу.

Алгоритм поиска правил для терма t уровня L_t в базе правил $\mathcal{R}$ состоит из следующих шагов:

1. **Выбор уровня:** Получаем $\mathcal{R}[L_t]$ — подмножество правил, соответствующее уровню терма L_t.
2. **Выбор правил по символам:** Для каждого функционального символа s, присутствующего в терме t, извлекаем из $\mathcal{R}[L_t]$ список правил $\mathcal{R}_s[L_t]$, привязанных к символу s. Объединяем все полученные списки в множество кандидатов $\mathcal{R}_1[t]$.

$$\mathcal{R}_1[t] = \bigcup_{s \in t} \mathcal{R}_s[L_t]$$

3. **Фильтрация по входящим символам:** Из $\mathcal{R}_1[t]$ исключаем те правила, шаблон которых содержит хотя бы один функциональный символ, отсутствующий в t. В результате остаются только правила $\mathcal{R}_2[t]$, все символы шаблона которых присутствуют в обрабатываемом терме.

Такая организация позволяет на этапе сбора кандидатов быстро отбросить заведомо неприменимые правила (те, что не содержат ни одного общего символа с термом), а на этапе финальной фильтрации — уточнить результат, проверив полное вхождение символов шаблона.

Псевдокод данного алгоритма приведён в приложении В.

4.3. Представление контекста задачи

Термы используются системой для представления правил, целевых установок, условий и выведенных утверждений. Термы, представляющие утверждения, а также термы, описывающие целевую установку, образуют рабочее множество[1] $\mathbb{T}$. Все термы сопровождаются дополнительной информацией, которую будем называть *метаданными*. Метаданные включают в себя уровень терма, историю его вывода, списки применённых и заблокированных правил и другую служебную информацию.

4.3.1. Иммутабельность термов

Все термы в $\mathbb{T}$ неизменяемы (иммутабельны) — если применение правила порождает новый терм, отсутствующий в $\mathbb{T}$, он добавляется в $\mathbb{T}$, не затрагивая исходный. Это свойство касается только самого терма, но не его метаданных.

Для случаев, когда необходимо изменить исходный терм, предусмотрена его *замена*: терм помечается как неактивный и больше не выбирается для обработки в основном цикле. В текущей версии системы такой подход используется для замены терма после его упрощения с помощью подзадачи **Transform** или после применения к терму правила с атрибутом `replace`.

4.3.2. Представление терма

В системе используется классическое представление терма в виде древовидной структуры. Терм строится из трёх типов атомарных компонентов:

1. **Константы** — числовые значения.
2. **Функциональные символы** — n-арные операторы из заранее определённого множества для целых неотрицательных n. Узлы дерева представляют собой функциональные символы, аргументами которых являются дочерние термы.
3. **Идентификаторы** — строковые имена, не являющиеся константами или функциональными символами.

Рациональные числа представляются обыкновенными дробями, иррациональные числа — как результаты операций (например, $\sqrt{2}$), а широко

[1] Формально $\mathbb{T}$ — это мультимножество, где могут быть два синтаксически совпадающих терма разного происхождения: один выведенный из условий, другой — из целевой установки задачи.

применяемые константы (например, π, e) — как 0-арные функциональные символы.

Идентификаторы в системе бывают двух типов: *переменные* и *параметры*. Тип идентификатора зависит от роли терма в системе, таким образом все идентификаторы конкретного терма являются либо переменными, либо параметрами. Переменные используются в выводимых термах, в то время как параметры используются в термах правил.

4.3.3. Нормализация терма

Одной из самых трудоемких задач при работе системы остаётся сопоставление терма с образцом правила. Отчасти это связано с неоднозначным представлением семантики выражения с помощью терма. Так, например, ассоциативные операции допускают произвольную расстановку скобок, а коммутативные — возможность менять местами аргументы.

Чтобы сократить перебор возможных вариантов, система пытается привести все термы к некоторому каноническому виду. Так вложенные ассоциативные операции преобразуются в одну операцию от нескольких аргументов, аргументы коммутативных операций упорядочиваются. Помимо этого функциональный символ может иметь дополнительные процедуры приведения, описанные внутри системы.

Данная процедура носит название *нормализация*. Некоторые примеры нормализации:

- Приведение подобных слагаемых: $x + 2 * x + 3 \to 3 * x + 3$.
- Вычисление константных выражений: $2 + 3 * 5 \to 17$.
- Упорядочивание: $3 + x \to x + 3$.
- Удаление лишних скобок и операций: $1 * (x + y) + z \to +(x, y, z)$.

Важно различать нормализацию и упрощение термов. Нормализация — это строгое синтаксическое приведение к канонической форме, которое гарантированно сохраняет семантическую эквивалентность. Упрощение же, реализуемое через подзадачу **Transform**, может применять более сложные, контекстно-зависимые преобразования (например, вычисление производной или раскрытие скобок по формуле), подразумевающие семантическую корректность, обеспечиваемую корректностью введённых в систему правил преобразования.

Однако существуют ситуации, когда нормализация может быть нежелательна. Например, правило может специально представлять терм в неканонической форме, чтобы обеспечить срабатывание другого правила.

В таких случаях для конкретного терма или класса преобразований нормализация может быть временно отключена или ограничена с помощью соответствующих атрибутов правил.

В рамках процесса обучения системы (определённого в разделе 2.3) процедура нормализации остаётся неизменной. Её модификация не входит в стандартный цикл обучения и требует внесения изменений на уровне ядра системы.

Пример 3. Нормализация терма $(4x + 7) - (2x + 1) = 0$ из примера 2:

- Устранение вложенности сложения: $+(4 * x, 7, -1 * 2 * x, -1 * 1) = 0$.
- Арифметические вычисления: $+(4 * x, 7, -2 * x, -1) = 0$.
- Приведение подобных: $+((4 - 2) * x, (7 - 1)) = 0$.
- Арифметические вычисления: $+(2 * x, 6) = 0$.
- Итоговый нормализованный терм: $2x + 6 = 0$.

4.3.4. Отслеживание вывода

Каждый терм в системе сопровождается метаданными, которые фиксируют историю его происхождения — так называемый *источник вывода*. Эта информация позволяет восстановить полную цепочку рассуждений, приведшую к получению терма.

Система различает три основных типа источников:

- **Condition** — терм входит в условие задачи. Такие термы образуют начальное множество, от которого начинается процесс вывода.
- **RuleApplication** — терм получен в результате применения правила преобразования. В метаданных сохраняются идентификатор применённого правила, используемая подстановка параметров, а также доказательства всех ограничений, проверенных при применении.
- **Transform** — терм является результатом упрощения другого терма через подзадачу типа **Transform** (см. раздел 4.1). Метаданные содержат ссылку на контекст подзадачи, что позволяет при необходимости воспроизвести процесс упрощения.

Хотя источник вывода не влияет непосредственно на процесс применения правил во время решения (все термы равноправны с точки зрения алгоритма вывода), он выполняет несколько важных функций:

1. **Реконструкция решения**: после успешного завершения задачи система может восстановить полную последовательность шагов, демонстрируя логику рассуждений.

2. **Диагностика**: при возникновении затруднений у системы в процессе решения новой задачи анализ источников вывода помогает идентифицировать проблемные участки в цепочке рассуждений.
3. **Сбор статистики**: информация о частоте применения различных правил и эффективности преобразований служит ценным материалом для дальнейшего обучения и оптимизации системы.
4. **Верификация**: наличие полной истории вывода позволяет проводить апостериорную проверку корректности решения.

4.3.5. Блокировка правил

В процессе обучения системы могут возникнуть ситуации, когда к результату применения правила нежелательно применение другого правила. Например, после получения явного значения переменной нет смысла переносить правую часть влево и решать ещё раз линейное уравнение с тем же результатом.

Для предотвращения подобных ситуаций правило может хранить ссылки на правила, которые следует заблокировать для выведенного терма. Пометки о блокировке сохраняются в метаданных нового терма и используются в качестве фильтров на этапе выбора нового правила для этого терма.

4.3.6. Термы-подстановки

В процессе решения задач иногда возникает необходимость получения следствия одновременно из нескольких термов. Рассмотрение всех возможных подмножеств термов для такого преобразования является вычислительно трудоёмкой задачей.

Важным частным случаем такой проблемы являются термы вида *переменная = значение*, называемые *подстановками*. Обычно это равенство необходимо подставить в другие термы, содержащие эту переменную.

Обработка подстановок осуществляется в два этапа.

1. При обнаружении подстановки в метаданных соответствующего терма устанавливается специальный флаг.
2. Применение подстановки к другому терму производится с помощью специального правила лишь в тот момент, когда этот терм окажется выбран процедурой переключения внимания (см. раздел 3).

4.3.7. Краткий список метаданных терма

Ниже приведён сводный список ключевых метаданных терма t, используемых системой, часть из которых уже была рассмотрена в предыдущих разделах:

1. **Уровень терма** L_t — определяет приоритет обработки и ограничивает множество применяемых правил (см. раздел 3).
2. **Источник вывода** — указывает, как был получен терм (см. раздел 4.3.4).
3. **Фильтры правил**:
   - $\mathcal{R}_{\text{applied}}(t)$ — правила, уже применявшиеся к терму t (во избежание повторной обработки).
   - $\mathcal{R}_{\text{blocked}}(t)$ — правила, заблокированные для применения к t (см. раздел 4.3.5).
4. **Кэшированные данные**:
   - Symbols(t) — множество функциональных символов, входящих в терм. Вычисляется однократно и используется при поиске правил в базе (см. раздел 4.2.3).
5. **Флаги состояния**:
   - Simplified — терм уже был упрощён через подзадачу **Transform**.
   - IsSubstitution — терм имеет вид $x = v$ и может быть использован для подстановки (см. раздел 4.3.6).
   - IsAlternativeTarget — терм является альтернативной целевой установкой, например, равносильным утверждением в задаче на доказательство. Подробнее процесс получения альтернативных целевых установок описывается в разделе 5.1.
   - IsReplaced — терм был исключён из рассмотрения после применения правила с атрибутом `replace` или преобразования через подзадачу **Transform**.

Для удобства введём обозначение для множества активных термов:

$$\mathbb{T}_{\text{active}} = \{t \in \mathbb{T} \mid t.\text{IsReplaced} = \text{false}\}.$$

4.4. Инициализация и начальные преобразования

Перед запуском основного цикла (раздел 5) система преобразует исходную постановку задачи во внутреннее представление.

1. **Обработка условий задачи**.
   Каждое условие преобразуется в терм и нормализуется согласно разделу 4.3.3. Терму назначаются метаданные: уровень $L_t = 0$, источник $\text{Source}(t) = \text{Condition}$, вычисляется множество символов $\text{Symbols}(t)$, списки применённых и заблокированных правил инициализируются как пустые. Полученные термы образуют множество $\mathbb{T}_{\text{cond}}$.
2. **Обработка целевой установки**.
   Из целевой установки извлекается внутренний терм t_{goal} и преобразуется аналогично термам из условий: не будет противоречием считать источником целевой установки обобщённое условие задачи. Поэтому метаданные инициализируются следующим образом: уровень $L_t = 0$, источник $\text{Source}(t) = \text{Condition}$. Дополнительно терм целевой установки получает флаг состояния $\text{IsAlternativeTarget} = \text{true}$. Образуется множество $\mathbb{T}_{\text{goal}} = \{t_{\text{goal}}\}$.
3. **Инициализация структур данных**.
   Начальное рабочее множество термов получается объединением мультимножеств:
   $$\mathbb{T} = \mathbb{T}_{\text{cond}} \uplus \mathbb{T}_{\text{goal}}.$$
   Создаётся пустой кэш решённых подзадач из раздела 4.1.
4. **Инициализация служебных структур**.
   Вычисляется максимальный уровень правил $L_{\max}$. Сохраняется информация о типе исходной цели.

После выполнения этих шагов система переходит к основному циклу работы.

5. Цикл работы алгоритма

Решение задачи происходит в цикле, на каждой итерации которого выполняются следующие шаги:

1. **Выбор терма** — из множества активных термов $\mathbb{T}_{\text{active}} \subseteq \mathbb{T}$ для обработки выбирается первый в порядке получения терм t с наименьшим уровнем. Это может быть терм из условия задачи, ранее

выведенный терм или терм целевой установки (подробнее о преобразованиях терма целевой установки в разделе 5.1).

2. **Упрощение** — попытка упростить терм через подзадачу **Transform**, если ранее терм не был упрощён (подробнее про подзадачи в разделе 4.1). Для этого вызывается подзадача с целевой установкой **Transform**(t), за исключением случая, когда исходная задача уже имеет целевую установку вида **Transform**$(\dots)$. Исходный терм заменяется на ответ подзадачи с отметкой, исключающей повторное срабатывание.
3. **Усмотрение ответа** — проверка, является ли выбранный терм ответом на задачу (подробнее про усмотрение ответа в разделе 5.2).
4. **Создание пометок** — проверяется, может ли терм быть использован для замены переменной, как описано в 4.3.6. В случае успеха добавляется соответствующая пометка.
5. **Формирование списка правил** — из базы правил $\mathcal{R}$ выбирается список правил $\mathcal{R}_2[t]$ для терма t текущего уровня L_t, как это было описано в разделе 4.2.3. Затем происходит дополнительная фильтрация: отбрасываются правила, применённые к t ранее (раздел 4.3.7), либо заблокированные для применения к нему (раздел 4.3.5):

$$\mathcal{R}_3[t] = \mathcal{R}_2[t] \setminus \mathcal{R}_{\text{applied}}(t) \setminus \mathcal{R}_{\text{blocked}}(t)$$

6. **Применение правил** — система последовательно пытается применить каждое правило из отфильтрованного списка $\mathcal{R}_3[t]$ к терму t до первого успешного применения. Подробное описание этого процесса приведено в разделе 5.4.
7. **Обновление статуса терма** — Если в результате применения правила r с атрибутом замены был получен новый терм, исходный терм t помечается как заменённый (IsReplaced = true) и таким образом исключается из множества активных термов $\mathbb{T}_{\text{active}}$.
8. **Обновление уровня терма** — если процесс подошёл к концу списка правил, но не смог получить новых термов, уровень выбранного терма увеличивается на единицу, а алгоритм возвращается к выбору нового терма.

Цикл продолжается до одного из следующих условий:

- Найден ответ.
- Достигнут лимит итераций.
- Не удалось выбрать терм для обработки.

Псевдокод алгоритма приведён в приложении Д.

5.1. Преобразование целевого терма

Преобразуемым термом может быть терм из условия задачи, ранее выведенный терм или терм целевой установки. Это может быть полезно для упрощения доказываемого терма, либо для вывода термов, равносильных доказываемому. Термы, полученные преобразованием терма целевой установки, помечаются флагом состояния IsAlternativeTarget.

В задачах с целевой установкой вида **Prove**(...) выводится множество термов, равносильных доказываемому. Эти термы используются при усмотрении ответа (см. раздел 5.2).

Задача с целевой установкой вида **Transform**(...) подразумевает преобразование целевой установки для получения ответа.

5.2. Усмотрение ответа

Процедура *усмотрения ответа* определяет, является ли данный терм t ответом на задачу с текущей целевой установкой. Эта проверка вызывается в цикле работы системы (см. раздел 5) для каждого выбранного терма.

Система предоставляет возможность явно формировать ответ на задачу в выводе правила, что позволяет реализовать сложную логику усмотрения ответа на этапе обучения решателя.

Также присутствуют некоторые общие правила усмотрения ответа.

- **Prove**(s): терм t считается ответом, если t не является целевым термом сам по себе (t.IsAlternativeTarget = false), и синтаксически совпадает с s или с одним из термов, помеченных флагом состояния IsAlternativeTarget (см. 5.1).
- **Find**(x): терм t считается ответом, если он имеет одну из следующих форм:
  1. $x = v$, где v — известное выражение (подробнее термы, считающиеся известными, описаны в разделе 5.3).
  2. $x \in A$, где A — известное.

В отличие от других типов задач, раннее усмотрение ответа для Transform не производится. Процесс преобразования заключается в генерации цепочки термов с флагом IsAlternativeTarget. По завершении основного цикла, но перед завершением алгоритма, ответом помечается последний сгенерированный терм, имеющий этот флаг.

5.3. Концепция «Известное» (known)

Важным понятием в системе является концепция *«известное»* (known). Изначально известными считаются числовые константы. Также допускается пометка известными переменных в условии задачи, если допускается их появление в ответе (так называемые задачи с параметром).

Проверка, что терм является известным, осуществляется через **Prove**-подзадачи. Чаще всего правила выводят известность для арифметических операций над известными термами.

Концепция «известное» имеет важное значение в процедуре усмотрения ответа, а также в ограничениях для срабатывания правила. Для задач типа **Find**(x) ответ считается найденным, если получен терм вида *переменная* = *значение*, где значение известно, либо когда *переменная* $\in A$, где A — известно.

5.4. Применение правил

Применение правил к текущему терму t выполняется последовательно для каждого правила $r_i \in \mathcal{R}_3[t]$ из списка правил, подготовленных для применения к данному терму.

1. **Поиск подстановок параметров**
   Для правила r_i строится последовательность $\{p_k\}$ всевозможных подстановок параметров, обращающих шаблон r_i в терм t.
2. **Формирование гипотезы**
   Подстановка p_k применяется к выводу правила r_i, образуя новый терм $\hat{t}$. Также подстановка p_k применяется к ограничениям правила r_i для получения списка требований $\{c_m\}$, необходимых для обоснования терма $\hat{t}$. Пара $(\hat{t}, \{c_m\})$ в системе называется *гипотезой*. Термы гипотезы проходят нормализацию в процессе её формирования.
3. **Исключение дубликатов**
   Прежде чем приступать к обоснованию гипотезы, система проверяет, не встречается ли $\hat{t}$ среди выведенных ранее термов с тем же значением флага IsAlternativeTarget. В случае, если $\hat{t}$ уже был выведен, система переходит к следующей подстановке p_{k+1}, не тратя время на обоснование уже полученного утверждения.
4. **Обоснование гипотезы**
   Обоснование гипотезы осуществляется методом последовательного решения подзадач с целевой установкой **Prove**(c_m) (подробнее подзадачи описаны в разделе 4.1). Если какая-то из этих задач не

может быть решена системой, происходит переход к следующей подстановке p_{k+1}.

5. **Добавление нового терма**
   При успешном обосновании гипотезы терм $\hat{t}$ добавляется в множество выведенных термов $\mathbb{T}$ со следующими метаданными:
   - Уровень: $L_{\hat{t}} = 0$
   - Источник: **RuleApplication** с указанием применённого правила r_i и подстановки параметров p_k.
   - Список применённых правил: пустой.
   - Список заблокированных правил: правила, указанные как блокируемые в r_i.
6. **Обновление списка применённых правил**
   Если для правила r_i все допустимые подстановки параметров исчерпаны, но ни одна из них не привела к добавлению нового терма, правило помечается как применённое к терму t во избежание повторной обработки, а алгоритм переходит к следующему правилу r_{i+1}.

Правило может быть применено не только к терму t, но и к его подтерму t'. В таком случае новый терм $\hat{t}'$ получается из t методом замены подтерма t' на вывод правила r_i с применённой подстановкой p_k. Все шаги применения правил при этом проводятся аналогично.

Псевдокод алгоритма применения одного правила к терму приведён в приложении Г.

Пример 4. Рассмотрим правило для решения линейного уравнения:

$$ax + b = 0 \Rightarrow x = -b/a$$

Его ограничения: $\{a \neq 0, b$ — известно$\}$.

И покажем, как оно может быть применено к результату нормализации из примера 3: $2x + 6 = 0$.

1. Подстановка параметров: $\{a \mapsto 2, b \mapsto 6, x \mapsto x\}$. Подстановка $x \mapsto x$ может показаться бессмысленной, однако стоит помнить, что идентификатор слева является параметром, а идентификатор справа — переменной (см. 4.3.2).
2. Гипотеза: $x = -3$, требования: $2 \neq 0$, 6 — известно.
3. Принятие гипотезы, поскольку все требования — истинные утверждения.

4. Новый терм: $x = -3$.

Заметим, что возможна альтернативная подстановка:

$$\{a \mapsto x, b \mapsto 6, x \mapsto 2\},$$

однако требование $x \neq 0$ удастся доказать только после решения исходного уравнения, поэтому такая гипотеза будет отвергнута системой.

6. Языки системы

В данном разделе описываются языки, используемые для описания правил и задач в экспериментальной системе. Приведённые ниже примеры синтаксиса служат для иллюстрации описанного подхода — язык находится в разработке и может существенно изменяться.

6.1. Язык описания правил

Архитектура системы не зависит от конкретного синтаксиса записи правил, что позволяет в будущем адаптировать или полностью заменить язык описания правил без изменения ядра системы. Однако для формирования более конкретного представления о процессе обучения системы и лучшего понимания описанных ранее механизмов ниже приводятся примеры описания правил, используемых в текущей реализации.

6.1.1. Структура правила

Правила преобразования термов описываются с использованием специального синтаксиса, который позволяет формально задать шаблоны преобразований и условия их применения. Новое правило создаётся с помощью ключевого слова `rule`, далее в фигурных скобках перечисляется содержимое правила.

```
rule {
    attr <атрибуты_правила>;

    <шаблон> => <результат>;

    <ограничение_1>,
    <ограничение_2>,
    ...;
}
```

Листинг 1: Общая структура правила

6.1.2. Основная часть правила

Первая строка в теле правила, не выделенная ключевым словом, считается действующей частью правила и должна иметь вид:

1. `pattern => resolution;` — одностороннее преобразование
2. `pattern <=> resolution;` — равносильное преобразование

Отличие между двумя приведёнными записями состоит в том, что переход (2) считается равносильным. Это позволяет применять правило к цели задачи на доказательство, чтобы получать альтернативную цель, эквивалентную исходной. По возможности следует использовать именно форму (2) там, где это возможно.

6.1.3. Ограничения правила

Последующие строки считаются ограничениями (на наборы параметров) правила; они перечислены через запятую и оканчиваются точкой с запятой. Ограничения проверяются перед применением правила.

6.1.4. Атрибуты правил

Ключевое слово `attr` позволяет задавать атрибуты правила:

- `level(<num>)` — уровень срабатывания;
- `goal(<target>)` — ограничение по целевой установке;
- `id(<id>)` — уникальный идентификатор правила;
- `replace` — в случае успешного применения исключает исходный терм из рассмотрения в процессе дальнейшего решения (см. раздел 4.3.1);
- `block(<rule_id>)` — блокировка применения указанного правила к результату;
- `zero(<param>)` — требование рассмотрения случая, когда параметр равен нулю;
- `one(<param>)` — требование рассмотрения случая, когда параметр равен единице;
- `normalize(<level>)` — уровень нормализации результата применения правила. Каждый уровень включает операции предыдущих уровней. 0 — только раскрытие скобок ассоциативных операций, 1 — устранение тривиальных операций (добавление и вычитание

нуля, умножение на единицу и т. д.), упорядочивание аргументов коммутативных операций, 2 — проведение арифметических вычислений, 3 — приведение подобных, сворачивание степеней (например, $x \cdot x \to x^2$).

6.1.5. Генератор правил

Атрибуты `zero(<param>)` и `one(<param>)` позволяют описать несколько логически связанных случаев преобразований в рамках единого правила. Это полезно в случаях, когда шаблонное выражение может сильно поменять свой вид при обращении параметров в 0 или 1 из-за отсутствия соответствующих слагаемых или множителей.

6.1.6. Примеры

Рассмотрим пример простого правила из школьной алгебры, переносящего правую часть уравнения влево с противоположным знаком (Листинг 2). Система применяет это правило при решении уравнений, чтобы привести одно из условий к виду, узнаваемому другими правилами. Отсюда целевая установка `find(x)`. Заметим, что `x` не используется в терме правила или его ограничениях; это означает, что `x` может быть каким угодно, лишь бы сама целевая установка относилась к задаче на поиск значений переменных.

Атрибут `level(0)` устанавливает для правила максимальный приоритет. Таким образом, правило будет применено к новым термам раньше, чем правила с уровнем выше. Однако возможны случаи, когда это правило потребуется заблокировать. Чтобы иметь возможность указать это правило в качестве блокируемого, ему присваивается текстовый идентификатор `id(move_left)`. Пример блокировки правила будет рассмотрен далее.

Единственное ограничение проверяет, что подтерм `b` не равен терму `0` синтаксически, что отличается от ограничения `b != 0`, которое подразумевает семантическое равенство (в таком случае системе бы пришлось доказать, что выражение b не обращается в ноль в контексте задачи). Это ограничение носит оптимизационный характер и говорит о том, что константу 0 не нужно переносить влево. Конечно, можно было и не указывать этот момент, потому что при нормализации лишний ноль в левой части будет удалён, и новый терм совпадёт со старым и не будет добавлен к списку термов, однако проверить такое ограничение быстрее.

```
rule {
    attr replace, level(0), goal(find(x)), id(move_left);

    a == b <=> a - b == 0;

    !symbolic_eq(b, 0);
}
```

Листинг 2: Правило переноса в левую часть

Рассмотрим ещё один пример правила, применяемого при решении линейных уравнений (Листинг 3). Стоит отметить применяемый генератор правил. При обращении параметра `a` в единицу, либо при обращении в ноль параметра `b` терм, семантически являющийся линейным уравнением, перестаёт синтаксически подходить под общий шаблон. Данная языковая конструкция требует от системы подставить указанные значения параметров для получения альтернативных вариантов шаблона и провести их нормализацию.

Конструкция `block(move_left)` блокирует применение правила переноса в левую часть, которое могло бы помешать усмотрению ответа, либо сделать замену переменной `x` на правую часть выражения в другом терме.

Требования в данном правиле лишь уточняют определение линейного уравнения. Конструкция `b is known` требует отсутствия вхождения неизвестных в подтерм `b`, в частности отсутствие вхождения `x`.

```
rule {
    attr level(1), goal(find(x)), one(a), zero(b), block(
    move_left);

    a*x + b == 0 => x == - b/a;

    a != 0,
    b is known;
}
```

Листинг 3: Правило решения линейного уравнения

Последний пример приведён с целью продемонстрировать ограничения работы нормализации (Листинг 4). В некоторых задачах может потребоваться явное раскрытие скобок. Данный пример осуществляет понижение степени методом отделения множителя, когда выражение под степенью является суммой нескольких известных аргументов, например, когда одно из слагаемых является иррациональным числом, записанным с помощью радикалов. Однако при нормализации может быть обнаружено произведение одинаковых выражений, и множитель вернётся под знак

степени до того, как другое правило раскроет скобки в операции умножения. Здесь используется понижение уровня нормализации до второго, где такое преобразование уже не происходит.

```
rule {
   attr replace, level(4), goal(transform(x)), normalize(2);

   (a+b)^n <=> (a+b)^(n - 1)*(a + b);

   a is known,
   b is known;
}
```

Листинг 4: Правило разложения степени суммы

6.2. Язык описания задач

Синтаксис задач позволяет задать условия и целевую установку. Новая задача вводится с помощью ключевого слова `task`, далее в фигурных скобках вводятся данные задачи. Ключевое слово `goal` обозначает целевую установку задачи.

```
task {
    goal <целевая_установка>;
    text "Текст задачи для пользователя";
    answer Терм ответа для проверки;

    <условие_1>;
    <условие_2>;
    ...
}
```

Листинг 5: Общая структура задачи

6.2.1. Целевые установки

Целевые установки полностью соответствуют описанным ранее:

1. `find(x)` — выражение значения переменной x через известные параметры.
2. `prove(<statement>)` — доказать истинность утверждения (вывести его из условий задачи).
3. `transform(<statement>)` — преобразовать (упростить) выражение.

Целевая установка `goal` задачи обязательна, все прочие элементы опциональны.

6.2.2. Дополнительные элементы

- Ключевое слово `text` позволяет ввести текст задачи на любом языке. Этот текст никак не используется системой при решении задачи, он просто выводится пользователю.
- Ключевое слово `answer` позволяет задать ожидаемый ответ на задачу. При решении система не использует этот ответ, однако проводит сверку в конце процедуры. Таким образом, накопленная база задач выступает в роли набора интеграционных тестов для системы.
- Все прочие утверждения считаются условиями задачи.

6.2.3. Примеры

Приведем несколько примеров вводимых задач.

```
task {
    goal find(x);

    2*(x+4) + 3*(x-5) == 0;
}
```

Листинг 6: Линейное уравнение

```
task {
    goal find(x);

    2*x-5+t == 0;
    t is known;
}
```

Листинг 7: Линейное уравнение с параметром

```
task {
    goal find(x);

    3*x^2+2*x-5 == 0;
}
```

Листинг 8: Квадратное уравнение

```
task {
    goal prove(x > 0);

    x == 2;
}
```

Листинг 9: Простейшая задача на доказательство требующая подстановки

Список литературы

[1] А. С. Подколзин, *Компьютерное моделирование логических процессов*. Т. 1: *Архитектура и языки решателя задач*, Физматлит, Москва, 2008.

[2] Дж. Малпас, *Реляционный язык пролог и его применение*, Наука, Москва, 1990.

[3] Э. Мендельсон, *Введение в математическую логику*, Наука, Москва, 1971.

[4] F. Baader, T. Nipkow, *Term Rewriting and All That*, Cambridge University Press, Cambridge, 1998. DOI: 10.1017/CBO9781139172752.

Приложение А. Исходный код программы

Исходный код программы доступен по ссылке:
`https://github.com/Quotique/laffie/tree/v0.5`

Приложение Б. Пример работы алгоритма

Данный пример демонстрирует работу алгоритма на примере простой задачи: целевая установка $\mathbf{Find}(x)$, условия: $\{4x + 7 = 2x + 1\}$.

Используемые правила

1. r_1: $a = b \Rightarrow a - b = 0; L_{r_1} = 0;$ $\{$`!symbolic_eq`$(b, 0)\}$ — правило переноса в левую часть.
2. r_2: $a \cdot u + b = 0 \Rightarrow u = -b/a; L_{r_2} = 1;$ $\{a \neq 0,$ b `is known`$\}$ — правило решения линейного уравнения.

Шаги решения

0. Инициализация:
   - goal : Find(x).
   - t_1: $4x + 7 = 2x + 1$, $L_{t_1} = 0$.

- t_2: x, $L_{t_2} = 0$, IsAlternativeTarget = true.

1. Выбран $t = t_1, L_t = 0$.
   - Производится упрощение терма через подзадачу Transform($4x + 7 = 2x + 1$), однако в системе отсутствуют правила, применимые для упрощения этого терма, поэтому терм помечается как упрощённый: t_1.Simplified $\leftarrow$ true.
   - Терм t_1 не имеет вид $x = \ldots$ или $x \in \ldots$, поэтому не является ответом на задачу.
   - Выбирается правило r_1 уровня $L_r = L_t$: $a = b \Rightarrow a - b = 0$.
   - Выбирается подстановка: $P = \{a \mapsto 4x + 7, b \mapsto 2x + 1\}$.
   - Формирование гипотезы: $(4x + 7) - (2x + 1) = 0$.
   - Нормализация: $2x + 6 = 0$.
   - Проверка ограничения: `!symbolic_eq`$(2x + 1, 0)$ — истина.
   - Добавление нового терма: $t_3 : 2x + 6 = 0$, $L_{t_3} = 0$.
2. Выбран $t = t_1, L_t = 0$.
   - Упрощение терма и проверка на ответ повторно не производятся.
   - Правило r_1 не допускает альтернативных подстановок, дубликат терма t_3 пропускается.
   - Других правил уровня 0 не найдено, уровень терма увеличивается: $L_{t_1} \leftarrow 1$.
3. Выбран $t = t_2, L_t = 0$.
   - Аналогично t_1, t_2 не может быть упрощён (t_2.Simplified $\leftarrow$ true) и не является ответом на задачу.
   - Правил для преобразования t_2 не найдено, уровень увеличивается: $L_{t_2} \leftarrow 1$.
4. Выбран $t = t_3, L_t = 0$.
   - Аналогично, терм не может быть упрощён и помечается как упрощённый: t_3.Simplified $\leftarrow$ true.
   - Терм не является ответом.
   - Выбирается правило r_1 уровня $L_r = L_t$.
   - Выбирается подстановка: $P = \{a \mapsto 2x + 6, b \mapsto 0\}$.
   - Формирование гипотезы: $2x + 6 - 0 = 0$.
   - Нормализация: $2x + 6 = 0$.

- Данный терм уже есть в списке термов, обоснование не производится.
- Откат к выбору подстановки. Других подстановок не найдено.
- Переход к следующему правилу.
- Других правил не найдено, уровень увеличивается: $L_{t_3} \leftarrow 1$.

5. Выбран $t = t_1, L_t = 1$.
   - Упрощение терма и проверка на ответ повторно не производятся.
   - Правил для преобразования t_1 не найдено, уровень увеличивается: $L_{t_1} \leftarrow 2$.
6. Выбран $t = t_2, L_t = 1$.
   - Упрощение терма и проверка на ответ повторно не производятся.
   - Правил для преобразования t_2 не найдено, уровень увеличивается: $L_{t_2} \leftarrow 2$.
7. Выбран $t = t_3, L_t = 1$.
   - Упрощение терма и проверка на ответ повторно не производятся.
   - Выбирается правило r_2 уровня $L_r = L_t$: $a \cdot u + b = 0 \Rightarrow u = -b/a$.
   - Выбирается подстановка: $P = \{a \mapsto x, b \mapsto 6, u \mapsto 2\}$.
   - Формулируется гипотеза: $2 = -6/x, \{x \neq 0,\ 6\ \texttt{is known}\}$.
   - Для обоснования гипотезы вызывается подзадача Prove($x \neq 0$). В рамках этой подзадачи не будет выведено новых термов, ввиду отсутствия подходящих правил, поэтому задача так и останется нерешённой.
   - Гипотеза отвергается, происходит откат к выбору подстановки.
   - Выбирается подстановка: $P = \{a \mapsto 2, b \mapsto 6, u \mapsto x\}$.
   - Формулируется гипотеза: $x = -6/2$, $\{2 \neq 0,\ 6\ \texttt{is known}\}$.
   - Нормализация: $x = -3$, $\{2 \neq 0,\ 6\ \texttt{is known}\}$.
   - Ограничения выполнены тривиально.
   - Добавление нового терма: t_4: $x = -3$, $L_{t_4} = 0$.
8. Выбран $t = t_4, L_t = 0$.
   - Терм помечается как упрощённый: t_4.Simplified $\leftarrow$ true.
   - Проверка на ответ: терм имеет форму $x = v$, где $v = -3$ — известное значение.
   - **Ответ найден**: $x = -3$.

Приложение В. Алгоритм поиска правил в базе правил

Алгоритм 1: Поиск правил для терма в базе правил

Вход : Терм t, L_t, база правил $\mathcal{R}$
Выход: $\mathcal{R}_2 \subseteq \mathcal{R}$

$\mathcal{R}_1 \leftarrow \emptyset$
for each $s \in \text{Symbols}(t)$ **do**
    $\mathcal{R}_1 \leftarrow \mathcal{R}_1 \cup \mathcal{R}_s[L_t]$
$\mathcal{R}_2 \leftarrow \emptyset$
for each $r \in \mathcal{R}_1$ **do**
    if $\text{Symbols}(t) \supseteq \text{Symbols}(r.\text{pattern})$ **then**
        $\mathcal{R}_2 \leftarrow \mathcal{R}_2 \cup \{r\}$
return $\mathcal{R}_2$

Приложение Г. Алгоритм применения правил к терму

Алгоритм 2: Применение правил к терму

Вход : t, $\mathcal{R}_3$, $\mathbb{T}$, Goal
Выход : $(\text{found}, t_{\text{new}})$

for each $r \in \mathcal{R}_3$ **do**
  if Goal.type = Prove **and** t.IsAlternativeTarget **and** r — *неэквивалентное преобразование* **then**
    continue
  for each $m \in \text{matches}(r.\text{pattern}, t) \cap \text{matches}(r.\text{goal}, \text{Goal})$ **do**
    for each $\sigma \in \text{substitutions}(m)$ **do**
      $\hat{t}_{\text{raw}} \leftarrow \sigma(r.\text{conclusion})$
      $\hat{t} \leftarrow \text{normalize}(\hat{t}_{\text{raw}})$
      if $\hat{t} \in \{s \in \mathbb{T} \mid s.\text{IsAlternativeTarget} = t.\text{IsAlternativeTarget}\}$ **then**
        continue
      ok $\leftarrow$ true
      for each $c \in r.\text{constraints}$ **do**
        $c_{\text{norm}} \leftarrow \text{normalize}(\sigma(c))$
        if $\neg\text{prove}(c_{norm})$ **then**
          ok $\leftarrow$ false
          break
      if ok **then**
        $\hat{t}$.IsAlternativeTarget $\leftarrow t$.IsAlternativeTarget
        $\hat{t}$.IsReplaced $\leftarrow$ false
        $\hat{t}$.Simplified $\leftarrow$ false
        $\mathbb{T} \leftarrow \mathbb{T} \cup \{\hat{t}\}$
        $L_{\hat{t}} \leftarrow 0$
        if r.replace **then**
          t.IsReplaced $\leftarrow$ **true**
        $\mathcal{R}_{\text{applied}}(\hat{t}) \leftarrow \emptyset$
        $\mathcal{R}_{\text{blocked}}(\hat{t}) \leftarrow r.\text{blocks}$
        $\text{Inference}(\hat{t}) \leftarrow (r, \sigma)$ + доказательства условий r.constraints
        return $(\text{true}, \hat{t})$
  $\mathcal{R}_{\text{applied}}(t) \leftarrow \mathcal{R}_{\text{applied}}(t) \cup \{r\}$
return $(\text{false}, 0)$

Приложение Д. Основной алгоритм работы системы

В данном приложении представлен псевдокод основного цикла работы алгоритма, детальное описание которого приведено в разделе 5.

Алгоритм 3: Основной алгоритм работы системы

Вход : $\mathbb{T}$, Goal, $\mathcal{R}$, $I_{\max}$ — лимит итераций
Выход : (found, ans)

$L_{\max} \leftarrow \max_{r \in \mathcal{R}} L_r$
found $\leftarrow$ false
for $i \leftarrow 1$ **to** I_{max} **do**
  if $\mathbb{T}_{\text{active}} = \varnothing$ **then**
    break
  $t \leftarrow \arg\min_{t' \in \mathbb{T}_{\text{active}}} L_{t'}$
  if $L_t > L_{\max}$ **then**
    break
  if $\neg t$.Simplified $\wedge$ Goal.type $\neq$ Transform **then**
    $t' \leftarrow \text{transform}(t)$
    if $t' \neq t$ **then**
      t'.IsAlternativeTarget $\leftarrow$ t.IsAlternativeTarget
      Inference(t') $\leftarrow$ контекст подзадачи **Transform**
      $L_{t'} \leftarrow 0$
      t.IsReplaced $\leftarrow$ **true**
      t'.Simplified $\leftarrow$ **true**
      $\mathbb{T} \leftarrow \mathbb{T} \cup \{t'\}$
      $t \leftarrow t'$
      continue
    else
      t.Simplified $\leftarrow$ **true**
  if is_answer(t, Goal) **then**
    found $\leftarrow$ **true**
    ans $\leftarrow t$
    break
  if check_substitution(t) **then**
    t.IsSubstitution $\leftarrow$ **true**
  $R \leftarrow \text{query\_rules}(\mathcal{R}, L_t, t) \setminus \mathcal{R}_{\text{applied}}(t) \setminus \mathcal{R}_{\text{blocked}}(t)$
  (ok, t) $\leftarrow$ apply_rules($t, R, \mathbb{T}$, Goal)
  if $\neg$ok **then**
    $L_t \leftarrow L_t + 1$
if Goal.type = Transform $\wedge \neg$found **then**
  found $\leftarrow$ **true**
  ans $\leftarrow$ last_from $\{t \in \mathbb{T} \mid t.\text{IsAlternativeTarget}\}$
return (found, ans)

Статья поступила 29 января 2026 г.

Architecture of a Simplified Automated Reasoning System Based on A. S. Podkolzin's Approach

A. P. Annenkov

Solving formalizable mathematical problems using computers remains a relevant challenge with several fundamentally different approaches. A significant contribution to this field was made by A. S. Podkolzin, who proposed an original solver architecture based on logical inference using term rewriting rules. This article describes the architecture of a simplified system that implements the key ideas of this approach: an attention-switching mechanism via levels, organization of a rule base, and context representation with metadata.

Keywords: mathematical problem solver, automated reasoning, logical processes, logical language, logical formalization of problems.

References

[1] A. S. Podkolzin, *Computer modeling of logical processes*. V. 1: *Architecture and languages of the problem solver*, Fizmatlit, Moscow, 2008.

[2] J. Malpas, *Prolog: A Relational Language and Its Applications*, Prentice-Hall, Englewood Cliffs, N. J., 1987.

[3] E. Mendelson, *Introduction to Mathematical Logic*, Van Nostrand, Princeton, N. J., 1964.

[4] F. Baader, T. Nipkow, *Term Rewriting and All That*, Cambridge University Press, Cambridge, 1998. DOI: 10.1017/CBO9781139172752.

Received on January 29, 2026