Московский государственный университет имени М.В. Ломоносова

Факультет Вычислительной математики и кибернетики

На правах рукописи

Корухова Юлия Станиславовна

СИСТЕМА АВТОМАТИЧЕСКОГО СИНТЕЗА ФУНКЦИОНАЛЬНЫХ ПРОГРАММ

Специальность 05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:

кандидат физико-математических наук,

доцент В.Н. Пильщиков

Москва

2005

Оглавление

Введение. Существующие подходы к синтезу программ

На современном этапе развития программирования компьютеры всё больше начинают применяться не только на этапе кодирования программы, но и во время постановки задачи, для решения которой требуется написать программу. В пятидесятые годы 20-го века, когда только приступали к автоматизации программирования, явно выделялись две фазы программирования. Под автоматизацией первой фазы подразумевалось автоматическое получение алгоритма. Вторая фаза заключалась в получении машинной программы по сформулированному алгоритму. В течение тридцати лет автоматизировалась в основном только вторая фаза. С появлением новых разработок по искусственному интеллекту и развитием технологии программирования стала возможной и автоматизация первой фазы. В 70-е годы появилась самостоятельная область исследований – автоматический синтез программ. Под синтезом понимается получение текста программы на основе информации, описывающей эту программу.

Существуют три различных подхода к синтезу программ:

- дедуктивный, при котором построение программы проводится на основе описания ее цели, заданной в виде спецификации (описания задачи). При таком подходе используется конструктивное доказательство утверждения о том, что решение задачи существует, и из него извлекается требуемая программа;
- индуктивный, при котором программа строится по примерам, непосредственно задающим ответ для некоторых исходных данных;
- трансформационный, где программа получается путем преобразования исходного описания задачи по правилам, совокупность которых представляет знания о решении задач. Трансформационный синтез позволяет получать из менее эффективных программ эквивалентные им, но более эффективные.

В данной работе мы будем рассматривать дедуктивный подход. Преимущество данного подхода в том, что для синтеза используются методы,

для которых формально доказано, что они порождают программу, соответствующую спецификации. Такая программа в дальнейшем не требует верификации, поэтому актуальной является задача разработки методов и алгоритмов, позволяющих проводить дедуктивный синтез автоматически.

Дедуктивный синтез программ

При дедуктивном подходе путь от задачи к программе выглядит так:

описание задачи → теорема существования решения → доказательство теоремы → программа.

Описание задачи содержится в спецификации; кроме того, используется некоторый набор аксиом, задающих свойства предметной области. Спецификация формулирует цель программы, описывая связь между входными и выходными данными. Хорошая спецификация близка к намерениям пользователя, понятна и читаема. Спецификация рассматривается как теорема существования некоторого объекта (выхода программы). Для спецификации-теоремы строится конструктивное доказательство. Такое доказательство, утверждая о существовании объекта, должно предъявлять способ его вычисления, по которому и будет напрямую сформирован текст программы на некотором языке программирования.

Задача дедуктивного синтеза программ, таким образом, напрямую связана с задачей доказательства теорем. Если же доказательство выполняется автоматически, то автоматически будет получена программа, точно соответствующая описанию задачи. Работа человека в этом случае сводится только к корректному описанию задачи.

В дальнейшем в работе будем подразумевать под термином синтез именно дедуктивный синтез программ.

Одной из первых систем синтеза, в которой был применен такой подход, была система **PROW** [Lee et al., 1974] Спецификации в этой системе записываются в следующем виде:

$\forall\ x\ (P(x) \supset \exists\ z.\ R(x,z))$

Эта запись означает, что по заданному x, удовлетворяющему входному условию программы P(x), требуется вычислить z, удовлетворяющее условию R(x,z), которое связывает вход x и выход z программы.

Доказательство теоремы строится в исчислении предикатов первого порядка на основе метода резолюций [Robinson, 1965].

Система PROW показала возможность синтеза линейных и ветвящихся программ. Для синтеза рекурсии или цикла необходимо было использовать особое правило вывода, которое опирается на метод математической индукции.

Синтез программ, содержащих цикл или рекурсию

Выявление правил логики, лежащих в основе построения циклических и рекурсивных программ, было важным шагом в автоматизации синтеза программ. Для появления таких структур в программе требовалось проводить доказательство по индукции.

В системе ПРИЗ [Тыугу, 1984] была реализована возможность автоматического синтеза программ, содержащих циклы. В системе заранее был задан набор управляющих структур цикла для каждой возможной версии математической индукции. Построение тела цикла при этом рассматривалось как подзадача.

Синтез в системе ПРИЗ рассматривается как составление алгоритма решения задачи на так называемой вычислительной модели. Вычислительная модель представляет собой модель предметной области и формируется заранее. Она состоит из совокупности переменных, соответствующих понятиям предметной области, и отношений вычислимости. Каждое из отношений вычислимости связывает значения нескольких переменных модели и интерпретируется так: по известным значениям одних переменных (являющихся входными) можно вычислить значения других (выходных) переменных. Задача на вычислительной модели состоит в нахождении значений некоторых переменных по известным значениям других переменных на основе отношений вычислимости. Спецификация, записанная на специальном языке УТОПИСТ,

состоит из двух частей: декларативной (указание значений входных переменных) и процедурной (которая представляет собой оператор задачи с перечисленными в нём выходными переменными). Дополнительно в декларативной части спецификации могут быть заданы ещё некоторые отношения между переменными.

Метод построения программы в системе ПРИЗ предполагает три этапа работы:

1. Перевод спецификации во внутреннее представление. На этом этапе, исходя из спецификации и отношений вычислимости вычислительной модели, формируется набор отношений, представляющий вычислительную модель решаемой задачи.
2. Планирование решения задачи. На этом этапе применяется либо метод «прямой волны» (при котором проводится преобразование переменных, начиная от входных, с учётом отношений вычислимости, до получения выходных переменных), либо метод «обратной волны» (при движении в обратном направлении – от выходных переменных к входным).
3. Запись алгоритма на одном из входных (целевых) языков системы.

Такой вариант метода синтеза называется структурным синтезом. При исследовании логических основ этого метода синтез на вычислительных моделях был тоже представлен как доказательство математической теоремы существования решения в теории специального вида. Формулы этой логической теории состоят из логических связок и отношений вычислимости, имеющих вид $u \to_f v$ (смысл этой записи: по u вычислимо v применением функции f). Тогда теорему существования решения задачи можно записать следующим образом: $\exists f . (u \to_f v)$. Задача состоит в выводе этой теоремы из аксиом, задающих отношения вычислимости.

Стратегия поиска доказательства в случае предложений вычислимости простейшего вида $u \to_f v$ заключается в последовательном выводе на основе этого правила и аксиом (известных отношений вычислимости) всех

возможных новых предложений, пока не будет выведено предложение, представляющее собой доказываемую теорему.

Ветвление появляется в программе, когда теорема существования не может быть доказана только на основе предложений простейшего вида и необходимо использовать предложения с условиями вычислимости. Для каждого такого условного предложения синтезируется ветвление, а ветви условного выражения строятся рекурсивным применением описываемого алгоритма при допущении истинности или, соответственно, ложности условия вычислимости.

В доказательство может быть включено предложение вычислимости с условием в виде теоремы существования решения другой задачи (подзадачи). Тогда в синтезируемой программе должен появиться вызов процедуры, реализующей решение этой подзадачи. При возникновении подзадачи с той же теоремой, что и доказываемая, возможно построение рекурсивной процедуры. Но на практике эта возможность не используется, так как в этом случае сильно затрудняется проверка завершаемости работы построенной рекурсивной процедуры. Циклы же в системе ПРИЗ используются довольно часто.

В системе ПРИЗ, таким образом, используется метод, дающий возможность синтезировать последовательность операторов, условное выражение и цикл. Кроме того, метод структурного синтеза позволяет сократить перебор предложений вычислимости, так как для каждой задачи строится вычислительная модель, а, следовательно, подбираются только подходящие аксиомы. Однако значительная часть работы для такого синтеза (описание вычислительной модели) должна быть проделана заранее. При решении задачи из другой области соответствующая дополнительная информация должна снова добавляться в систему и её становится сложно расширять.

Вариант правила вывода, позволяющий получить рекурсивное обращение к функции, был предложен Манной и Валдингером в **методе дедуктивных таблиц** [Manna and Waldinger, 1980, 1992]. Суть метода заключается в следующем: каждому шагу доказательства, проводимого на основе известных аксиом и правил вывода, соответствует определенный шаг синтеза. Применяя

дедуктивные правила, можно получить три базовые конструкции функционального языка программирования: применение функции, условное выражение и рекурсию, которые напрямую переводятся в конструкции функционального языка программирования (например, Лиспа [McCarthy, 1960], [Хювенен и Сеппянен, 1990]).

С помощью метода дедуктивных таблиц были синтезированы, например, программы сортировки для списков [Traugott, 1989] и алгоритм унификации [Nardi, 1989], но этот синтез проводился вручную. В интерактивном варианте метод дедуктивных таблиц был реализован в системе, описанной в работе [Burback et al., 1990]. Применение дедуктивного правила эта система выполняла автоматически, но выбор правила на каждом шаге доказательства делал пользователь.

При проведении доказательства обычно находится несколько правил вывода, которые могут быть применены в текущей ситуации, однако далеко не многие из них ведут к успешному завершению доказательства, и возникает огромный перебор вариантов. Для сокращения перебора либо доказательство должно быть сделано интерактивным, направляемым пользователем, либо необходимо использование дополнительных эвристик.

Автоматизированные системы доказательств

Один из методов, позволяющих сократить перебор вариантов применения правил вывода при построении доказательства, подразумевает направление этого доказательства человеком. Система, реализующая такой метод, автоматически выполняет рутинные операции, а решения о порядке применения правил, построении стратегии доказательства принимает пользователь. Одной из первых таких систем была система **Nqthm** [Boyer and Moore, 1979]. Часть работы при построении доказательств в этой системе выполняется автоматически с помощью метода резолюций, некоторых методов преобразования равенств и эвристик для структурной индукции, но взаимодействие с человеком требуется при поиске вывода, так как системе

нужно явно указывать некоторые промежуточные утверждения, используемые при доказательстве как вспомогательные леммы.

Другой подход – направление доказательства с помощью дополнительной информации, задаваемой по ходу его выполнения – был использован в системе **Isabelle** [Paulson, 1988]. В ней были разработаны так называемые тактики и тактические конструкции, позволяющие управлять процессом логического вывода при автоматизированном проведении доказательства. Тактика представляет собой объединение нескольких шагов доказательства в единое целое. Применение той или иной тактики, откат доказательства на предыдущий шаг, составление цепочки доказательств из нескольких тактик с помощью тактических конструкций – задачи, которые выполняет пользователь системы.

Состояние доказательства в системе представляет собой некоторую теорему. Тактики – это функции, преобразующие это состояние. Тактики резолюции, просматривая список правил, возвращают следующее состояние для каждой комбинации правила и унификатора. Тактики предположения выполняют присваивание и возвращают следующее состояние для каждой комбинации предположения и унификатора. Если применить тактику невозможно, система возвращает `fail`.

Примеры тактик:

`assume_tac i` Тактика пытается решить подцель i.

`resolve_tac thms i` Тактика резолюции. `thms` – это список правил, которые решают i-ю подцель. Для каждого из этих правил резолюция формирует следующее состояние, унифицируя заключение с подцелью, и подставляет конкретизированную предпосылку на место подцели. Результат применения будет ложным, если ни для одного правила не удалось создать следующее состояние.

В системе предоставлена возможность комбинирования тактик с помощью специальных операторов – тактических конструкций. Основные тактические конструкции: `THEN, ORELSE, REPEAT.`

`tac1 THEN tac2` Конструкция последовательного применения: применяется `tac1`, затем `tac2`. Если применение одной из тактик возвращает fail, то происходит возврат к начальному состоянию.

`tac1 ORELSE tac2` Применяется `tac1`; если применение прошло успешно, то `tac2` не применяется, иначе же применяется `tac2`.

`REPEAT tac` Тактика `tac` применяется до тех пор пока не вернет `fail`.

Для взаимодействия с пользователем в системе использован язык Standart ML [Harper, 1989]. В нем предоставлена возможность управления контекстом доказательства. Основными из возможностей являются следующие:

`Goal f` Начать доказательство теоремы `f`.

`by tac;` Применить тактику `tac` к текущему состоянию доказательства.

`undo();` Откатить состояние доказательства к предыдущему.

`result();` Возвращает теорему, которая уже доказана и `fail`, если еще есть недоказанные подцели.

`qued name` Обычный путь завершения доказательства. Доказанная теорема сохраняется в базе данных Isabelle.

В основе системы Isabelle лежит металогика, в которой объектные логики (например, логика первого порядка, теория множеств) могут быть представлены как её частные случаи. В металогике имеется набор аксиом, которые, будучи конкретизированными для объектной логики, становятся правилами вывода. Они и используются для проведения доказательств утверждений рассматриваемой объектной логики.

В работе [Ayari and Basin, 2001] описана реализация метода дедуктивных таблиц с помощью логики высшего порядка в системе Isabelle, таким образом, система может быть использована для синтеза программ. Вместо перебора вариантов применения правил перебор происходит на более высоком уровне – при выборе тактики доказательства. Недостатком такого подхода, на наш взгляд, является тот факт, что для многих примеров надо придумывать свою тактику доказательства, какой-либо универсальной тактики нет.

Системы автоматического синтеза с использованием планирования доказательств

Для сокращения перебора в некоторых системах применяется планирование доказательства. Суть его заключается в следующем: сначала составляется план – схема доказательства, записанная на более высоком уровне абстракции, а затем по нему строится уже само доказательство, в котором строго доказываются шаги, оставшиеся недоказанными при построении плана. Во многих системах план доказательства строится, исходя из знаний о предметной области, в которой проводятся рассуждения. При дедуктивном синтезе программ, содержащих рекурсию, возникает доказательство по индукции. Рассмотрим системы, в которых планирование применяется для сокращения перебора при построении доказательства по индукции. Это системы автоматического доказательства Oyster/Clam и λCLAM.

Oyster/Clam [Bundy et al., 1990] представляет собой объединение двух систем, Clam – планировщик доказательств, он строит подходящую тактику, которая потом выполняется системой Oyster.

Oyster – интерактивный редактор для построения доказательства, реализованный на Прологе. Он основан на логике высшего порядка, включающей индукцию. Система содержит определения основных конструкций и библиотеку теорем.

Тактики системы Oyster записаны на Прологе. Предикаты, описывающие свойства проводимого доказательства, доступны пользователю. Примитивные шаги доказательства выполняются с помощью применения только правил доказательства. Тактики можно комбинировать, используя специальные тактические конструкции. Встроенные механизмы Пролога (автоматическое сопоставление и поиск с возвратом) помогают автоматизировать поиск доказательства. Система Oyster используется как для доказательства теорем, так и для синтеза программ.

Система Clam позволила сделать из интерактивной системы Oyster систему автоматического доказательства теорем. Для каждой тактики из системы Oyster записывается спецификация для Clam. Эти спецификации называются

методами. Метод состоит из предусловия, входной формулы, выходной формулы и постусловия. Метод может быть применен к некоторой формуле, если она может быть сопоставлена с входной формулой и предусловие выполнено. Предусловия, сформулированные на языке металогики, описывают некоторые синтаксические свойства входной формулы. Используя предусловие и входную формулу, Clam определяет, можно ли применить тактику, до её непосредственного применения в Oyster. Выходная формула дает примерный шаблон получаемого в результате применения тактики результата, а постусловие описывает синтаксические свойства входной формулы. Метод представляет собой некоторый эвристический оператор, который описывает необходимые свойства тактики, не вдаваясь в детальные вычисления, требующие значительного времени.

План доказательства система Clam строит следующим образом. На каждом шаге по внешнему виду входной формулы с учётом предусловия система определяет, какой метод может быть применён. В результате его применения получается выходная формула и постусловие. Данная выходная формула является входной для следующего шага доказательства. Этот процесс продолжается до тех пор, пока все формулы не будут доказаны. Так как в некоторых ситуациях несколько методов могут оказаться применимыми, возникает дерево доказательства. На практике при использовании стратегии поиска вглубь по дереву доказательства результат получается за приемлемое время. По рассматриваемому пути доказательства для каждого метода Clam, выполняется соответствующая тактика Oyster.

Система автоматического синтеза программ **λClam** [Lacey et al., 2000] - расширение системы Clam. Система λClam была реализована на языке λProlog, который позволяет решать задачи на языке логики более высокого порядка, что упрощает, например, работу с кванторами. В то же время, используя λProlog в качестве металогики, система может решать и задачи в области логики предикатов первого порядка.

Ключевая тактика, применяемая в системах Clam и λClam при доказательстве теорем по индукции, – это применение волновых правил[1]. В соответствии с ней, при доказательстве шага индукции надо определить различия между гипотезой и заключением индукции, а затем так их преобразовывать, чтобы разница между ними уменьшилась. Для этого применяются только правила преобразования, которые уменьшают различия, таким образом отсекается часть правил, которые, скорее всего, не приведут доказательство к успешному завершению. Это делается с помощью специальных аннотаций на формулах: отмечаются части формул, которые могут быть изменены, и направление изменений. Метод может быть применен не только при выполнении шага индукции, но и при других преобразованиях одной формулы (гипотезы) к другой (цели). Цель преобразуется так, чтобы она содержала пример гипотезы как подвыражение. Тогда мы сможем воспользоваться истинностью гипотезы, что поможет доказать цель. Существенным достоинством такого подхода является значительное сокращение перебора при доказательстве, но использование волновых правил для синтеза осложняется тем, что обычно используемые правила преобразования формируются из определений участвующих в доказательстве объектов, а синтезируемая функция ещё не известна и для неё правила преобразования не могут быть выписаны.

Синтез логических программ с использованием волновых правил был реализован в системе **Periwinkle** [Kraan et al., 1996]. При синтезе спецификация рассматривается как терема существования объекта (программы), которую надо доказать. Доказываемые свойства заданы в спецификации, а сам объект – синтезируемая программа – обозначается метапеременной и получает свое значение в процессе проведения доказательства. В этой системе заранее задан набор различных схем индукции и схем соответствующих им программ. Задача состоит в выборе подходящей схемы на основе анализа рекурсии. Ограниченный набор схем программ ограничивает и набор задач, которые

[1] Эта тактика в англоязычной литературе называется rippling.

может решить система. В частности, система не смогла синтезировать функции сортировки списка и разделения списка на части.

Расширение идеи волновых правил было реализовано в системах **INKA** [Hutter and Sengler, 1996] и **NuPRL** [Pientka and Kreitz, 1999]. В системе INKA дополнительно к аннотации различий формул и направлению проведения преобразований задается еще «цвет» аннотации, что вводит новые ограничения. В системе NuPRL особое внимание уделено применению волновых правил для выражений, содержащих метапеременные, поиск значений которых проводится при проведении доказательства двух направлениях: от заключения индукции к гипотезе и, наоборот, от гипотезы в сторону заключения (обратный rippling).

В рассмотренных системах проводилось доказательство, при котором могла быть построена логическая программа. Логическая программа представляет собой некоторую формулировку задачи, а не детальный способ описания её решения. Тогда как на функциональном языке программирования (например, на Лиспе) описывается именно способ решения.

Мы будем использовать волновые правила при синтезе функциональных программ. Эта идея частично была рассмотрена в [Armando et al., 1999], но в этой работе синтез был лишь частично автоматизирован, за счёт того, что автоматически выполнялись некоторые этапы доказательств, а ведущая, направляющая роль – определение последовательности выполнения этих этапов – отводилась человеку. В данной диссертации предлагается подход, который объединяет преимущества нескольких методов синтеза: метода дедуктивных таблиц, позволяющего синтезировать все базовые конструкции функциональной программы и формировать структуру программы в процессе доказательства, а также преимущества применения волновых правил, которые позволяют существенно сократить перебор при доказательстве.

Постановка задачи

Целями диссертации являются:

- разработка нового метода дедуктивного синтеза функциональных программ, позволяющего расширить (по сравнению с известными методами) класс

программ, автоматическое построение которых можно осуществлять за приемлемое время;

- разработка и реализация на основе данного метода системы автоматического синтеза программ, которая осуществляет построение функциональных программ на языке Лисп по их формальным спецификациям и заданным аксиомам.

Основой нового метода является метод дедуктивных таблиц, который требуется дополнить рядом эвристик, в том числе волновыми правилами, чтобы сократить перебор вариантов при синтезе.

Содержание работы

Работа состоит из четырёх глав. В первой главе рассмотрены основы метода дедуктивных таблиц, который является базовым для разработанного в диссертации метода. Во второй главе приведены основные понятия теории применения волновых правил при построении доказательств по индукции. Глава 3 рассказывает о разработанном в диссертации методе синтеза программ. В главе 4 описана реализация системы АЛИСА[2] и результаты экспериментов с ней. Основные выводы работы сформулированы в заключении.

В приложениях содержится полное описание языка спецификаций, по которым осуществляется синтез в системе АЛИСА, информация о встроенных знаниях системы, описание используемого в ней внутреннего представления выражений и термов, примеры синтеза некоторых функций, а также приведён алгоритм унификации различий.

---

[2] Название образовано от Автоматический ЛИсп Синтезатор

Глава 1. Метод дедуктивных таблиц

В работах [Manna and Waldinger, 1980, 1992] предложен метод дедуктивных таблиц, предназначенный для синтеза функциональных программ по их формальным спецификациям. Основы этого метода рассмотрены в настоящей главе.

Функциональная программа строится на основе трёх структур управления: композиции функций, разветвления и рекурсии. Особенностью чисто функциональной программы является отсутствие присваиваний и переходов, что отличает её от процедурной программы, в основе которой лежит взятие значения переменной, совершение над ним действия и сохранение нового значения с помощью оператора присваивания. Функциональные программы отличаются и от логических программ, написанных, например, на Прологе [Братко, 1990], где обычно описываются только условия, которым должен удовлетворять результат, в функциональной же программе явно задается способ вычисления результата.

Функциональная программа представляет собой совокупность определений функций и ряда обращений к этим функциям. Вычисление каждого из этих обращений, в свою очередь, вызывает функции, входящие в определение вызванной функции и т. д. Каждый вызов функции возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается. Этот процесс повторяется до тех пор, пока запустившая процесс вычислений функция не вернёт результат.

В данной работе рассматривается синтез функций по отдельности. Одновременный синтез нескольких функций допустим лишь в случае, если описание всех их задано одной спецификацией. Поэтому в дальнейшем вместо термина синтез программы мы будем иногда использовать термин синтез функции или синтез функций (если задана спецификация сразу для нескольких функций).

Спецификация функции задается на языке логики предикатов первого порядка. В общем виде спецификация может быть представлена следующим образом:

```
<f(a)> <=  find z such that Q[a,z].
```

Эта запись означает, что значением искомой функция `f(a)` является некоторое `z`, удовлетворяющее условию `Q[a,z]`, где `a` – входной параметр функции `f`. Чтобы синтезировать функцию `f(a)`, заданная спецификация рассматривается как теорема существования объекта `z` (выхода программы), удовлетворяющего условию `Q[a,z]`, и ставится задача получить конструктивное доказательство этой теоремы, то есть такое доказательство, в котором мы получаем способ вычисления `z`. Затем этот способ вычисления непосредственно переводится в текст программы.

В методе дедуктивных таблиц доказательство записывается в специальной структуре, называемой дедуктивной таблицей. Таблица содержит несколько колонок: колонки для проведения доказательства и колонки для отображения соответствующих ему шагов синтеза. При этом правила вывода (правила преобразования таблицы) таковы, что каждому шагу доказательства соответствует определенный шаг построения функциональной программы. Так, например, в результате использования свойства добавления примера для строки таблицы в синтезируемой программе может появиться константа или вызов известной функции, в результате применения правила резолюции или замены эквивалентных термов возможно формирование условного терма, а применение правила индукции позволяет получить рекурсивное обращение к синтезируемой функции.

1.1 Основные понятия

Дадим определения основных терминов, которые будут использованы при описании метода дедуктивных таблиц.

Термы - это константы, переменные, обращения к функциям, аргументы которых – термы, а также “условные” конструкции вида `if F then s else t`, где `F` – некоторое логическое выражение, а `s` и `t` – термы. Обращение к функции мы будем записывать как в префиксной форме (например, `atom(x)`), так и в инфиксной (`a+1`). Примеры термов: `123, x, a+1, if t=1 then 0 else 1`.

Атомарное выражение - это предикат от некоторых термов, то есть конструкция, принимающая значение ”истина” (true) или ”ложь” (false). Возможны два вида записи предикатов: префиксная, например: `elem(a,tail(x))`, и инфиксная, например: `head(L)=2`. Символ =, как и знаки других операций сравнения (>, < ,>= ,<=, <>), является предикатным символом.

Логические выражения (или просто **выражения)** – это `true, false`, атомарные выражения, а также конструкции, построенные из выражений и логических операций ¬ (`not`), ∧ (`and`), ∨ (`or`) и кванторов ∀`x` и ∃`x`, например: ∀`x (elem(x,L)` ∧ `atom(x))`. Переменная `x`, записанная непосредственно после квантора `Q` в выражении `Qx(A)`, называется **связанной** в выражении `A` (области действия квантора). В выражение могут входить и свободные (не связанные кванторами) переменные. Переменная с одним и тем же именем может оказаться и связанной, и свободной в одном выражении, например, переменная `x` входит в выражение `atom(x)` ∧ ∀`x (elem(a,tail(x))=0)` как свободная и как связанная. По смыслу это две различные переменные, поэтому в таких ситуациях будем давать им разные имена: `atom(x)` ∧ ∀`y (elem(a,tail(y))=0`).

При вычислении значения логического выражения наивысший приоритет имеет операция ¬ (`not`), следующей вычисляется ∧ (`and`), затем ∨ (`or`), далее вычисляются функции и предикаты, кроме операций арифметических отношений, которые вычисляются последними. Для изменения порядка вычислений используются круглые скобки.

Выражения могут быть записаны с использованием условных логических связок if-then-else и if-then, то есть в виде if A then B else C или if A then B, где A, B, C – выражения (а не термы, в отличие от конструкторов if-then-else и if-then для термов). Эти условные выражения эквивалентны $A \wedge B \vee \neg A \wedge C$ и $\neg A \vee B$ соответственно.

Общим термином **предложение** будем называть терм или выражение.

Подстановкой называется множество вида $\{x_1 \leftarrow t_1, x_2 \leftarrow t_2, \ldots, x_n \leftarrow t_n\}$, где x_i – различные переменные, а t_i – термы, причем переменная x_i не входит в соответствующий ей терм t_i. После применения подстановки $\lambda = \{x_1 \leftarrow t_1, x_2 \leftarrow t_2, \ldots, x_n \leftarrow t_n\}$ к выражению A мы получим выражение $A\lambda$, в котором все свободные переменные x_i, входящие в A, будут заменены на соответствующие им термы t_i. Например, выполнив подстановку $\{x \leftarrow f(a), y \leftarrow b\}$ в $g(x)+h(y)=0$, мы получим $g(f(a))+h(b)=0$.

Для проведения доказательства методом дедуктивных таблиц используется специальная структура, называемая дедуктивной таблицей. Она выглядит следующим образом:

Assertions	Goals	Output $_1$	…	Output $_k$
A_1				
A_2				
…				
A_n				
	G_1	z_1	…	z_k
	G_2	u_1		u_k
…				
	G_m	t_1	…	t_k

Выходные колонки

Каждая строка такой таблицы содержит выражение, являющееся либо утверждением (assertion), которое записывается в левой колонке, либо целью (goal), записываемой во второй колонке. Одна и та же строка не может одновременно содержать утверждение и цель. Каждая строка таблицы может иметь выходные термы для синтезируемых функций. Количество выходных

колонок соответствует количеству функций, описываемых спецификацией. Таблица является наглядной формой записи логического выражения

$\mathtt{if}\ (A_1 \wedge A_2 \wedge \ldots \wedge A_n)\ \mathtt{then}\ (G_1 \vee G_2 \vee \ldots \vee G_m)$,

где A_1, …, A_n – известные в системе аксиомы, а G_1, …, G_m – доказываемые цели.

Из определения следует, что колонки утверждений и целей выделяются именно для наглядности, при проведении доказательства цель произвольной строки может быть перенесена в колонку утверждений с добавлением отрицания, и любое утверждение может быть перенесено в колонку целей с добавлением отрицания.

Выходные колонки служат для получения программы в процессе доказательства. Будем для краткости рассматривать таблицу с одной выходной колонкой, но все приведенные для неё утверждения могут быть аналогичным образом применены и к остальным выходным колонкам, если их в таблице несколько.

Терм t, не содержащий свободных переменных, **удовлетворяет строке таблицы**

A		s	или		G	s

если для некоторой подстановки λ выполняются два следующих условия:

- Утверждение $A\lambda$ не содержит свободных переменных и является ложным (или, соответственно, $G\lambda$ не содержит свободных переменных и является истинным),
- Если строка имеет выход s, то терм $s\lambda$ не содержит свободных переменных и равен t.

Не содержащий свободных переменных терм **удовлетворяет таблице**, если он удовлетворяет какой-нибудь строке этой таблицы. Две таблицы являются **эквивалентными**, если множества термов, которые им удовлетворяют, совпадают.

Рассмотрим свойства дедуктивных таблиц, позволяющие проводить их эквивалентные преобразования.

1.2 Свойства дедуктивных таблиц

Приведем основные свойства дедуктивных таблиц, следующие из их определения.

1. *Двойственность.*

Для любых выражений A и G и произвольного терма s справедливо:

A		s

эквивалентна

	¬A	s

	G	s

эквивалентна

¬G		s

То есть перенос выражения из одной колонки в другую с отрицанием, с сохранением того же выходного терма, приводит к получению эквивалентной таблицы.

2. *Переименование свободных переменных.*

Перестановкой называется частный случай подстановки, осуществляющей замену имён переменных, причём разным переменным соответствуют разные новые имена переменных, а одинаковым – одинаковые.

Для любых выражений A и G, произвольного терма s и перестановки π справедливо утверждение:

A		s

эквивалентна

Aπ		sπ

	G	s

эквивалентна

	Gπ	sπ

То есть допустимо переименование свободных переменных в любой строке, поэтому можно считать, что в разных строках используются разные свободные переменные.

3. *Добавление примера.* Для любой строки, содержащей утверждение A или цель G и имеющей выход s, добавление в таблицу примера (частного случая) этой строки, для некоторой подстановки λ (то есть строки с выходом sλ, содержащей утверждение Aλ или цель Gλ соответственно), порождает эквивалентную таблицу:

A		s

эквивалентна

A		s
Aλ		s

	G	s

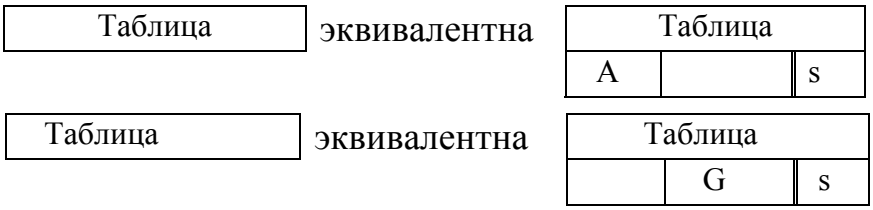
эквивалентна

	G	s
	Gλ	sλ

Заметим, что, в отличие от свойств двойственности и переименования свободных переменных, при добавлении примера исходная строка тоже должна быть сохранена в преобразованной таблице. Данное свойство позволяет добавлять в таблицу строки, содержащие более простые выражения, позволяет получить в выходной колонке константу или терм, содержащий обращение к функции.

4. Добавление и удаление истинного утверждения и ложной цели

Пусть `A` – тождественно истинное выражение, записанное в колонке утверждений (или G – тождественно ложное выражение, записанное в колонке целей). Тогда таблица, полученная добавлением строки, содержащей утверждение `A` (цель `G`) и произвольный выходной терм `s` эквивалентна исходной.



Чаще всего это свойство используется для удаления строк, содержащих тождественно истинное (`true`) утверждение или тождественно ложную (`false`) цель, являющихся бесполезными для доказательства.

5. Добавление/удаление выходной переменной

Строка таблицы, не имеющая выходной переменной, может быть заменена на такую же строку, но имеющую в качестве выхода любую переменную, которая не является свободной в утверждении (или цели) этой строки. Полученная при этом таблица будет эквивалентна исходной. Аналогично, выходная переменная, не являющаяся свободной переменной в рассматриваемой строке, может быть удалена из рассматриваемой строки. На практике это свойство используется именно для удаления из выходной колонки переменной, не содержащейся в утверждении или цели рассматриваемой строки.

6. Тождественные преобразования

Любое утверждение, цель или выходной терм произвольной строки таблицы могут быть заменены согласно правилам тождественных преобразований. При

этом в таблицу новые строки не добавляются, а заменяется некоторое предложение в уже присутствующей строке. Обычно такое преобразование нужно для упрощения предложений. В качестве примера, приведём следующие правила тождественных преобразований:

1. выражение `A ∧ A` может быть преобразовано в `A`,

2. выражение `A ∨ true` может быть преобразовано в `true`.

Полный список упрощений зависит от используемого множества логических операций, предикатов и функций, некоторые свойства которых и выражены в упрощениях. Упрощение обычно проводится перед добавлением любой новой строки в дедуктивную таблицу.

1.3 Дедуктивные правила

Дедуктивные правила позволяют добавлять новые строки в таблицу. Таблица, полученная в результате применения дедуктивных правил, может быть не эквивалентна исходной, но множества удовлетворяющих им вычислимых термов (это термы, которые известно как вычислить, используя уже определенные константы, предикаты и функции) совпадают. Такие таблицы описывают один и тот же класс программ. Обоснования применения правил приведены в работе [Manna and Waldinger, 1992].

1. Правила расщепления.

Эти правила предоставляют возможность работать с более простыми выражениями, хотя любое доказательство, построенное с использованием правил расщепления, может быть проведено и без них. При расщеплении строка с исходным выражением сохраняется в таблице.

Расщепление AND-утверждения и OR-цели. Утверждение, содержащее конъюнкцию (цель, содержащая дизъюнкцию), может быть расщеплено на составные части. Выходной терм полученных строк дублирует выходной терм исходной строки.

Строка

$A_1 \wedge A_2$		s

может быть преобразована в

$A_1 \wedge A_2$		s
A_1		s
A_2		s

Строка

	$G_1 \vee G_2$	s

может быть преобразована в

	$G_1 \vee G_2$	s
	G_1	s
	G_2	s

Расщепление IF-цели. С учётом того, что `if A then G` эквивалентно `¬ A ∨ B`, строка

	if A then G	s

может быть преобразована в

	if A then G	s
A		s
	G	s

Согласно общему правилу, выходной терм дублируется в добавляемых строках. Однако на практике часто возникают ситуации, когда в добавляемых строках выходная переменная не встречается в утверждении (или цели) этих строк. Например,

	if a>0 then z=f(a)	z
a>0		z
	z=f(a)	z

В этом случае мы используем свойство удаления выходной переменной для второй строки:

	if a>0 then z=f(a)	z
a>0		
	z=f(a)	z

Во второй строке полученной таблицы записано утверждение, о том, что `a>0`, оно может быть использовано как известный факт. Также в таблице указано, что если будет доказана цель `z=f(a)`, то в качестве результата будет выдано z.

Для описания остальных дедуктивных правил нам потребуется понятие **унификации**. Подстановка λ называется унификатором предложений A и B, если $A\lambda=B\lambda$. Унификатор λ называется **наиболее общим унификатором** для предложений A и B тогда и только тогда, когда для любого другого унификатора ϕ существует такая подстановка μ, что $\lambda=\mu\circ\phi$, где $\circ$ – обозначение операции композиции подстановок. Операция композиции определяется следующим образом: подстановка $\mu\circ\phi$ получается из $\mu=\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ и $\phi=\{y_1 \leftarrow u_1, \ldots, y_n \leftarrow u_m\}$ как подстановка $\{x_1 \leftarrow t_1\phi, \ldots, x_n \leftarrow t_n\phi, y_1 \leftarrow u_1, \ldots,$

$y_n \leftarrow u_m\}$, в которой вычеркиваются $x_i \leftarrow t_i\phi$, если $x_i = t_i\phi$, и $y_i \leftarrow u_i$, если $y_i = x_j$ при некотором j, $1 \le j \le n$.

Для унификации двух предложений A и B существует специальный алгоритм (описанный, например, в работе [Чень и Ли, 1983]), который либо находит наибольший общий унификатор данных выражений, либо выдает ответ о том, что выражения унифицировать невозможно.

Рассмотрим в качестве примера унификацию термов P(NIL) и P(x), где P – некоторый предикат, NIL – константа, x – переменная. Идея алгоритма состоит в следующем: предложения P(NIL) и P(x) просматриваются слева направо до нахождения первого несовпадающего символа. В данном случае это символы N и x. Начиная с этого символа, выписывается выражение (терм) из P(NIL), который не соответствует выражению (терму) из P(x). Эти термы (NIL и x) называются различием. Задача состоит в том, чтобы ликвидировать различия, если это возможно, выполнив некоторые подстановки. Если одно найденное различие состоит из переменной (в данном случае x) и некоторого терма (NIL), то после замены переменной на этот терм различие будет ликвидировано. Полученная подстановка {x←NIL} выполняется в обоих выражениях, и поиск различий во вновь полученных выражениях начинается сначала. В рассматриваемом примере после выполнения подстановки выражения становятся одинаковыми, найден наибольший общий унификатор {x←NIL}, алгоритм унификации останавливается. Если в процессе поиска нашлось различие, в которое не входит переменная, то процесс унификации тоже останавливается – унификация таких выражений невозможна. Формальная запись алгоритма унификации приведена в приложении 2.

2. Правило резолюции. Это правило позволяет получить в выходной колонке условный терм, то есть позволяет ввести разветвление в синтезируемую функцию.

*Правило резолюции для целей (GG-резолюция).*В таблицу, содержащую строки G1 и G2, может быть добавлена строка G:

G1:		$G_1[P]$	s
G2:		$G_2[P']$	t
G:		$G_1\lambda[false] \wedge G_2\lambda[true]$	if $P\lambda$ then $t\lambda$ else $s\lambda$

Здесь:

G_1, G_2 – выражения, не имеющие общих свободных переменных (если это не так, можно, используя свойство дедуктивных таблиц, переименовать соответствующие переменные);

P, P' – их подвыражения, не содержащие кванторов, такие что P и P' могут быть унифицированы некоторой подстановкой λ: $P\lambda=P'\lambda$.

Заменив все вхождения $P\lambda$ в $G_1\lambda$ на $false$, получаем $G_1\lambda[false]$, а все вхождения $P'\lambda$ в $G_2\lambda$ на $true$, получаем $G_2\lambda[true]$. В таблицу добавляется новая цель, содержащая конъюнкцию $G_1\lambda[false] \wedge G_2\lambda[true]$, с указанным условным выходом.

Используя аналогичные обозначения, запишем ещё три варианта правила резолюции, полученные из GG-резолюции с помощью свойства двойственности таблицы.

AG-резолюция:

A1:	$A_1[P]$		s
G2:		$G_2[P']$	t
G:		$\neg A_1\lambda[false] \wedge G_2\lambda[true]$	if $P\lambda$ then $t\lambda$ else $s\lambda$

AA-резолюция:

A1:	$A_1[P]$		s
A2:	$A_2[P']$		t
A:	$A_1\lambda[false] \vee A_2\lambda[true]$		if $P\lambda$ then $t\lambda$ else $s\lambda$

GA-резолюция:

G1:		$G_1[P]$	s
A2:	$A_2[P']$		t
G:		$G_1\lambda[false] \wedge \neg A_2\lambda[true]$	if $P\lambda$ then $t\lambda$ else $s\lambda$

Замечания о выходном терме :

1. Если $s\lambda=t\lambda$, то условный терм можно упростить следующим образом:

$$\text{if } P\lambda \text{ then } t\lambda \text{ else } t\lambda \equiv t\lambda.$$

2. Если одна из строк (например, G2) не имеет выходного терма, то вместо условного выражения в качестве выхода новой строки записывается sλ – выходной терм второй строки G1.

3. Если у обеих строк нет выходных термов, то и у полученной по правилу резолюции строки его не будет.

Для остальных дедуктивных правил мы приведем только один вариант, применяющийся к целям. Аналогичные правила могут быть сформулированы для пары утверждений, для утверждения и цели, используя свойство двойственности таблицы.

3. Правило замены эквивалентных термов.

В таблицу, содержащую строки G1 и G2, может быть добавлена строка G:

G1:		$G_1[L=R]$	s
G2:		$G_2<L'>$	t
G:		$G_1\lambda[false] \wedge G_2\lambda<R\lambda>$	if $(L=R)\lambda$ then $t\lambda$ else $s\lambda$

Здесь:

G_1, G_2 – выражения, не имеющие общих свободных переменных (если это не так, можно, используя свойство дедуктивных таблиц, переименовать соответствующие переменные);

L=R – подвыражение цели G1, а L′ – терм, входящий в G2, который может быть унифицирован с термом L некоторой подстановкой λ: Lλ=L'λ. Выражение L=R и терм L′ не содержат кванторов.

Заменив все вхождения (L=R)λ в $G_1\lambda$ на false, получаем $G_1\lambda$[false], и заменив некоторые (не обязательно все) вхождения L′λ в $G_2\lambda$ на Rλ, получаем $G_2\lambda<R\lambda>$. Так как заменяются не все, а только некоторые вхождения L'λ, мы используем при записи не квадратные скобки [], а угловые <>. В таблицу добавляется новая строка с указанным условным выходом, содержащая цель $G_1\lambda[false] \wedge G_2\lambda<R\lambda>$. Относительно выходного терма верны те же замечания, что и в правиле резолюции.

Данное правило также позволяет ввести в синтезируемую программу условную конструкцию.

4. Правило замены эквивалентных выражений.

Это правило аналогично правилу замены эквивалентных термов, только в нём `L, R, L'` рассматриваются не как термы, а как выражения, и вместо равенства термов (=), рассматривается эквивалентность логических выражений (≡).

5. Индукционное правило.

Если G – исходная цель (содержащая выражение из спецификации), то в таблицу может быть добавлена гипотеза индукции - следующее утверждение A:

	Assertions	Goals	f(a)
G:		Q[a,z]	z
A:	if $x <_{wf} a$ then Q[x,f(x)]		

Здесь:

`Q[a,z]` – исходная цель (спецификация), `z` – её выходная переменная, a – входной параметр синтезируемой функции,

`f` – имя синтезируемой функции,

$<_{wf}$ - wf- отношение в рассматриваемой теории.

Отношение $<_{wf}$ является wf-отношением (well-founded), если в рассматриваемой теории нельзя построить бесконечно убывающую последовательность, то есть такую последовательность $x_1, x_2, x_3, \ldots$, что $x_2 <_{wf} x_1$, $x_3 <_{wf} x_2, \ldots$ В качестве примера wf-отношения можно привести отношение < для натуральных чисел.

Гипотеза индукции строится только для исходной цели таблицы, она фактически является предположением, что заданное в спецификации свойство выполнено для аргумента, меньшего, чем исходный. В результате применения правила индукции в строке таблицы появляется терм, содержащий обращение к синтезируемой функции. Таким образом, это правило позволяет получить рекурсивную конструкцию.

6. Правило удаления кванторов.

Выражения, содержащие кванторы, должны быть специальным образом переписаны перед применением дедуктивных правил, так как дедуктивные

правила позволяют проводить рассуждения только о выражениях, не содержащих кванторов.

В колонке утверждений дедуктивной таблицы все свободные переменные по определению считаются связанными кванторами всеобщности, тогда как в колонке целей все свободные переменные считаются связанными кванторами существования. Дело в том, что колонка утверждений содержит аксиомы, заданные для некоторой теории (например, для целых чисел), то есть это утверждения, описывающие общие свойства объектов этой теории. Дополнительные ограничения задаются в виде явных кванторов. Колонка целей содержит логическое выражение, истинность которого надо доказать для *некоторого* объекта, входящего в спецификацию. Построение конструктивного доказательства существования этого объекта (доказательства цели) и является процессом синтеза требуемой функции. Поэтому в колонке целей свободные переменные подразумеваются с квантором существования. Аналогично утверждениям, дополнительные ограничения на переменные целей могут быть заданы в виде явных кванторов.

Задача проводимых преобразований – избавиться от явных кванторов в таблице. Для этого применяем алгоритм, описанный в [Чень и Ли, 1983].

На первом этапе все различные связанные переменные получают уникальные имена. Например, выражение

$\forall x$ (elem(x,c)) $\wedge$ $\exists x$ (not(emptylist(x)))

будет преобразовано в

$\forall a_1$ (elem(c,a_1)) $\wedge$ $\exists b_1$ (not(emptylist(b_1)).

Затем кванторы выносятся из внутренних выражений и заданное утверждение или цель принимает вид:

[Кванторная приставка] Логическое выражение.

Пример: $\forall a_1$ $\exists b_1$(elem(c,a_1)) $\wedge$ (not(emptylist(b_1))

После этого для удаления кванторов $\exists$ в колонке утверждений проводятся следующие преобразования:

1. Если кванторная приставка утверждения имеет вид

$\forall a_1, \ldots, \forall a_n$, [Кванторная приставка 1],

то она может быть преобразована к виду

[Кванторная приставка 1],

то есть стоящие в начале кванторы всеобщности могут быть удалены.

2. Если утверждение имеет вид

$\exists a_1$ [Кванторная приставка] $A(a_1, x_1, \ldots, x_n)$,

где $x_1, \ldots, x_n$ – все свободные переменные, входящие в A, то преобразуем его в утверждение

[Кванторная приставка] $A(f(x_1, \ldots, x_n), x_1, \ldots, x_n)$,

где f – новое имя функции, не совпадающее ни с одним из присутствующих в рассматриваемой дедуктивной таблице; f - называется скулемовской функцией. Существенной особенностью работы со скулемовскими функциями является то, что правила вычисления этих функций не известны. Поэтому не допускается появления скулемовских функций в выходной колонке, где происходит формирование вычислимой функции.

Указанные преобразования проводятся для утверждения до тех пор, пока не будут удалены все содержащиеся в нем кванторы.

В колонке целей удаление кванторов всеобщности происходит аналогичным образом:

1. Если кванторная приставка цели имеет вид

$\exists a_1, \ldots, \exists a_n$, [Кванторная приставка 1],

то она может быть преобразована к виду

[Кванторная приставка 1],

то есть стоящие в начале кванторы существования могут быть удалены.

2. Если цель имеет вид

$\forall a_1$ [Кванторная приставка] $G(a_1, x_1, \ldots, x_n)$,

где $x_1, \ldots, x_n$ – все свободные переменные, входящие в G, то преобразуем ее в цель

[Кванторная приставка] $G(f(x_1, \ldots, x_n), x_1, \ldots, x_n)$,

где f – новая скулемовская функция.

Приведенные выше преобразования применяются к цели до тех пор, пока из неё не будут удалены все кванторы. Затем в таблицу может быть добавлена новая строка, содержащая полученное утверждение (или цель) без кванторов, в качестве её выхода берется тот же выход, что и в исходной строке. В качестве примера рассмотрим преобразование цели

G:		∀x, ∀y (x+y+g(z)=0)	z

Цель содержит переменные x, y и z: x и y связаны кванторами всеобщности, z – свободная переменная. Чтобы избавиться от кванторов мы можем заменить x на скулемовскую функцию fx с одним параметром z, а затем заменить y на fy(z). Полученная цель

G1:		fx(z)+fy(z)+g(z)=0	z

будет добавлена в таблицу.

7. Правило замены аргументов в отношениях.

Это правило предложено и обосновано в работе [Traugott, 1989]. Под отношением в данном случае понимается любой предикат, имеющий два аргумента, записанный в префиксной или инфиксной форме. Пусть rel - некоторое отношение, связывающее термы t_1 и t_2 – rel(t_1,t_2) или t_1 rel t_2 . При такой записи будем считать, что терм t_1 "меньше" t_2 согласно отношению rel. Терм t имеет положительную полярность в выражении P(t) относительно rel (обозначается +rel), если заменив такой терм на "больший" согласно отношению rel, мы получим логическое следствие исходного выражения $P(t_1) \rightarrow P(t_2)$. Полярность терма t называется отрицательной (обозначается -rel), если заменив такой терм на "меньший" согласно отношению rel, мы получим логическое следствие исходного выражения $P(t_1) \rightarrow P(t_2)$. Правило замены аргументов в отношениях позволяет добавить в дедуктивную таблицу со строками G1 и G2 строку G:

G1:		G_1[L rel R]	s
G2:		G_2<L'{-rel}>	t
G:		$G_1\lambda$[false] $\wedge$ $G_2\lambda$<Rλ>	if (L rel R)λ then tλ else sλ

Здесь:

G_1, G_2 – выражения, не имеющие общих свободных переменных (если это не так, можно, используя свойство дедуктивных таблиц, переименовать соответствующие переменные);

L rel R – подвыражение цели G1, rel – некоторое отношение (также оно может быть записано и в префиксной форме), а L′ – терм, входящий в G2, имеющий отрицательную полярность относительно rel. Терм L′ должен быть унифицируемым с термом L некоторой подстановкой λ: Lλ=L'λ.

Заменив все вхождения (L rel R)λ в G_1λ на false, получаем G_1λ[false], заменив некоторые вхождения L′λ в G_2λ (имеющие отрицательную полярность согласно rel) на Rλ, получаем G_2λ<Rλ>. Так как заменяются не все, а только некоторые вхождения L'λ, мы используем при записи не квадратные скобки [], а угловые <>. В таблицу добавляется новая строка с указанным условным выходом, содержащая цель G_1λ[false] ∧ G_2λ<Rλ>. Относительно выходного терма верны те же замечания, что и в правиле резолюции.

Другой вариант применения этого же правила состоит в том, что в таблицу, содержащую строки G1’ и G2’, может быть добавлена строка G’:

G1’:		G_1[L rel R]	s
G2’:		G_2<R'{+rel}>	t
G’:		G_1λ[false] ∧ G_2λ<Lλ>	if (L rel R)λ then tλ else sλ

Правило замены аргументов в отношениях является обобщением правила замены эквивалентных термов, которое можно рассмотреть как правило для отношения равенства (=).

1.4 Порождение программы

Пусть задана спецификация:

<f(a)> <= find z such that Q[a,z] (1.1)

Предположим, что в выражении Q[a,z] нет свободных переменных, кроме z. Все входные переменные, в данном случае это одна переменная a,

объявляются константами (то есть в процессе преобразования таблиц их, в отличие от переменных, нельзя ни на что заменять, кроме специальной замены, предусмотренной при применении индукционного правила); функциональный символ `f` включается в множество известных для синтеза имён функций, и выражение `Q[a,z]` заносится в таблицу как цель. В таблице могут также присутствовать известные аксиомы $A_1, \ldots, A_n$.

Assertions	Goals	f(a)
A1		
…		
An		
	Q[a,z]	z

К этой таблице применяются допустимые дедуктивные правила и эквивалентные преобразования. Согласно теореме, доказанной в [Manna and Waldinger, 1992], если любой терм `t(a)` без свободных переменных удовлетворяет такой таблице, то он дает алгоритм получения результата – значения функции `f(a)`, удовлетворяющего спецификации (1.1).

Процесс преобразования таблицы продолжается до тех пор, пока не будет получена строка

false		t

или

	true	t

Терм `t` без свободных переменных будет удовлетворять любой таблице, содержащей одну из таких строк, и поэтому будет представлять собой искомую программу. На этом процесс порождения завершается, а текст программы строится по выходному терму `t`.

В качестве простого примера рассмотрим синтез функции `fact(n)`, возвращающей в качестве результата факториал натурального числа n. Спецификация функции fact(n) может быть записана следующим образом:

```
<fact(n)> <= find <z> such that
                    if n=1 then z=1 else z=n*fact(n-1).
```

Для синтеза этой функции дополнительные аксиомы не потребуются. Записываем в таблицу исходную цель для доказательства:

	Assertion	Goals	fact(n)
1		if n=1 then z=1 else z=n*fact(n-1)	z

Условное предложение преобразуется согласно тождеству

if A then B else C≡ (A∧B)∨(¬A ∧C):

2		((n=1) ∧ (z=1)) ∨ (¬(n=1) ∧ (z=n*fact(n-1)))	z

Применяем правило расщепления OR-цели. К исходной таблице добавляются строки:

3		(n=1) ∧ (z=1)	z
4		¬(n=1) ∧ (z=n*fact(n-1))	z

Используя свойство дедуктивной таблицы, добавляем пример строки 3, выполнив подстановку {z←1}, и пример строки 4, выполнив подстановку {z←n*fact(n-1)}:

5		(n=1)	1
6		¬(n=1)	n*fact(n-1))

К строкам 6 и 5 может быть применено правило резолюции. Их общее подвыражение `(n=1)`, унификатор – пустая подстановка. Заменяем выражение `(n-1)` в 6 строке на false, а в 5 строке на true. Конъюнкцию полученных выражений упрощаем согласно правилу `¬(false) ∧ true ≡ true`. В таблицу добавляется новая строка 7 с условным выходом:

7		true	if n=1 then 1 else n*fact(n-1))

Строка 7 содержит тождественно истинную цель, процесс вывода завершен. Полученный текст функции содержится в выходной колонке строки (7):

```
fact(n)= if n=1 then 1 else n*fact(n-1))
```

В рассмотренном примере алгоритм вычисления результата функции был фактически задан в спецификации, поэтому синтез получился достаточно простым. Примеры синтеза более сложных функций приведены в приложениях 4, 5, 6.

1.5 Проблема комбинаторного взрыва.

Дедуктивная таблица – удобная и наглядная структура для проведения доказательства и, одновременно, построения программы. В описании метода заданы правила преобразования таблицы, с помощью которых проводится доказательство одновременно с синтезом программы. Однако вопрос о

последовательности проведения преобразований таблицы остается открытым. На каждом шаге доказательства обычно находится несколько правил, которые могут быть применены, или же существует несколько вариантов применения одного и того же правила. При проведении доказательства вручную или автоматизированно вопрос о том, какое правило следует применить и каким образом, решается человеком. Вручную с помощью метода дедуктивных таблиц были синтезированы, например, несколько алгоритмов сортировки [Traugott, 1989] и алгоритм унификации [Nardi, 1989],

Однако при использовании метода дедуктивных таблиц для автоматического синтеза возникает огромный перебор вариантов. Например, при автоматическом синтезе функции `mod(i,j)`, вычисляющей остаток от деления целого числа `i` на натуральное `j`, даже при задании минимального набора аксиом в системе АЛИСА было рассмотрено около 30300 вариантов применения дедуктивных правил. При проведении же синтеза вручную оказалось достаточным сделать 20 шагов, на каждом из которых применялось одно из дедуктивных правил и проводилось упрощение полученных выражений. Таким образом, чтобы использование метода дедуктивных таблиц для автоматического синтеза стало возможным, необходимы дополнительные эвристики, сокращающие перебор. Особенно это важно при использовании в доказательстве индукционного правила, то есть при построении рекурсивной ветви функции, так как на практике на эту часть доказательства приходится более 80% перебора. Для сокращения перебора при проведении доказательства по индукции в системе АЛИСА были использованы волновые правила. При их использовании, например, при автоматическом синтезе функции `mod(i,j)` рассматривалось уже менее 100 вариантов применения правил.

Глава 2. Доказательство с использованием волновых правил

В данной главе будет рассмотрено применение для доказательства правил переписывания с дополнительными ограничениями – так называемых волновых правил, которые используются при построении доказательств методом математической индукции.

При проведении доказательства свойств объектов, событий или процедур, содержащих повторение, обычно нужно использовать доказательство методом математической индукции. В качестве примеров таких сущностей могут выступать натуральные числа, списки, деревья, программы, содержащие операторы цикла или рекурсивные вызовы, и т.д. Доказательство по индукции основывается на следующей общей схеме: сначала доказывается базовый случай, а затем – шаг индукции. На шаге индукции делается некоторое предположение (гипотеза индукции), исходя из истинности которого мы пытаемся доказать заключение шага индукции.

В качестве примера рассмотрим правило индукции для неотрицательных целых чисел (индукцию Пеано):

$$\frac{P(0),\quad \forall n{:}nat\ (P(n) \rightarrow P(n+1))}{\forall n{:}nat\ (P(n))}, \qquad (2.1)$$

Здесь `n` – переменная, по которой ведётся индукция, `n:nat` обозначает, что `n` – натуральное число, `P(0)` – базовый случай, то есть утверждение о том, что некоторая формула `P` истинна для числа `0`, `P(n)` – гипотеза индукции, `P(n+1)` – заключение. Запись (2.1) означает следующее: если доказан базовый случай и доказано, что из гипотезы индукции следует заключение (эти факты записаны в (2.1) над чертой), то рассматриваемая формула P истинна для всех натуральных чисел (этот факт указан под чертой).

Фактически при доказательстве шага индукции надо установить истинность формулы, содержащей логическое следствие:

гипотеза индукции → заключение индукции.

Это возможно сделать, применяя некоторые правила преобразования. Обычно на каждом шаге доказательства можно применить несколько правил, поэтому пространство поиска быстро разрастается. Возникает вопрос о том, как сократить перебор вариантов.

Для этого выбор правила вывода нужно сделать не произвольным, а направленным, учитывая, например, факт, что очень часто гипотеза и заключение индукции синтаксически похожи. Нужно применять только те правила, которые уменьшают различия между гипотезой и заключением. В общем случае применение только этих правил может не привести к успеху доказательства, но на практике обычно такой подход оказывается успешным и, что очень важно, позволяет существенно сократить перебор. В случае, если заключение и гипотеза индукции не похожи, данная эвристика оказывается неприменимой, но на практике такие случаи встречаются не очень часто.

Синтаксические различия гипотезы и заключения индукции фиксируются с помощью специального аннотирования. Аннотации отмечают, какие части гипотезы и заключения совпадают и должны быть сохранены, а какие различаются и должны быть преобразованы, и указывают направление преобразований.

Эта эвристика применима не только в доказательствах по индукции, но и везде, где одна формула (цель) должна быть доказана с помощью другой формулы (предположения), и эти формулы синтаксически похожи. Цель преобразуется к такому виду, чтобы она содержала пример предположения, который затем можно заменить на истину и тем самым упростить дальнейшее доказательство цели.

Правила преобразования, содержащие аннотации, называются волновыми правилами[3]. В качестве обоснования терминологии её авторы, ученые из Эдинбургского университета, проводят в [Bundy et al., 1993, 2005] аналогию

[3] Эта эвристика была названа авторами rippling, что означает распространение волн, волнение на поверхности воды.

между применением эвристики и изменением отражения некоторого объекта в озере. Пусть четкость отражения объекта нарушена камнем, брошенным в воду. Гипотеза индукции сравнивается с самим рассматриваемым объектом, а заключение – с его отражением. Волны на поверхности воды расходятся кругами от центра возмущения, и точность отражения восстанавливается. Эвристику можно представить как перемещение волн (различий) туда, где они не задевают отражение объекта. Так, при преобразовании формул различия между целью и исходными данными должны перемещаться так, чтобы сделать эти две формулы максимально похожими.

Данная эвристика основывается на применении правил переписывания [Dershowitz and Jouannaud, 1990], [Baader and Nipkow, 1999].

2.1 Системы переписывания

В математике любое доказательство можно рассматривать как последовательность шагов, на каждом из которых некоторый терм доказываемого соотношения заменяется другим термом. Эта замена проводится согласно заранее заданным правилам переписывания. Правило переписывания представляет собой соотношение вида α ==> φ, где α и φ - термы. Множество таких правил называется системой переписывания. Отметим, что здесь и далее мы будем использовать символ ==> для обозначения операции переписывания, а символом → будем обозначать операцию логического следствия, строчными латинскими буквами будут обозначаться константы и термы, а прописными – свободные переменные.

Правила переписывания применяются следующим образом. Чтобы применить правило вида α ==> φ к некоторому выражению A, в выражении A необходимо найти внутренний терм ψ, который может быть унифицирован с α (левой частью правила) с помощью некоторой подстановки λ. Тогда к выражению A применяется подстановка λ, и все или только некоторые вхождения терма $\psi\lambda$ заменяются на $\varphi\lambda$ - на правую часть правила переписывания, к которой

применена подстановка λ. Аналогичным образом могут быть определены правила переписывания и для логических выражений. В этом случае обе части правила переписывания α ==> φ являются выражениями.

Правила переписывания формируются из определений объектов, о которых ведется рассуждение, и из известных соотношений между этими объектами. Например, функцию s(N), возвращающую для каждого неотрицательного целого числа N следующее за ним целое число, можно определить следующим образом:

```
s(0)=1,
s(N)=N+1.
```

Из этого определения могут быть сформированы следующие правила переписывания:

```
s(0)==>1,
s(N)==>N+1,
1==>s(0),
N+1==>s(N).
```

Применение систем переписывания позволяет определить, является ли соотношение логическим следствием заданного набора аксиом. В качестве примера рассмотрим систему переписывания, которая позволяет получить остаток от деления на 2 суммы из 0 и 1 без применения операций деления.

Действия, которые мы можем совершать с заданным выражением, следующие:

```
0+0 ==>0,                    (2.2)
1+0 ==>1,                    (2.3)
0+1 ==>1,                    (2.4)
1+1==>0                      (2.5)
```

Запись этих действий представляет собой систему переписывания, её правила могут быть применены к заданному выражению в любом порядке.

Рассмотрим применение правил (2.2) – (2.5) к выражению

```
1+1+0+1.
```

Применив правило (2.3), мы преобразуем данное выражение к виду

`1+1+1`.

К полученному выражению можно применить только правило (2.5):

`0+1`.

Теперь применимо правило (2.4), которое и дает в результате `1`. Больше ни одно из правил применить нельзя, результат получен.

Существуют системы, в которых переписывание можно проводить бесконечно, то есть на любом шаге существует применимое правило. Например, при использовании системы переписывания, содержащей правила (2.6) и (2.7):

```
(a+b) * c ==> a*c+b*c          (2.6)
```

```
a*c+b*c ==> (a+b) * c          (2.7)
```

либо оба эти правила будут неприменимы, либо процесс их применения будет продолжаться бесконечно, так как к результату применения правила (2.6) применимо правило (2.7), а к его результату снова применимо (2.6):

`(a+b) * c ==> a*c+b*c ==> (a+b) * c ==>` …

Для таких систем необходимо задать дополнительное условие окончания применения правил. Для системы, содержащей правила переписывания (2.6) и (2.7), таким условием может быть завершение работы при получении выражения, которое не содержит скобок. Это условие запрещает применение правила (2.7) к результату, полученному после применения (2.6), и зацикливания не происходит.

2.2 Формирование волновых правил

2.2.1 Основные понятия, связанные с волновыми правилами

Волновые правила формируются из правил переписывания добавлением аннотаций. Некоторые (не обязательно все) одинаковые предложения левой и правой частей правила переписывания объявляются *основой*. Остальные фрагменты образуют *волновой фронт* и отмечаются для дальнейшего

преобразования. Если один или несколько фрагментов, выделенных как основа, оказываются внутри волнового фронта, то они образуют *волновую дыру*.

Рассмотрим правило переписывания

```
(X+Y)+Z ==> X+Y+Z
```
(2.8)

Из него может быть получено, например, следующее волновое правило:

```
( X +Y ) +Z ==> (X+Y+Z)
```
(2.9)

Здесь подчеркнутые фрагменты – это фронты волн (границы фронта волны обозначены подчеркнутыми скобками); терм Y в левой части правила переписывания и терм Y+Z в правой части находятся в волновых дырах. Основой является предложение, которое образуется из всех фрагментов, не входящих в волновой фронт, в том числе в основу входят и волновые дыры. Основа Y+Z левой части правила переписывания совпадает с основой правой части.

Другой вариант волнового правила, который может быть получен из правила переписывания (2.8):

```
( X + Y) +Z ==> (X+Y+Z)
```
(2.10)

Основа левой и правой частей этого правила – `X+Z`. Заметим, что хотя все три переменные `X, Y` и `Z` входят и в левую, и в правую часть правила переписывания, они все одновременно не могут быть основой. Основа должна являться синтаксически правильным предложением, поэтому вместе с `X, Y, Z` в нее вошли бы и все знаки операций, содержащиеся в выражении, из-за чего волновой фронт оказался бы пустым (скобки не считаются операцией, это способ задания порядка вычислений, поэтому волновой фронт, содержащий только (), не рассматривается), то есть исходное правило переписывания осталось бы в неизмененном виде. Поэтому при формировании волновых правил из правила переписывания (2.8) необходимо, чтобы хотя бы одна переменная вошла в волновой фронт.

Процесс расстановки волновых фронтов называется аннотацией. В аннотированном выражении наряду с символами объектного уровня, такими как константы, переменные и функции, появляются символы метауровня –

аннотации, с помощью которых мы будем контролировать поиск доказательства.

2.2.2 Алгоритм унификации различий

Для расстановки в двух предложениях волновых фронтов применяется метод унификации различий (difference unification) [Basin and Walsh, 1993]. Его основная идея состоит в том, что общие части сопоставляемых предложений считаются входящими в основу, а различия записываются в волновой фронт. Для этого оба предложения просматриваются слева направо до нахождения в них первой позиции, в которой стоят разные символы. Начиная с этой позиции выделяются термы, которые формируют различие. Ликвидировать различия можно по-разному. В каждой конкретной ситуации могут оказаться применимыми несколько правил алгоритма унификации различий, что может привести к получению нескольких волновых правил из одного правила переписывания.

Существуют следующие варианты удаления различий:

1. Функция может быть "спрятана" в волновой фронт (правило HIDE). Например, если различие состоит из предложения `x` (но не переменной) и некоторой функции `f(y)`, то функцию f можно записать в волновой фронт и рассматривать различие x и y. В случае если `y=x`, `x` окажется в волновой дыре, получим `f(x)`.

2. Переменная, образующая различие, может быть удалена (правило ELIMINATE). Если различие состоит из переменной `X` и терма `y`, причем терм `y` не содержит `X`, то в рассматриваемых предложениях можно выполнить подстановку `{X←y}` и продолжить унификацию различий дальше.

3. Переменная, образующая различие, может быть заменена на некоторую функцию (правило IMITATE). Если различие состоит из переменной `X` и функции `f(y)`, то переменная `X` может быть заменена на ту же функцию `f` от некоторой новой переменной $f(X_{new})$. В унифицируемых выражениях проводится подстановка $\{X \leftarrow f(X_{new})\}$, а новое различие образуют X_{new} и `y`.

После того, как различие ликвидировано, начинается поиск следующего. При новом поиске рассматриваются только те части предложений, которые еще не были отмечены как волновой фронт. Процесс продолжается до тех пор, пока не будут ликвидированы все различия. Предложения считаются унифицированными, если их основы совпадают. Формальное описание алгоритма унификации различий приведено в приложении 7.

В качестве примера рассмотрим унификацию различий термов `f(X,a)` и `f(g(a),X)`, где `X` – переменная, `a` - константа. Первое различие встречается, начиная с 3-ей позиции, различающиеся термы – X и g(a). Применим к ним правило HIDE: функция g будет "спрятана" в волновой фронт. Следующее различие (между X и a) ликвидируется применением правила ELIMINATE: выполним подстановку {X←a}. Получим термы `f(a,a)` и `f(g(a),a)`. Больше различий в основах этих термов нет, в итоге выражения аннотированы следующим образом: `f(X,a)` и `f(g(a),X)`.

Унификация различий проводится для левой и правой частей правил переписывания для формирования волновых правил, а также для гипотезы и заключения индукции (в этом случае вся гипотеза считается основой).

2.2.3 Направление распространения волнового фронта

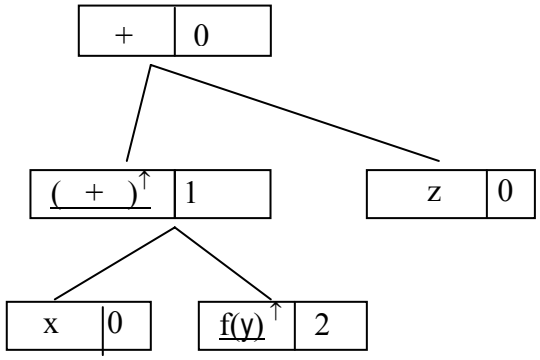
Помимо выделения в волновых правилах основы, которая не изменяется, и волновых фронтов, которые будут преобразовываться, ещё задаётся направление преобразования. Существует два таких направления: наружу (отмечается стрелкой $^{\uparrow}$) и внутрь ($^{\downarrow}$).

Рассмотрим доказательство формулы `H` → `G`, где H, G – логические выражения, H представляет собой некоторое предположение и считается истинным, а G – выражение, истинность которого нужно доказать. Представим преобразуемое выражение в виде дерева. Задача состоит в перемещении волнового фронта таким образом, чтобы можно было использовать предположение H. Для этого можно переместить волновой фронт либо к корню дерева, то есть наружу (тогда, возможно, будет найдено поддерево

совпадающее с предположением), либо, наоборот, внутрь, к листьям дерева, чтобы волновой фронт был отождествлен с переменной, находящейся в каком-нибудь листе дерева. Кроме этого, при переписывании с учётом направления волновых фронтов не возникает бесконечных цепочек вывода. Для расстановки направлений волновых фронтов вводится понятие меры аннотированного выражения. Мера описывает движение волновых фронтов по выражению, представленному в виде дерева. Разрешается проводить только те преобразования, которые уменьшают меру. Уменьшение меры не может происходить бесконечно, так как множество мер ограничено снизу нулем, следовательно, процесс преобразований завершится.

Приведём пример меры, которую можно ввести для волновых правил. Далее мы будем рассматривать такое представление аннотированных выражений, в котором каждый волновой фронт включает в себя только один функциональный символ. Если по смыслу в волновом фронте содержится несколько вложенных термов, то возникают вложенные волновые фронты. Например, в выражении $(x + f(\ y)^{\uparrow}\ )^{\uparrow} + z$ волновой фронт $f(\ldots)^{\uparrow}$ вложен в другой волновой фронт $(\ +\ )^{\uparrow}$. Для краткости записи иногда не будем явно отмечать все вложенные фронты, обозначая их одним общим: $(x + f(\ y\ ))^{\uparrow} + z$.

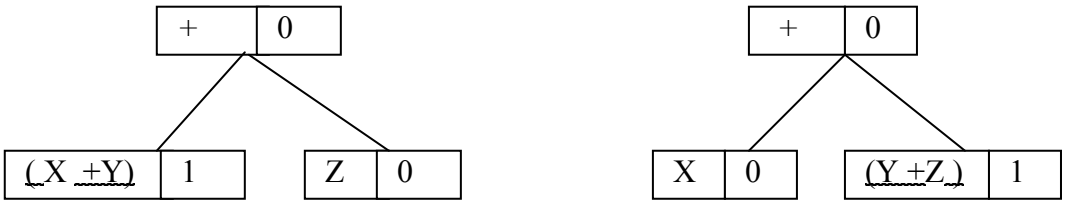
Рассмотрим представление выражения в виде дерева и припишем каждой вершине «вес» - количество волновых фронтов, в которые входит находящееся в вершине предложение. Если в вершине записан знак операции, то вес – это количество волновых фронтов, включающих в себя все выражение, заданное поддеревом с корнем в рассматриваемой вершине. Например, представление выражения $(x+f(\ y)^{\uparrow})^{\uparrow}+z$ выглядит следующим образом:



Если волновые фронты распространяются только наружу, то мерой такого выражения будем считать список из сумм весов на каждом уровне от листьев к корню дерева: [2,1,0]. Мера выражения, содержащего направленный внутрь волновой фронт, определяется аналогичным образом, только суммы весов рассматриваются в противоположном порядке (от корня к листьям дерева).

В качестве порядка на таком множестве мер рассматривается лексикографический порядок.

Рассмотрим волновое правило $(X+Y)+Z \Rightarrow X+(Y+Z)$. Учитывая расстановку волновых фронтов, получим:



Единственный возможный вариант расстановки направления распространения волн в данном случае $(X+Y)^{\uparrow}+Z \Rightarrow X+(Y+Z)^{\downarrow}$, так как тогда мера правой части – [0, 1] – будет меньше меры левой – [1,0].

2.2.4 Условия применимости волновых правил

Волновые правила применимы в доказательствах, где одна формула (цель) должна быть доказана с помощью другой формулы (предположения), и эти формулы синтаксически похожи. Тогда они могут быть аннотированы для доказательства с использованием волновых правил.

Общими условиями применимости волновых правил на произвольном этапе доказательства являются следующие:

1. Цель содержит фронт волны.
2. Существует волновое правило, левая часть которого может быть сопоставлена с подвыражением или внутренним термом цели, содержащим фронт волны.

При доказательстве могут применяться как волновые правила, так и обычные правила переписывания, но обычные правила переписывания могут

осуществлять преобразования только в рамках волновых фронтов таким образом, чтобы не затрагивалась основа.

2.3 Применение волновых правил с распространением волн наружу

При использовании волновых правил с распространением волн наружу преобразования проводятся таким образом, чтобы волновой фронт содержал внутри себя все бóльшую часть выражения и, в конце концов, основа оказалась бы целиком внутри волновой дыры. То есть, если представить выражение в виде дерева, то рассматривается перемещение волнового фронта вверх по дереву, по направлению к корню, до тех пор, пока не появится поддерево, не содержащее волновых фронтов и соответствующее предположению. Предположение считается истинным, чем и можно воспользоваться для продолжения доказательства.

Рассмотрим в качестве примера доказательство свойства ассоциативности операции сложения для натуральных чисел: `t+(Y+Z)=(t+Y)+Z`. Индукцию будем вести по переменной `t`. Шаг индукции этого доказательства запишем следующим образом:

$$t+(Y+Z)=(t+Y)+Z \quad \rightarrow \quad s(t)^{\uparrow}+(y+z)=(s(t)^{\uparrow}+y)+z$$

Здесь `s(t)` – функция, выдающая `t+1` в качестве результата для неотрицательного целого числа `t`. В рассматриваемом примере волны распространяются наружу (это обозначено стрелками).

На основе свойств операции сложения

`s(T) +L = s(T+L)` ,

`s(X)=s(Y)) ≡ (X=Y)`

можно сформулировать следующие волновые правила:

$$s(T)^{\uparrow}+L \Longrightarrow s(T+L)^{\uparrow} \qquad (2.11)$$

$$s(X)^{\uparrow}=s(Y)^{\uparrow} \Longrightarrow X=Y \qquad (2.12)$$

Используя эти волновые правила, преобразуем заключение индукции:

$s(t)^{\uparrow} + (y+z) = (s(t)^{\uparrow} + y) + z$ ==>

{используем правило 2.11} ==> $s(t+(y+z))^{\uparrow} = s((t+y))^{\uparrow} + z$

{используем правило 2.11} ==> $s(t+(y+z))^{\uparrow} = s((t+y)+z)^{\uparrow}$

{используем правило 2.12} ==> $t+(y+z) = (t+y)+z$

Получено выражение, в точности совпадающее с гипотезой индукции, доказательство успешно завершено.

На практике иногда возникают ситуации, когда ни одно из волновых правил применить нельзя и при этом пример предположения в цели не получен, но получен пример некоторой части предположения. Если предположение имеет вид $H_1=H_2$, а после проведении преобразований цель содержит предложение $H_1\lambda$ – пример H_1, то для доказательства цели можно воспользоваться предположением как правилом переписывания $H_1 ==> H_2$. Аналогично, если цель содержит $H_2\lambda$, то предположением можно воспользоваться как правилом переписывания $H_2 ==> H_1$.

Вернемся к рассмотренному примеру доказательства свойства ассоциативности сложения неотрицательных целых чисел. Предположим, что правило (2.12) не известно. Тогда доказательство остановится после преобразования заключения к виду

$s(t+(y+z))^{\uparrow} = s((t+y)+z)^{\uparrow}$

Правило (W*) неприменимо, пример гипотезы индукции $t+(Y+Z)=(t+Y)+Z$ в заключении не получен, но получен пример ее левой части, поэтому воспользовавшись гипотезой как правилом переписывания

$t+(Y+Z) ==> (t+Y)+Z$,

преобразуем заключение к виду

$s((t+y)+z))^{\uparrow} = s((t+y)+z)^{\uparrow}$

Полученное выражение истинно, так как слева и справа от операции = стоят одинаковые термы. Истинность заключения индукции доказана, доказательство на этом успешно завершается.

2.4 Применение волновых правил с распространением фронта волны внутрь

Распространение фронта волны наружу – лишь один из способов применения волновых правил. Фронт волны может распространяться и внутрь выражения, в направлении к свободной переменной. Идея таких преобразований состоит в том, что свободной переменной может быть сопоставлен произвольный терм, в том числе и содержащий волновой фронт. После такого сопоставления волновой фронт будет “поглощен” свободной переменной.

Свободные переменные возникают, когда проводится доказательство теоремы, содержащей не менее двух переменных, связанных квантором всеобщности.

Рассмотрим выражение $\forall$ N:nat.$\forall$M:nat.Q(N,M), обозначающее, что для двух натуральных чисел M и N справедливо соотношение Q(N,M). Если в качестве индукционной переменной будет выбрана N, то шаг индукции может быть записан следующим образом:

$\forall$ N:nat.[$\forall$M:nat.Q(N,M)$\rightarrow\forall$M :nat.Q(N+1,M)]

Перед доказательством кванторы необходимо вынести из внутренних выражений:

$\forall$N:nat.[($\forall$M:nat.Q(N,M))$\rightarrow$($\forall$m:nat.Q(N+1,M))]$\rightarrow$

{так как A $\rightarrow$ B $\equiv$ $\neg$A $\vee$ B}

$\rightarrow\forall$N:nat.[$\exists$M:nat.$\neg$Q(N,M)$\vee\forall$K :nat.Q(N+1,K)]$\rightarrow$

$\rightarrow\forall$N:nat.$\exists$M:nat.$\forall$K :nat. $\neg$Q(N,M)$\vee$ Q(N+1,K)$\rightarrow$

$\rightarrow$ $\forall$N:nat.$\forall$K :nat.[Q(N,f(M)) $\rightarrow$Q(N+1,K)]

Таким образом, первое вхождение переменной M, которая была связана квантором всеобщности и не являлась индукционной переменной, может быть заменено на некоторый терм, и, вообще говоря, оно не связано со вторым вхождением, которое остается переменной, связанной квантором всеобщности. В рассмотренном примере это переменная K. Такая переменная называется *стоком* и обозначается $\lfloor$K$\rfloor$.

Если фронт волны целиком окажется на месте стока, он будет сопоставлен с `f(N)`, а в полученном выражении будет пример индукционной гипотезы.

Рассмотрим в качестве примера функцию `rotate(n,l)`, которая удаляет `n` элементов из начала списка и добавляет их в конец (то есть осуществляет циклический сдвиг списка на n элементов влево). Докажем её свойство

`∀l:list(τ).∀k:list(τ).rotate(length(l), l<>k) = k<>l,`

где `length(l)` – функция, возвращающая длину списка `l`, `<>` – операция конкатенации списков. Это свойство означает, что если взять `length(l)` элементов списка `l<>k` и поставить их в конец списка, то будет получен список `k<>l`. Действия функций определим с помощью следующих правил переписывания:

`length(nil)⇒ 0` (`nil` – терм, обозначающий пустой список)

`length(H::T)⇒ length(T) + 1`

(`H::T` – операция добавления элемента `H` в начало списка `T`)

`rotate(0,L)⇒ L`

`rotate(N+1,nil) ⇒ nil`

`rotate(N+1,H::T) ⇒ rotate(N,T<>(H::nil))`

`nil <> L ⇒ L`

`(H::T) <> L ⇒ H ::(T <> L)`

Для доказательства может быть использован следующий вариант построения индукции:

базовый случай: `rotate(length(nil),nil<>k)=k<>nil,`

гипотеза индукции:

`∀h:τ.∀l:list(τ).rotate(length(l),l<>K)=K<>l →`

заключение:

`rotate(length(h::l),h::l<>k)=k<>h::l`

Применим волновые правила

`rotate(s(N)↑ , (H::T)↑) ⇒ rotate(N,(T <>(H::nil))↓ )`

$L <> (H::T)^{\uparrow} \Rightarrow (L <> (H::nil))^{\downarrow} <> T$

и приведенные выше правила переписывания к заключению индукции:

$rotate(length(h::l)^{\uparrow}, (h::l)^{\uparrow} <> \lfloor k \rfloor) = \lfloor k \rfloor <> (h::l)^{\uparrow} \Rightarrow$

$\Rightarrow rotate((length(l)+1)^{\uparrow}, (h::l <> \lfloor k \rfloor)^{\uparrow}) = \lfloor k \rfloor <> (h::l)^{\uparrow} \Rightarrow$

$\Rightarrow rotate(length(l), (l <> \lfloor k \rfloor <> (h::nil))^{\downarrow}) = \lfloor k \rfloor <> (h::l)^{\uparrow} \Rightarrow$

$\Rightarrow rotate(length(l), l <> \lfloor k \rfloor <> (h::nil)^{\downarrow}) = \lfloor k \rfloor <> (h::nil)^{\downarrow} <> l$

$\Rightarrow rotate(length(l), l <> \lfloor k <> (h::nil) \rfloor) = \lfloor k <> (h::nil) \rfloor <> l$

После выполнения подстановки $\{K \leftarrow k <> (h::nil)\}$ заключение индукции будет сведено к гипотезе. Утверждение доказано.

2.5 Преимущества применения волновых правил. Особенности применения волновых правил для синтеза программ

По сравнению с применением обычных правил переписывания, применение волновых правил дает существенные преимущества.

При применении волновых правил накладываются дополнительные ограничения на направление движения волнового фронта и, следовательно, на применение правил, что уменьшает перебор.

Процесс применения волновых правил всегда завершается (так как при каждом применении правила уменьшается мера преобразуемого предложения, а её уменьшение не может быть бесконечным). Завершение связано либо с неприменимостью ни одного из правил, либо с тем, что результат получен (например, волновой фронт окружает все выражение – при распространении волны наружу).

Применение волновых правил в доказательстве по индукции, таким образом, позволяет сделать процесс преобразования индукционного заключения к гипотезе более эффективным. Однако из-за того, что волновые правила, как и правила переписывания, формируются из определений объектов, о которых ведётся рассуждение, при применении их для синтеза возникает новая

проблема. Объект (программа), заданное свойство которого (спецификацию) мы хотим доказать, не известен, поэтому для него и нет соответствующих аксиом, с помощью которых можно было бы проводить преобразования. Уже упомянутые во введении применения волновых правил для синтеза программ основаны на том, что задается библиотека шаблонов программ, а в процессе доказательства выясняется, какой из них является подходящим. Такой подход оказывается не очень удобным, так как для многих задач приходится создавать новые шаблоны. Тогда как в процессе синтеза с помощью метода дедуктивных таблиц структура программы формируется именно во время доказательства и, таким образом, метод приспособлен для синтеза более широкого круга задач. Разработанный в диссертации новый метод объединяет преимущества этих двух методов.

Глава 3. Автоматизация синтеза программ в дедуктивной таблице

Метод дедуктивных таблиц даёт набор правил, необходимых для построения базовых конструкций функциональной программы. Однако выбор правила на каждом шаге доказательства недетерминирован. При применении всех правил ко всем строкам таблицы возникает огромный перебор вариантов доказательства. В данной главе будут рассмотрены эвристики, которые позволяют сократить перебор и дают возможность использовать метод дедуктивных таблиц для автоматического синтеза программ. Поиск и разработка таких эвристик явились одной из основных задач диссертации.

В диссертации рассматриваются эвристики двух типов, которые могут использоваться независимо друг от друга. Во-первых, это эвристики, блокирующие применение некоторых правил, ведущих к бесполезным для построения программы выводам. К таким эвристикам относится учёт полярности заменяемых выражений и учёт типов операндов при проведении вывода. Ко второму типу относятся эвристики, позволяющие наметить план доказательства и после этого проводить синтез в дедуктивной таблице в соответствии с найденным планом. В качестве такой эвристики используются волновые правила.

Отметим, что применение указанных эвристик не приводит к потере решения. Если с помощью "чистого" метода дедуктивных таблиц можно синтезировать требуемую функцию, то она будет синтезирована и с использованием эвристик. Эвристики могут оказаться применимыми не всегда, но если их применение возможно, то они дают заметное сокращение перебора вариантов при синтезе. Как показывает практика, использование ограничений на применение дедуктивных правил позволяет сократить перебор в среднем в 2-6 раз, а при планировании доказательства с использованием волновых правил - на 2 порядка. Именно это и позволило за приемлемое время автоматически

синтезировать некоторые программы, дедуктивный синтез которых ранее удавалось провести лишь вручную.

3.1 Эвристики, ограничивающие число применимых правил

3.1.1 Учёт полярности логических выражений

Эта эвристика была предложена в работе [Manna and Waldinger, 1992]. Полярность подвыражения, находящегося в колонке целей дедуктивной таблицы, считается положительной, если это подвыражение находится в области действия чётного числа отрицаний, и отрицательной, если число отрицаний нечётно. В отличие от определения полярности для термов (введенного в главе 1) здесь мы рассматриваем полярность логических выражений. Так, например, полярность подвыражения `A` в выражении `A and B` положительна, а в выражении `not(A) and B` – отрицательна. Некоторые подвыражения могут иметь и двойную полярность, как, например, `D` и `E` в выражении `D≡E`; это объясняется тем, что `D≡E` эквивалентно выражению `(¬D∨E)∧(D∨¬E)`, содержащему вхождения `D` и `E` с положительной полярностью и с отрицательной.

Для подвыражений в колонке утверждений дедуктивной таблицы полярности определяются противоположным образом, так как любое утверждение может быть перенесено в колонку целей с отрицанием.

Эвристика заключается в том, что в правилах эквивалентной замены и резолюции на `false` следует заменять только подвыражения, хотя бы одно вхождение которых имеет отрицательную или двойную полярность, а на `true` – только при положительной или двойной полярности.

Для пояснения этой эвристики рассмотрим в качестве примера применение правила резолюции к строкам

	Assertions	Goals	f(a)
A	$[a=NIL]^{-}$		
G		$[(a=NIL)]^{+}$ and (z=1)	z

В утверждении A выражение a=NIL имеет отрицательную полярность (обозначена символом –), в цель G входит такое же выражение, но его полярность положительна (обозначена +). Перенесем A в колонку целей:

AG		not([a=NIL]⁻)	

Для строк G и AG есть несколько вариантов применения правила резолюции. В первом варианте, выражение a=NIL цели AG будет заменено на false, а соответствующее выражение в цели G – на true. Получим новую цель not(false) and true and (z=1), которая после упрощения добавляется в таблицу:

G1		(z=1)	z

Второй вариант предполагает замену выражения a=NIL цели G на false, а соответствующего выражения в цели AG – на true. При этом получаем цель not(true) and false and (z=1), которая после упрощения преобразуется в false. Добавлять в таблицу такую строку не имеет смысла, так как она является бесполезной для дальнейшего доказательства. Применение стратегии полярности исключает варианты применения правил, подобные второму.

3.1.2 Учёт типов термов при выводе

Рассмотрим еще одну эвристику, которая состоит в том, что вместе с каждым термом, встречающимся в дедуктивной таблице, хранится его тип, и этот тип учитывается при синтезе. Это позволяет избежать появления бессмысленных конструкций. Например, предикат emptylist(x), проверяющий, является ли список x пустым, не должен иметь в качестве аргумента целое число, поэтому ветвь доказательства, содержащая конструкцию emptylist(2+3) отбрасывается.

Кроме введения типов, задается их иерархия, которая позволяет упорядочить проверку условий во вложенных условных конструкциях выходного терма и отбросить некоторые некорректные проверки. Например, при вычислении выражения (b=1/a) ∧ (a>0) необходимо, чтобы сначала было проверено

условие (a>0) и лишь после него (b=1/a), поскольку конструкция if (b=1/a) then if (a>0) then Q является ошибочной при a=0.

В диссертации рассматриваются объекты только двух типов: списки и целые числа, но аналогичные утверждения могут быть применены к другим типам.

Под списком подразумевается последовательность чисел, заключенная в скобки. Для списков в диссертации введена следующая иерархия типов (перечислены в порядке убывания):

list_(0) – произвольный список,

list_(1) – список, содержащий хотя бы 1 элемент,

list_(2) - список, содержащий как минимум 2 элемента,

и т.д.

Для целых чисел выделена следующая иерархия типов (перечислены в порядке убывания):

int_(0) – произвольное целое число,

int_(1) – целое число, не равное нулю,

int_(2) – положительное целое,

и т.д.

В системе АЛИСА, реализованной в диссертации типы термов определяются следующим образом.

Во-первых, в спецификацию синтезируемой функции добавлена специальная конструкция с ключевым словом for, с помощью которой задаются типы входных параметров функции. Так, например, в спецификации

```
<div(i,j),mod(i,j)> <==  for number (i) and number (j) find <y,z>
such that if (i>=0) and (j>0) then (i=y*j+z) and (z>=0) and (z<j)
```

после ключевого слова for записано, что i, j – целые числа. Это условие не отражается явно в дедуктивной таблице, но позволяет учитывать тип входных параметров.

Во-вторых, типы термов определяются по применяемым к ним операциям и функциям. Например, если терм является слагаемым в некоторой сумме, то его

типом считается `int_(0)` – целое число, если терм используется как делитель, то его тип – `int_(1)`, а если терм является аргументом функции `emptylist`, то тип терма – `list_(0)`. Если же присутствует операция сравнения терма с термом некоторого типа `Type`, то типом такого терма тоже считается `Type`; например, терм `t`, входящий в выражение `t=NIL`, имеет тип `list_(0)`, а типом терма `x`, входящего в выражение `x<2`, считается `int_(0)`.

Если тип терма не удается определить по контексту, то считается, что этот терм имеет произвольный тип, который обозначается как `unknown_`. Этот тип считается максимальным в любой иерархии типов.

При проведении синтеза типы термов учитываются следующим образом. Во-первых, при унификации выражений дополнительно учитывается соответствие их типов. Так, терм-список и терм-число не будут унифицированы, а для терма типа `unknown_` можно проводить унификацию с термом любого типа.

Учёт информации о типах термов позволяет исключить появление в таблице некоторых бесполезных для синтеза выражений. Например, если не учитывать типы, то при применении правила замены эквивалентных термов к следующим строкам A и G:

	Assertions	Goals	f(a)
A	0*v=0		
G		if emptylist(x) then z=0 else z=head(x)+a	z

в таблицу будут добавлены 4 строки:

G1		if emptylist(0*x1) then z1=0 else z1=head(x1)+a	z1
G2		if emptylist(0) then z2=0 else z2=head(0)+a	z2
G3		if emptylist(x3) then 0*z3=0 else 0*z3=head(x3)+a	0*z3
G4		if emptylist(x) then 0=0 else 0=head(x)+a	0

Однако строки G1 и G2 являются бесполезными для дальнейшего доказательства, так как в этих строках предикат `emptylist`, проверяющий, является ли список пустым, имеет в качестве входного параметра целочисленный аргумент, а такое выражение является некорректным. Такие строки исключаются из таблицы и не участвуют в последующем синтезе.

В диссертации предложено также и другое использование информации о типах – для упорядочения условий в выходном терме. Проблема упорядочения условий возникает из-за того, что в колонках утверждений и целей дедуктивной таблицы операции $\wedge$ и $\vee$ являются коммутативными, а в выходной колонке порядок вычисления операндов конъюнкции и дизъюнкции существенен. Например, терм

`if ¬(emptylist(a)) then if ¬(emptylist(tail(a))) then Q` (где Q – вычислимый терм) может быть вычислен даже для пустого списка `(a=NIL)`, тогда как вычисление терма

`if not(emptylist(tail(a))) then if not(emptylist(a)) then Q`, полученного из исходного терма с помощью правил эквивалентных преобразований, невозможно для пустого списка, так как для него функция `tail` не определена. Поэтому, при формировании в выходной колонке таблицы вычислимого выражения, необходимо упорядочить условия, проверяемые в конструкциях `if-then-else`. Для этого при применении дедуктивных правил (правила резолюции, замены эквивалентных термов, выражений, аргументов в отношениях) на `true` (или `false`) в первую очередь заменяется выражение, содержащее терм минимального типа (среди тех термов, типы которых можно сравнить).

Проиллюстрируем использование этой эвристики на простом примере (более сложный пример представлен в приложении 5). Предположим, в дедуктивной таблице содержатся строки G1, G2 и G3, имеющие некоторые выходные термы t1, t2, и t3 соответственно:

G1		¬(b=1/a) ∧ (a>0)	t1
G2		¬(a>0)	t2
G3		(b=1/a)	t3

Применив правило резолюции к строкам G2 и G1 для общего подвыражения `(a>0)`, а затем правило резолюции к полученной строке и к G3, мы добавим в таблицу новую строку с тождественно истинной целью и условным выходом:

G4		true	if (b=1/a) then t3 else (if (a>0) then t1 else t2)

Формально правило резолюции применимо к таким строкам, но в результате получена строка, выходной терм которой не может быть вычислен при a=0. Учитывая информацию о типе переменной a в строке G1, получим, что выражение `(a>0)`, в котором `a` имеет тип `int_(0)` (целое число), не может быть заменено на true или false раньше, чем выражение ¬`(b=1/a)`, тип переменной `a` в котором – это `int_(1)` (целое число, не равное нулю). Поэтому правило резолюции к строкам G1 и G2 оказывается неприменимо. Но это правило применимо к строкам G1 и G3 для общего подвыражения `(b=1/a)`, а затем к полученной строке и G2. В этом случае в таблицу будет добавлена новая строка с тождественно истинной целью и условным выходом, который может быть вычислен и при `a=0`:

G4		true	if (a>0) then (if (b=1/a) then t3 else t1) else t2

Дадим обоснование предложенной эвристики. При применении дедуктивных правил выражение, заменяемое на true или false, в общем случае появляется в выходной колонке в качестве условия в условном терме. При появлении вложенных условных термов их условия должны проверяться в такой последовательности, чтобы сначала были проверены самые "общие" условия (содержащие элементы максимальных типов, т.е. `int_(0)` или `list_(0)`), и только потом проверялись те, которые от этих условий зависят, так как при невыполнении "общих" условий вычисление остальных может оказаться невозможным. Например, чтобы проверить какое-либо условие, касающееся "хвоста" списка (`tail`), надо знать, что список непустой, иначе вычисление функции `tail` будет возвращать ошибку. Таким образом, нужно применять дедуктивные правила так, чтобы сначала в выходной колонке появлялись выражения с аргументами минимального типа, затем бóльшего и т.д.

Отметим, что не все типы упорядочены, например, сравнение типов `int_(i)` и `list_(j)` невозможно, так как они входят в разные иерархии. В таком случае при применении дедуктивного правила возможно несколько вариантов: любое из подвыражений может рассматриваться в качестве выражения, содержащего минимальный тип (при условии, что в выражение не входит

рассматриваемый терм типа int_(i_1), где i_1>i, и типа list_(j_1), где j_1>j). Тип unknown_, по определению, считается бóльшим, чем любой другой тип.

Использование информации о типах аргументов и иерархии типов, таким образом, позволяет отсечь бесполезные для синтеза ветви доказательства и частично сократить перебор.

3.2 Применение волновых правил для планирования доказательства в дедуктивной таблице

3.2.1 Описание метода

При проведении синтеза в дедуктивной таблице значительная часть перебора приходится на построение рекурсивных ветвей функций. На этом этапе, как отмечено в главе 1, применяется доказательство по индукции, причём наиболее сложным является доказательство шага индукции. Здесь существенную помощь могут оказать волновые правила (см. главу 2), позволяющие заметно сократить перебор. Предлагается использовать это их преимущество следующим образом.

Поскольку техника доказательства с помощью волновых правил отличается от техники доказательства в дедуктивной таблице, то при доказательстве шага индукции мы временно "выходим" из дедуктивной таблицы и рассматриваем доказательство шага индукции как самостоятельную задачу. Эта задача будет решаться с помощью волновых правил, которые были заранее сформированы из аксиом, содержащихся в дедуктивной таблице. Если эта задача была успешно решена, то мы запоминаем путь доказательства – номера аксиом, соответствующих примененным волновым правилам, и "возвращаемся" в дедуктивную таблицу, где воспроизводим доказательство согласно найденному пути, чтобы одновременно с ним синтезировать в выходной колонке таблицы рекурсивную ветвь функции.

Отметим, что если с помощью волновых правил не было построено доказательство шага индукции, то поиск доказательства будет продолжен с помощью дедуктивных правил.

3.2.2 Построение волновых правил

Волновые правила формируются из известных аксиом, содержащихся в дедуктивной таблице. Для одной аксиомы возможно получение нескольких правил. Правила формируются один раз в момент появления аксиомы в дедуктивной таблице, затем информация о правиле сохраняется для дальнейшего использования. В эту информацию помимо левой и правой частей правила входит:

- номер аксиомы, из которой правило было получено,
- условия применимости правила (полярность, условие, заданное в виде логического выражения),
- соответствующее этому волновому правило дедуктивное правило (оно определяется при построении волнового правила).

Чтобы построить волновые правила, для каждой из аксиом записываются соответствующие ей правила переписывания, а затем к ним добавляются аннотации. При этом для каждой аксиомы возможно построение нескольких правил переписывания, а для каждого из них может существовать несколько вариантов аннотации. Чтобы сформированные правила были корректными волновыми правилами, они должны сохранять основу при переписывании и мера правой части правила должна быть меньше меры левой. Для аннотации правил переписывания используется алгоритм унификации различий (см. приложение 7).

Формирование правил переписывания происходит следующим образом.

1. Аксиоме вида `if A then B` соответствует правило переписывания `B==>A`. Мы учитываем, что доказательство с помощью волновых правил ведется в "обратном" направлении: от заключения индукции к гипотезе, поэтому и правила переписывания ориентированы в противоположном направлении по отношению к операции логического следствия. В качестве соответствующего дедуктивного правила записывается правило резолюции. Обоснование выбора правила резолюции следующее. На практике мы работаем с выражениями, которые приведены к специальному виду (к конъюнкции

выражений, если это утверждение, и к дизъюнкции выражений, если это цель), а первыми применяются дедуктивные правила расщепления. После этого остальные правила применяются к целям, имеющим вид $E_1 \wedge \ldots \wedge E_n$ (соответственно, к утверждениям вида $E_1 \vee \ldots \vee E_n$), полученным в результате применения правил расщепления. Таким образом, при применении правила резолюции к цели A и G для общего подвыражения B:

A	if A then B		
G		$E_1 \wedge \ldots \wedge B \wedge \ldots \wedge E_n$	

мы получим строку

G		$E_1 \wedge \ldots \wedge A \wedge \ldots \wedge E_n$	

в которой B будет заменено на A, что соответствует применению правила переписывания (заметим, что в общем случае мы ищем в цели G подвыражение B_1, которое можно унифицировать с B и в результирующей строке потребуется выполнение подстановки λ, которая является унификатором B и B_1).

Для аксиомы `if A then B` также могут быть сформированы те же правила, что и для аксиомы `B`, но с добавлением условия `A`, проверяемого при применении правил.

2. Аксиома, содержащая операцию эквивалентности `A≡ B`, рассматривается как две аксиомы: `if A then B` и `if B then A`, волновые правила для которых формируются так же, как описано в предыдущем пункте, только в качестве соответствующего дедуктивного правила записывается правило эквивалентной замены логических выражений (а не правило резолюции).

3. Аксиома, содержащая операцию равенства `A=B`, порождает два правила переписывания: `A==>B` и `B==>A`. Соответствующее им дедуктивное правило – правило замены эквивалентных термов.

4. Аксиома, представляющая собой конъюнкцию $A_1 \wedge \ldots \wedge A_n$, рассматривается как n аксиом $A_1, \ldots, A_n$, для каждой из которых строятся волновые правила.

5. Аксиома, содержащая дизъюнкцию целей, преобразуется в условное выражение с помощью правил $\neg A \vee B$ = `if A then B`, но с учетом информации

о типах термов, входящих в A и B: если терм x входит в A и в B, то тип его вхождения в B должен быть не больше, чем тип вхождения в A.

6. Если аксиома A является отношением (предикатом от двух аргументов, записанным в префиксной или инфиксной форме, то есть A=rel(L,R) или A= L rel R), и отношение отлично от =, то формируются два правила переписывания L==>R и R==>L с пустым условием и с отрицательной и положительной (относительно rel) полярностями соответственно. В качестве соответствующего дедуктивного правила записывается правило замены аргументов в отношениях.

Пример. Согласно сказанному, для аксиомы (a+b)*c = a*c + b*c будут получены следующие волновые правила

(a + b) $^{\uparrow}$ *c ==> (a*c + b*c) $^{\uparrow}$,

(a + b) $^{\uparrow}$ *c ==> (a*c + b*c) $^{\uparrow}$,

(a*c + b*c) $^{\downarrow}$ ==> (a + b) $^{\downarrow}$ *c,

(a*c + b*c) $^{\downarrow}$ ==> (a + b) $^{\downarrow}$ *c.

Все эти правила рассматриваются с пустым условием, они применимы для термов любой полярности, а в качестве соответствующего дедуктивного правила записано правило замены эквивалентных термов.

3.2.3 Доказательство шага индукции с помощью волновых правил

В записи шага индукции участвуют гипотеза и заключение. Гипотеза формируется в результате применения правила индукции (см. главу 1) к исходной цели таблицы.

Конкретный вариант гипотезы индукции получается при выборе конкретного wf-отношения. Эти отношения заранее заданы и записаны в дедуктивной таблице в виде аксиом. Выбор подходящего отношения осуществляется с учетом типа переменной, по которой ведется индукция. Например, для целочисленной переменной i может быть выбрано wf-отношение $i-1 <_{wf} i$. Это отношение сформулировано в виде аксиомы

if $i \geq 0$ then $i-1 <_{wf} i$		

Аксиома выражает факт, что для использования этого wf-отношения нужно доказать условие `i≥0`. Чтобы доказать условие, в таблице проводится поиск утверждения `i≥0` или цели ¬(`i≥0`) (или цели, которая унифицируема с ними). Если условия применения wf-отношения доказаны, то оно может быть использовано для построения гипотезы. Если находится несколько подходящих wf-отношений, то они рассматриваются одно за другим.

Если гипотеза индукции является конъюнкцией выражений, то каждое из них отдельно, а также любая их комбинация может быть рассмотрена в качестве гипотезы индукции. Например, если гипотеза, полученная с помощью дедуктивного правила, имеет вид H1 ∧ H2, то существует 4 способа записи шага индукции: H1 ∧ H2 → C, H2 ∧ H1 →C, H1 → C и H2 → C, где C – заключение индукции. Так как при добавлении аннотаций предложения просматриваются слева направо, а коммутативность конъюнкции явно не учитывается, то необходимо рассмотреть варианты гипотез, включая различные перестановки гипотез в конъюнкции.

Заключение индукции формируется из исходной цели таблицы `Q[b,f(b)]`, для которой выполняется специальная подстановка для переменной `b`, по которой ведётся индукция. С помощью этой подстановки исходная переменная `b` выражается через переменную x, с которой она связана wf-отношением $x <_{wf} b$, использованном при построении гипотезы индукции, согласно правилу индукции (описанному в главе 1): `if` $x <_{wf} b$ `then Q[x,f(x)]`. Например, если было использовано wf-отношение $b-1 <_{wf} b$, то в заключении индукции нужно выполнить подстановку `b` ← `(b-1)+1`.

После того, как гипотеза и заключение индукции сформированы, производится расстановка волновых фронтов. Основой считается гипотеза индукции. Различия между гипотезой и заключением отмечаются в заключении.

Возможно, таким образом, получение нескольких вариантов шага индукции. Они рассматриваются последовательно, пока не будет найдено доказательство, либо пока все варианты не будут рассмотрены.

Для доказательства шага индукции мы используем "обратный" вывод – с помощью волновых правил преобразуем заключение к гипотезе. При этом последовательность применяемых волновых правил запоминается. Так как волновые правила формируются из известных аксиом, каждому волновому правилу можно однозначно сопоставить аксиому, из которой оно получено, и название дедуктивного правила, которое нужно будет применять в дальнейшем. Пусть, например, при преобразовании заключения индукции G, было применено волновое правило (P*R+Q*R)$^{\downarrow}$==>(P+Q)$^{\downarrow}$*R. Оно сформировано из заданной аксиомы (P+Q)*R = P*R+Q*R, при этом в качестве соответствующего дедуктивного правила записано правило замены эквивалентных термов. Номер аксиомы и название дедуктивного правила будут добавлены к пути доказательства. При построении доказательства в дедуктивной таблице (по заданному пути) рассмотренному применению волнового правила будет соответствовать применение правила эквивалентной замены термов для строк A и G, где A – аксиома, соответствующая примененному волновому правилу:

A	(P+Q)*R= P*R+Q*R		
G		G	z

Последовательности применённых волновых правил переписывания сопоставляется последовательность номеров аксиом и названий дедуктивных правил, которую мы назовем *путем доказательства*. Доказательство считается успешно завершенным, если получен целиком пример гипотезы индукции в заключении, а волновой фронт либо был убран в процессе доказательства, либо содержит истинное выражение. В результате проведения доказательства с помощью волновых правил будет получен путь доказательства, по которому мы восстанавливаем доказательство в дедуктивной таблице. Для этого дедуктивные правила последовательно применяются к исходной цели и

аксиомам из найденного пути, а в выходной колонке формируется текст функции. Если же доказательство с помощью волновых правил закончилось неудачей (пример гипотезы не найден, и ни одно из волновых правил больше не применимо), то происходит возврат назад и поиск другого варианта аннотации шага индукции, при котором может быть сформирована новая гипотеза (если исходная состояла из конъюнкции выражений). Если все варианты аннотации рассмотрены, а доказательства не найдено, возможен возврат назад и построение новой гипотезы индукции, в котором участвует новое wf-отношение.

3.2.4 Применение волновых правил для построения пути доказательства

Рассмотрим использование волновых правил для построения пути доказательства на примере синтеза функций div и mod, вычисляющих неполное частное и остаток от деления для двух целых чисел. Отметим, что ниже приводится только окончательное доказательство, а не все варианты, которые рассматривались при его поиске.

Спецификация функций div и mod может быть записана следующим образом:

```
<div(i,j),mod(i,j)> <==  for number(i) and number (j) find <y,z>
such that if (i>=0) and (j>0) then (i=y*j+z) and (z>=0) and (z<j)
```

С помощью дедуктивных правил в таблице может быть получена строка, соответствующая нерекурсивной ветви функции:

Assertions	Goals	div(i,j)	mod(i,j)
	i<j	0	i

При доказательстве шага индукции (будем проводить индукцию по переменной i) используется wf-отношение $i-k <_{wf} i$. Это отношение получено из аксиомы

if (i>=0) $\wedge$ (k>0) $\wedge$ (k<=i) then i-k $<_{wf}$ i			

и его условия выполнены при k=j, так как $j>0$, $i \geq 0$ согласно условию задачи, а случай j>i уже рассмотрен и соответствует нерекурсивной ветви функций. Поэтому для указанного wf-отношения мы можем построить следующую гипотезу индукции:

(i-j=div(i-j,j)*j+mod(i-j,j))∧(mod(i-j,j)≥0)∧(mod(i-j,j)<j)

Заключение же индукции может быть представлено в виде дизъюнкции трёх целей:

not (i>=0)

not(j>0)

(i-j=div(i-j,j)*j+mod(i-j,j)) ∧ (mod(i-j,j)≥0) ∧ (mod(i-j,j)<j)

из которых только третья является существенной для построения рекурсивной ветви, так как эта цель содержит выходную переменную (z). Выполним в ней специальную подстановку i=(i-j)+j и запишем шаг индукции:

(i-j=div(i-j,j)*j+mod(i-j,j))∧(mod(i-j,j)≥0)∧(mod(i-j,j)<j)→

((i-j)+j)=y*j+z ∧ (z≥0) ∧ (z<j)

Различия между гипотезой и заключением (волновой фронт) выделены подчеркиванием в заключении. Отметим, что y и z не включены в волновой фронт, потому что они являются свободными переменными и могут быть унифицированы с соответствующими выражениями из гипотезы индукции.

Чтобы получить в заключении индукции пример гипотезы, будем использовать следующие волновые правила:

$$((U + V)^{\uparrow} = W) \Longrightarrow (U = ((-1)*V + W)^{\downarrow}) \qquad (3.1)$$

$$(P*R + Q*R)^{\downarrow} \Longrightarrow (P + Q)^{\downarrow} *R \qquad (3.2)$$

полученные из аксиом

$$(U+V= W) \equiv (U = (-1)*V + W)$$

$$P*R + Q*R = (P + Q)*R$$

Применим волновое правило (3.1) для преобразования заключения индукции :

$$(i-j = ((-1)* j+ y*j+z)^{\downarrow}) \text{ and } (z>=0) \text{ and } (z<j) \qquad (3.3)$$

Затем, применив правило (3.2) к выражению (3.3), получим:

$$(i-j = ((-1)+y)^{\downarrow} *j+z) \text{ and } (z>=0) \text{ and } (z<j) \qquad (3.4)$$

В выражении (3.4) y – свободная переменная. Мы можем ввести новую свободную переменную y1, заменив (-1+y)↓ на y1. Преобразованное выражение

$$(i\text{-}j = y1 * j + z) \text{ and } (z >= 0) \text{ and } (z < j) \qquad (\mathbf{3.5})$$

не содержит волновых фронтов и после подстановки {y1←div(i-j,j), z←mod(i-j,j)} становится эквивалентным гипотезе индукции. Применение волновых правил успешно завершено.

Из проведенного доказательства мы получили подстановку для выходных переменных исходной цели {y←div(i-j,j)+1, z←mod(i-j,j)} и путь проведения доказательства для получения рекурсивной ветки, состоящий из двух шагов:

Шаг 1: используется аксиома (U+V=W) ≡ (U=W+(-1)*V) (выражение, заменяемое в заключении индукции, – это (i-j+j=(div(i-j,j)+1)*j+mod(i-j,j))), дедуктивное правило – правило замены эквивалентных выражений;

Шаг 2: используется аксиома (P+Q)*R=P*R+Q*R, (выражение, заменяемое в заключении индукции, – это ((-1)*j+(div(i-j,j)+1)*j)), дедуктивное правило – правило замены эквивалентных термов.

Теперь возвращаемся к дедуктивной таблице, применяем правило расщепления к исходной цели (так как она представляет собой дизъюнкцию, записанную в форме условного выражения if-then). В результате расщепления будут получены три цели, одна из которых содержит выходную переменную. Пример этой цели с полученной подстановкой {y←div(i-j,j)+1, z←mod(i-j,j)} добавляется в таблицу:

	(i= (div(i-j,j)+1)*j+mod(i-j,j)) and (mod(i-j,j)>=0) and (mod(i-j,j)<j)	div(i-j,j) +1	mod(i-j,j)

Затем применяем правило эквивалентной замены термов к полученной строке и аксиоме, зафиксированной в пути доказательства на шаге 1, а после этого правило эквивалентной замены выражений для аксиомы, определенной на шаге 2.

шаг 1		i-j=(-1)*j+(1+div(i-j,j))*j+mod(i-j,j) and (mod(i-j,j)>=0) and (mod(i-j,j)<j)	1+ div(i-j,j)	mod(i-j, j)
шаг 2		(i-j =(-1+1+div(i-j,j))*j+mod(i-j,j)) and (mod(i-j,j)>=0) and (mod(i-j,j)<j)	1+ div(i-j,j)	mod(i-j, j)

После выполнения преобразования -1+1==>0 для строки, полученной на шаге 2, мы получим один из примеров гипотезы индукции, резолюция с которой доказывает цель полученную на шаге 2, а в выходных колонках таблицы содержатся рекурсивные обращения к функциям. Заключительный шаг доказательства – резолюция строк, содержащих рекурсивную и нерекурсивную ветви функции:

	not(i<j)	div(i-j,j)+1	mod(i-j,j)
	i<j	0	1

Получена истинная цель:

	true	if i<j then 0 else div(i-j,j)+1	if i<j then i else mod(i-j,j)

На этом процесс доказательства завершается, алгоритм вычисления результатов функций содержится в выходных колонках.

Глава 4. Система синтеза функциональных программ АЛИСА

Синтез функциональных программ методом дедуктивных таблиц с использованием эвристик, описанных в главе 3, реализован в системе АЛИСА. Синтез программ осуществляется по их спецификациям, сформулированным на специальном языке, а синтезированная программа записывается на языке Лисп. В данной главе приводятся основные сведения о системе АЛИСА, о её реализации и результатах работы.

4.1 Язык спецификаций

В качестве языка спецификаций используется язык, основанный на конструкциях логики предикатов первого порядка. Такое представление является достаточно понятным для человека и удобным для проведения дедуктивного синтеза. При описании языка спецификаций мы будем использовать формулы Бэкуса-Наура.

<спецификация>::=**<**заголовки определяемых функций **> <= for <**ограничения**>**

find <выходы функций**> such that** <логическое выражение>

В левой части спецификации (до знака <=) указываются имена функций, которые мы собираемся синтезировать, и для каждой функции – список имён её формальных параметров. Конструкция **for** <ограничение> введена для определения ограничений на входные параметры синтезируемой функции. Если входные параметры не удовлетворяют этим ограничениям, то считается, что функция должна выдавать значение NIL. Примеры спецификаций приведены в параграфе 4.9.

В настоящей версии системы все функции (и синтезируемые, и встроенные) работают только с целыми числами и списками. Понятие списка в данном случае определяется аналогично определению в языке Лисп: список - это заключенная в круглые скобки упорядоченная последовательность элементов,

разделенных пробелами. Элементом списка может быть либо число, либо список (подсписок). Пустой список () имеет еще одно обозначение - **NIL.** Слово NIL зарезервировано в списке служебных слов системы и трактуется как константа.

В ограничениях, записываемых в спецификации, можно указать, какие входные параметры являются числами (с помощью предиката **number**), а какие - списками (используя предикат **list**). В ограничениях можно использовать логические связки not, and, or.

После ключевого слова **find** в спецификации указываются имена переменных, которыми обозначены результаты описываемых функций. Логическое выражение, идущее следом за **such that,** задает ограничения (условия) на эти переменные, то есть на результаты функций. Логическое выражение может содержать связки and, or, not, if-then (логическое следствие), if-then-else. Входящие в выражение переменные могут быть связаны кванторами всеобщности (ForAll) или существования (Exist). Далее в тексте диссертации для краткости мы будем иногда использовать запись кванторов ForAll и Exist в виде $\forall$ и $\exists$, а логические связки not, and, or будем записывать как $\neg$, $\wedge$, $\vee$ соответственно.

В системе задан набор встроенных предикатов и функций, которые используются для записи логических выражений.

Помимо встроенных предикатов и функций в спецификации могут использоваться новые предикаты и ранее синтезированные функции.

Полное формальное описание используемого языка спецификаций приведено в приложении 1.

4.2 Язык синтезируемых программ

Синтезируемые программы записываются на подмножестве языка Лисп – на так называемом "чистом" Лиспе [McCarthy, 1960]. Этот язык является функциональным языком программирования. В нём имеется ограниченный набор функций, позволяющий создавать и видоизменять произвольные списки,

состоящие из атомов и элементов-подсписков, путем добавления нового элемента в начало списка или путем взятия первого элемента ("головы") списка или соответствующего остатка ("хвоста"). Используя предусмотренные в ”чистом” Лиспе условное выражение и рекурсию, уже можно программировать на этом языке. Современные версии Лиспа [Хювенен и Сеппянен, 1990] имеют богатую библиотеку элементарных операций и приобретают принципиально новые возможности, не характерные для функционального программирования: в них появляется возможность менять значения переменных, использовать циклы, переходы и т. д. Мы же будем рассматривать лишь то подмножество Лиспа, которое сохраняет функциональный стиль программирования; оно включает в себя средства определения новых функций языка Лисп, некоторый набор встроенных функций (в том числе арифметические), рекурсию. Циклы, глобальные переменные, а также особые функции (определяемые как nlambda) и функционалы использоваться не будут.

Синтезированная программа представляет собой определение лисповской функции. Поэтому иногда вместо термина "синтез программы" мы будем употреблять термин "синтез функции". В системе предусмотрена возможность одновременного синтеза нескольких функций, результаты которых взаимосвязаны и заданы с помощью одной спецификации, то есть допустима косвенная рекурсия. Таким образом, синтезированная программа может состоять из нескольких определений функций. Однако выделение некоторых повторяющихся конструкций во вспомогательные функции в системе не предусмотрено.

Для определения новых функций в Лиспе используется встроенная функция defun.

<определение функции>::=

(defun <имя> (<параметры>) <вычислимое выражение>)

У синтезированной функции может быть только фиксированное число аргументов. <Параметры> – это имена аргументов функции, <вычислимое выражение> - это лисповское выражение, задающее правило вычисления

функции. Таким выражением может быть атом или обращение к известной функции.

Под атомом понимается T, NIL, целое число или параметр определяемой функции. Атомы T и NIL имеют в Лиспе специальное назначение: T обозначает логическое значение ”истина” (true), а NIL, помимо того, что является обозначением пустого списка, обозначает ”ложь” (false). Интерпретация константы NIL зависит от контекста.

Обращение к функции – это список, состоящий из имени функции, которую надо вычислить, и фактических параметров для нее. Допускается обращение как к встроенным функциям языка, так и рекурсивное обращение к определяемой функции. В рассматриваемом подмножестве Лиспа содержатся следующие встроенные функции: CAR, CDR, CONS, COND, MEMBER, +, -, *, NULL, EQUAL, >, >=, <, <=, <>, = , NUMBERP и LISTP. Все эти функции, за исключением COND, перед началом работы вычисляют значения всех своих аргументов. Вычисление COND рассмотрено далее.

Семантика встроенных функций Лиспа:

(CAR L) – выдает в качестве результата первый элемент непустого списка L.

(CDR L) – для непустого списка L выдает его без первого элемента.

(CONS A B) – результатом является список, полученный из списка B, в начало которого вставлен элемент A.

Результатом арифметической функции (+ , - , *) является результат соответствующей арифметической операции, примененной к значениям аргументов, которые должны быть числами.

Предикат в Лиспе – это функция, значением которой является либо T, либо NIL. Семантика встроенных предикатов:

(MEMBER X L) – предикат, определяющий, является ли X элементом списка L на верхнем уровне.

(NULL X) – предикат, проверяющий, является ли X пустым списком.

(EQUAL A B) – предикат, проверяющий на равенство свои аргументы.

(NUMBERP X) – предикат, проверяющий, является ли X числом.

(LISTP X) – предикат, принимающий значение истина, если X – список.

Результатом функции сравнения (<, <=, >, >=, =, <>) является T или NIL в зависимости от того, выполняется или нет указанное соотношение. Значения аргументов этих функций – целые числа.

Условные предложения на Лиспе записываются в виде обращений к функции COND: (COND (p_1 a_1) (p_2 a_2) … (p_n a_n))

Здесь p_i – предикаты, a_i – выражения. Последовательно проводится вычисление p_i до получения первого истинного, после чего вычисляется выражение a_i, записанное в паре с этим предикатом, и значение a_i объявляется результатом функции. Если же все p_i равны NIL, то значением функции COND будет NIL.

Программа на Лиспе представляет собой последовательность выражений, которые одно за другим вычисляются. Если очередное выражение - это обращение к функции, отличной от встроенной функции defun, то результатом будет ее значение при заданных аргументах, если же это определение новой функции, то она добавится к множеству известных, и к ней можно будет позже обратиться с конкретными параметрами.

Ранее синтезированные функции, которые были сохранены в системе, тоже могут использоваться как известные при синтезе новых функций.

Окончательный текст функции на Лиспе, полученный в результате синтеза с помощью метода дедуктивных таблиц, формируется по выходному терму, соответствующему цели `true`. Для этого внутренние имена системы непосредственно заменяются на соответствующие имена лисповских функций и предикатов (`head`→`CAR`, `tail`→`CDR` и т. п.), оператор `if_then_else` – на функцию `COND`, а из обращений к функциям формируются лисповские выражений. К преобразованному таким образом выходному терму добавляется условие, задаваемое в спецификации после `for`, обращение к встроенной функции `defun`, имя определяемой функции и список ее параметров. Синтезированная функция будет иметь следующий вид:

```
(defun <имя> (<параметры>)
(COND (<Условие> <Выходной терм, переведенный на Лисп>)))
```

4.3 Использование встроенных механизмов языка Пролог для реализации системы

Система АЛИСА была реализована на языке Visual Prolog 5.2 [Адаменко и Кучуков, 2003]. Выбор этого языка объясняется следующими соображениями.

Организация распределения памяти в этой версии Пролога позволяет проводить перебор достаточно большого количества вариантов, так как программа работает с виртуальным стеком, имеющим размер порядка 10 Мб.

Встроенные механизмы Пролога - автоматическое сопоставление и автоматическая организация поиска с возвратом - предоставляют удобный аппарат для работы с дедуктивной таблицей.

Автоматическое сопоставление используется, во-первых, при упрощении выражений. Для выбора подходящего правила упрощения приходится сравнивать заданное выражение с левыми частями правил упрощения, что и реализуется с помощью встроенного механизма сопоставления.

Во-вторых, механизм сопоставления используется при определении идентичности строк дедуктивной таблицы. Это, например, нужно при проверке, не содержится ли в таблице строка, совпадающая с данной (с точностью до имен переменных); такая проверка выполняется для каждой строки перед ее добавлением в таблицу. Для проведения сравнения все переменные обеих строк обозначаются как "x" (специально введенное внутреннее имя) и проводится попытка отождествить их, используя механизм Пролога. Если строки не могут быть сопоставлены таким образом, то они являются различными и дальнейший их анализ проводить не нужно. Если же сопоставление прошло успешно, то необходима дополнительная проверка, проводится попытка унификации строк с помощью алгоритма, реализованного в системе. Этот алгоритм однозначно определяет, являются ли строки аналогичными (с точностью до имен переменных) или нет, но на практике его работа занимает существенно больше времени, чем встроенное сопоставление, поэтому его использование имеет смысл, если нельзя получить результат более быстрыми методами.

В-третьих, механизм сопоставления используется при проведении унификации предложений, но лишь частично. Дело в том, что в используемом алгоритме необходимо не только проверить, унифицируемы ли выражения, но и выделить их наибольший общий унификатор, поэтому алгоритм унификации не может быть полностью заменен встроенным механизмом сопоставления языка Пролог.

Что касается автоматического поиска с возвратом, то он полезен при сопоставлении логических предложений: например, если одно из подвыражений не подходит, можно сделать откат назад и проверить другое. Поиск с возвратом помогает также организовать просмотр утверждений в дедуктивной таблице, перебор дедуктивных и волновых правил.

4.4 Архитектура и схема работы системы

Архитектура системы АЛИСА представлена на рис. 1.

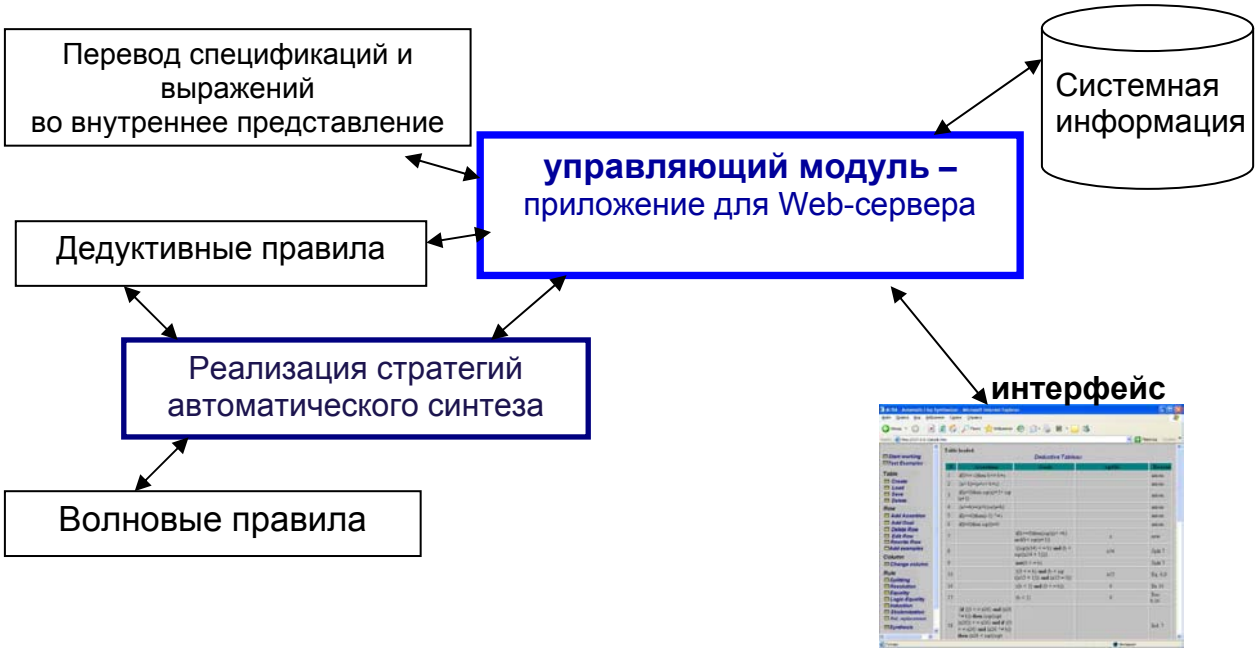


Рисунок 1. Архитектура системы АЛИСА.

Опишем вкратце каждую из частей системы.

Взаимодействие пользователя с системой осуществляется через интерфейс, позволяющий вводить и редактировать спецификации и аксиомы, просматривать дедуктивную таблицу, выбирать правила преобразования для

конкретных строк таблицы, определять новые предикаты. Сформулированный пользователем запрос обрабатывается управляющим модулем. Запрос может содержать спецификацию программы, которую требуется синтезировать, либо задавать преобразование, которое надо выполнить в текущей дедуктивной таблице.

Управляющий модуль взаимодействует со всеми остальными модулями и имеет доступ к системной информации, в которой содержатся известные аксиомы, синтезированные функции, добавленные в систему предикаты, дедуктивная таблица, соответствующая текущему синтезу.

Пользователь задает спецификации в текстовом виде, а система работает с выражениями, хранящимися во внутреннем представлении. В системе существует модуль перевода спецификаций и выражений во внутреннее представление и из такого представления в текстовый вид или в соответствующие конструкции языка Лисп. При осуществлении перевода также проверяется синтаксическая корректность введенного выражения или заданной спецификации.

Модуль, реализующий применение дедуктивных правил, выполняет изменения дедуктивной таблицы согласно дедуктивным правилам, описанным в главе 1, а также ряд вспомогательных действий (унификацию, поиск подвыражений).

В модуле, реализующем работу с волновыми правилами, содержатся функции расстановки аннотаций в заданном выражений, формирования волновых правил, применения волнового правила, проведения доказательства шага индукции с помощью волновых правил, а также вспомогательные функции (поиск аннотированных подвыражений, и их сопоставление).

Стратегия проведения автоматического синтеза задана в специальном модуле. Его задача состоит в проверке состояния дедуктивной таблицы – построена программа или нет, он же задает следующий этап доказательства в случае, если программа еще не построена. Например, подбирает следующий вариант шага индукции, который будет проводиться с использованием волновых правил.

Синтез может проводиться системой как автоматически, так и автоматизированно – с участием человека. При автоматизированном синтезе у пользователя есть возможность загрузить имеющиеся аксиомы и самому ввести новые, задать цель для доказательства и выбрать, какое правило применить к каким строкам таблицы. Результат применения правила отображается на экране, после чего возможны новые преобразования. Один из примеров работы с системой в автоматизированном режиме показан на рис. 2.

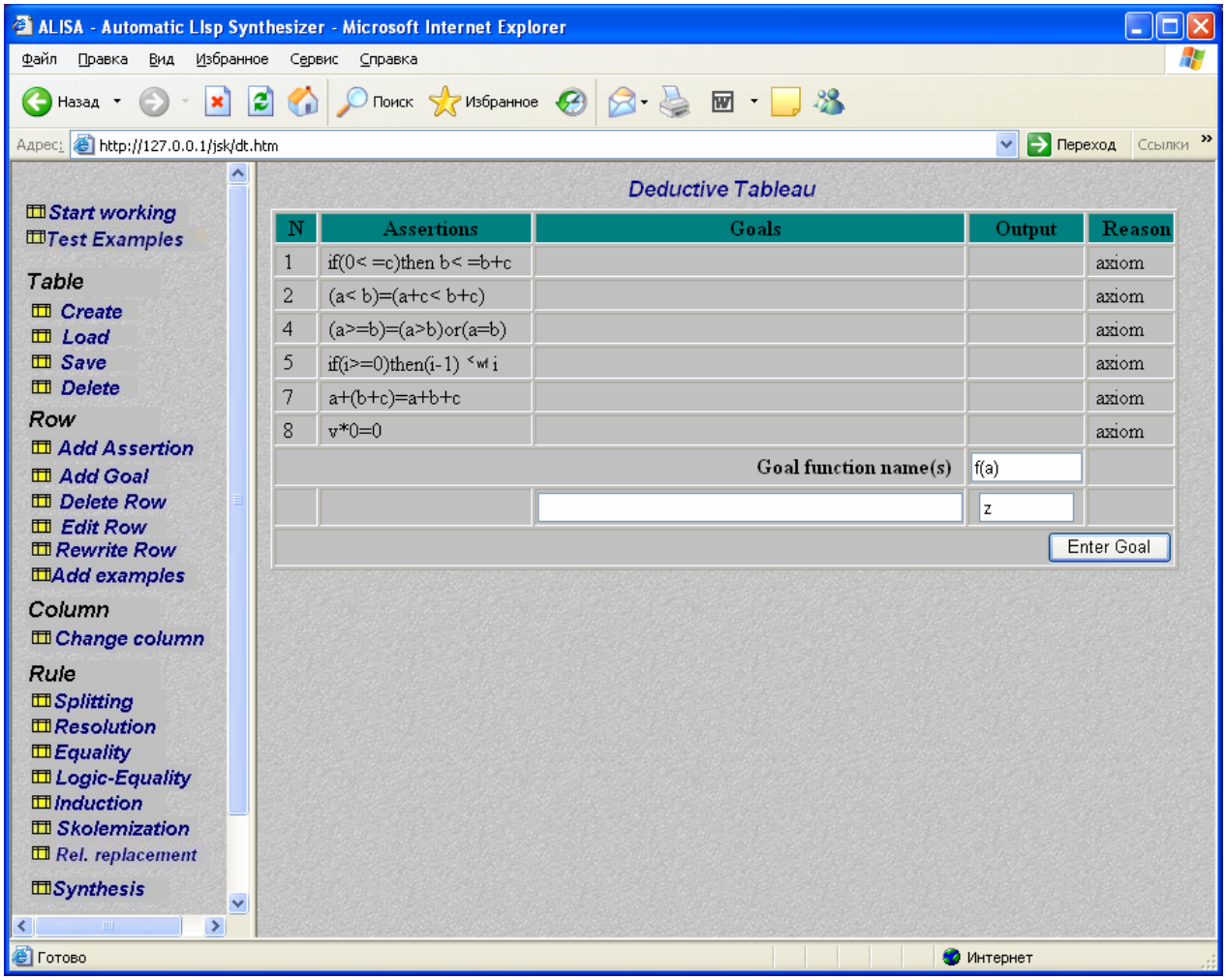


Рисунок 2. Пример работы с системой в автоматизированном режиме.

В автоматическом режиме система предлагает пользователю ввести спецификацию и задать аксиомы, например, загрузить аксиомы из имеющейся базы. Пользователь также выбирает способ проведения доказательства: с использованием волновых правил или без них. Вся введенная информация передается модулю, реализующему стратегии автоматического синтеза. Если

синтез прошел успешно, то система показывает текст полученной функции на Лиспе. После синтеза можно посмотреть последовательность проведенных преобразований, а также пользователь имеет возможность сохранить в системе полученную функцию и использовать ее при дальнейшей работе наряду со встроенными функциями.

4.5 Внутреннее представление дедуктивных таблиц

Разделение предложений в дедуктивной таблице на утверждения и цели нужно лишь для наглядности, в таком виде таблица показывается пользователю. В самой системе работа ведется только с целями (в соответствии со свойством двойственности таблицы, все утверждения переносятся в эту колонку с отрицанием). Однако дополнительно введены еще две колонки: одна содержит порядковый номер строки таблицы (это поле является ключом при работе с таблицей как с базой данных), а вторая – указывает способ получения этой строки (new – исходная цель, axiom – заданная в системе аксиома, Split N – строка получена в результате применения правила расщепления к строке N, и т.д.), чтобы пользователь мог проанализировать цепочку примененных правил. Во внутреннем представлении системы таблица, таким образом, имеет следующую структуру:

№	Goals	Output(s)	Reason
1	G	z	new

В первой колонке стоит номер строки, во второй – логическое выражение (которое записано в колонке Goals таблицы), в третьей – список выходных переменных или термов, а в четвертой – способ получения строки в таблице. Такая строка представлена в базе данных Пролога как факт вида: `g(1,G,[var("z",unknown_)],"new").`

Некоторые строки таблицы хранятся дополнительно в текстовом представлении. В него переводятся те строки, которые нужно будет отобразить на экране, например те, которые входят в окончательное доказательство.

Перед добавлением любой новой строки в таблицу всем ее свободным переменным даются новые имена, поэтому проблемы совпадения имен в системе не возникает. Новые имена переменных создаются динамически. Входные переменные, согласно требованиям метода дедуктивных таблиц, объявляются константами. Этот факт учитывается при их внутреннем представлении: так, `var("a", int_(0))` обозначает целочисленную переменную `a`, а `const("a", int_(0))` – константу с таким же именем (подробнее внутреннее представление выражений описано в приложении 3). Затем строка приводится к дизъюнктивной форме, а в случае, если в строке присутствуют кванторы существования, то они удаляются (так как свободные переменные колонки целей неявно подразумеваются связанными квантором существования), чтобы упростить дальнейший анализ строки и перебор ее подвыражений при применении дедуктивных правил.

После этого проводятся преобразования упрощения (если рассматриваемая строка добавляется не при построении доказательства по найденному пути). При этом, например, проводятся возможные вычисления для арифметических операций: `a+1+1` будет преобразовано в `a+2`, `x*0` преобразуется в `0`, раскрываются все возможные скобки `(a+b)*c ==> a*c + b*c`. (Полный список встроенных преобразований, проводимый системой перед добавлением строки в таблицу, приведен в приложении 2). Отметим, однако, что упрощение строк, полученных при построении доказательства по найденному пути, не проводится, так как при этом могут быть упрощены выражения, полученные в результате специальных подстановок (например, `{i←i-j+j}` в примере, описанном в п. 3.2.4), а их преобразовывать не нужно. После всех этих преобразований строка записывается в прологовскую базу данных, представляющую дедуктивную таблицу.

4.6 Реализация дедуктивных правил

Как уже отмечено, во внутреннем представлении дедуктивная таблица не содержит колонки утверждений, поэтому все приведенные в главе 1 дедуктивные правила реализованы в системе АЛИСА только для работы с целями.

Для применения дедуктивных правил требуется проведение унификации предложений. В системе используется расширенный алгоритм унификации, при котором учитывается не только структура сопоставляемых термов, но и их тип. В настоящей версии системы используются только два типа данных – списки и целые числа, а также введен тип `unknown_`, объекты которого могут трактоваться и как числа, и как списки. Так, терм типа `int_(i)` может быть унифицирован только с термом, имеющим тип `int_(j)` (где i, j – неотрицательные целые числа) или `unknown_`, а терм типа `list_(i)` может быть унифицирован только с термом, имеющим тип `list_(j)` или `unknown_`.

Определение типа терма происходит в момент ввода пользователем новой строки в дедуктивную таблицу. При этом учитывается информация, заданная в спецификации после ключевого слова **for** (если добавляется выражение из спецификации), информация о применяемых к этому терму функциях и предикатах, аргументом которых он является. Дополнительные ограничения могут быть наложены структурой выражения, в которую входит терм. Например, если добавляемое в таблицу утверждение имеет вид `if A[x] then B[x]`, где `x` – некоторый терм, тип которого отличен от `unknown_`, то тип терма `x` в выражении `B` не может быть больше типа терма `x` в выражении `A`. Если это не так, то типом терма `x` в `B` объявляется `int_(i+1)` (или `list_(i+1)`), где `int_(i)` (соответственно, `list_(i)`) – минимальный тип терма `x` в выражении `A`. Так, например, при добавлении аксиомы

```
if not(emptylist(x)) and emptylist(tail(x)) then ord(x)
```

в колонку утверждений таблицы типом терма x в выражении `ord(x)` будет `list_(2)`, так как имеется вхождение `x` типа `list_(1)` в условие (в

выражении `emptylist(tail(x))`. При таком способе определения типов учитывается факт, что с помощью конструкции `if A then B` задается порядок вычисления выражений, и условие `B` не должно появиться в выходной колонке раньше условия `A`. Заметим, что при добавлении цели в таблицу подобного дополнительного ограничения не требуется, так как такая цель представляет собой дизъюнкцию, для доказательства которой требуется установить истинность хотя бы одной из подцелей – дизъюнктов, то есть дизъюнкты рассматриваются независимо, в разных строках таблицы, поэтому типы их внутренних термов между собой не связаны.

Другое расширение унификации, используемое в системе, заключается в использовании свойств встроенных функций при унификации. Примерами таких свойств являются коммутативность сложения и умножения, конъюнкции и дизъюнкции. Так, например, выражения `head(a)+1` и `1+head(a)` не могут быть унифицированы с помощью обычного алгоритма унификации, однако учитывая коммутативность сложения, унификация таких термов становится возможной. Отметим, что встроенные свойства арифметических операций могут быть заданы с помощью аксиом, например

№	Goals	Output(s)	Reason
N	¬(x+y=y+x)	z	new

Однако в этом случае требуется дополнительное применение дедуктивных правил. Поэтому встраивание отдельной проверки таких свойств в алгоритм унификации позволяет сократить перебор. Список встроенных свойств приведен в приложении 2.

При синтезе рекурсивных программ применяется правило индукции. Конкретный вариант гипотезы индукции формируется при выборе конкретного wf-отношения. В качестве такого отношения может быть взято одно из существующих отношений, либо новое wf-отношение может быть добавлено пользователем. Wf-отношения задаются в системе в виде аксиом и добавляются в систему так же, как и обычные аксиомы. Для синтеза приведенных в диссертации функций использовались следующие wf-отношения:

9	if(i>=0)then(i-1) $<_{wf}$ i			axiom
10	if(i>=0)and(k>0)and (k<=i) then i-k $<_{wf}$ i			axiom
23	if not(x=NIL)then tail(x) $<_{wf}$ x			axiom
27	if perm(append(w,addfirst(u,x)),y)then append(w,x) $<_{wf}$ y			axiom

В строке 27 использована встроенная функция `addfirst(u,x)`, возвращающая в качестве результата список x, к которому добавлен первым еще один элемент – u, и функция `append`, возвращающая результат конкатенации двух заданных списков.

Например, наличие в системе аксиомы 23 позволяет при синтезе функции `f(a)` сформировать для цели `Q[a,z]` гипотезу индукции `if ¬emptylist(a) then Q[tail(a),f(tail(a))]`.

Эксперименты, проведенные с системой, показали, что для проведения синтеза функции, которая не только формально соответствует спецификации, но и может быть вычислена, необходимы дополнительные ограничения при работе со скулемовскими функциями. Общее ограничение следующее: скулемовские функции не должны появляться в выходной колонке, так как правила их вычисления неизвестны. Формально такие функции в выходной колонке выводят содержащий их терм за множество вычислимых термов.

В некоторых случаях возникновения скулемовских функций в дедуктивной таблице можно избежать, например, в случае, если в спецификации содержится переменная, отличная от выходной, связанная квантором существования. В такой ситуации мы проводим вместе с синтезом заданной функции синтез еще одной новой функции, рассматривая эту переменную как результат новой функции, и явные кванторы при проведении доказательства не появляются. Так, например, одновременно с синтезом функции mod (см. приложение 4) имеет смысл синтезировать функцию div, результатом которой будет свободная переменная из спецификации.

4.7 Стратегия применения дедуктивных правил

Порядок применения дедуктивных правил определяется следующим образом. Сначала система АЛИСА пытается построить вывод функции без использования гипотезы индукции. Для этого к исходной цели и заданным аксиомам по 2 раза применяем все дедуктивные правила (кроме правила индукции). На практике этого бывает достаточно, чтобы синтезировать нерекурсивную функцию, если это возможно. Если в результате в дедуктивной таблице будет получена строка, содержащая тождественно истинную цель, то синтез функции успешно завершается.

Если без применения правила индукции функцию синтезировать не удалось (тождественно истинная цель не найдена в дедуктивной таблице), система пробует найти подходящее wf-отношение. Wf-отношения записаны в системе как аксиомы вида `if C then x<`$_{wf}$ `a` и для использования отношения при формировании гипотезы требуется доказать условие C. Из доказательства этого условия (или условий, если в формировании функции будут участвовать несколько wf-отношений) формируется нерекурсивная ветвь (ветви) функции. Например, для использовании wf-отношения `tail(x)` $<_{wf}$ `x`, заданного аксиомой `if ¬emptylist(a) then tail(x)` $<_{wf}$ `x`, требуется доказать условие `¬emptylist(x)`. Если в таблице уже присутствует цель N

N	emptylist(a)	t	…

которая служит доказательством этого условия (иначе говоря, случай пустого списка оказывается уже рассмотренным в строке N), и будет построено доказательство шага индукции для гипотезы, сформированной с помощью рассматриваемого wf-отношения, то выход строки N будет являться нерекурсивной ветвью синтезируемой функции, а цель этой строки – условием нерекурсивной ветви.

Если на данном этапе синтеза не удается доказать ни одно из условий известных wf-отношений, то к строкам таблицы применяются одно за другим дедуктивные правила. После того, как некоторое правило применилось, снова производится поиск wf-отношения, условие которого можно доказать.

Формально процесс поиска может продолжаться до тех пор, пока применение правил добавляет какие-либо новые строки в таблицу. Но на практике хватает 2-3 проходов по всем правилам.

После того, как выбрана подходящая гипотеза индукции, шаг индукции доказывается с помощью волновых правил, как это было описано в главе 3. Если доказательство завершено успешно (в результате переписывания был получен пример гипотезы, а волновой фронт либо пуст, либо выражение, содержащееся в нем, может быть доказано в рассматриваемой дедуктивной таблице), то его результатом будет путь доказательства и подстановка для выходной переменной. Если доказательство завершилось неуспехом, то происходит возврат назад и снова ищется подходящее wf-отношение, и т.д.

При применении любого дедуктивного правила информация о попытке его применения сохраняется, поэтому одно и тоже правило не будет применено к одним и тем же строкам таблицы дважды.

После того, как рекурсивная ветвь (ветви) будет построена, применение правила резолюции с нерекурсивными ветвями функции завершает доказательство.

4.8 Реализация волновых правил

В систему АЛИСА входит модуль, реализующий работу с волновыми правилами. В его функции входит решение следующих задач: формирование волновых правил, аннотирование заключения индукции, доказательство шага индукции с использованием волновых правил при построении рекурсивной ветви функции (согласно алгоритму, описанному в главе 3). Рассмотрим подробнее особенности работы с волновыми правилами.

В системе АЛИСА волновые правила хранятся как факты в прологовской базе данных. Внутреннее представление волновых правил выглядит следующим образом

```
<волновое правило> ::= rw_rule(<номер правила>,
    <условие применения>, <полярность>,
```

<левая часть правила>, <правая часть правила>,

<соответствующее дедуктивное правило>,

<номер соответствующей аксиомы>)

Номер правила служит ключом для доступа к правилу. Условие применения представляет собой логическое выражение, истинность которого должна быть проверена перед применением правила. Отдельно указывается полярность подвыражения, которое может быть переписано согласно этому правилу. Для применения правила необходимо, чтобы его условие было выполнено, а заменяемое подвыражение имело указанную полярность. В правиле может быть записано пустое условие (означающее, что никаких проверок проводить не нужно), полярность тоже может быть не задана (это означает, что это условие полярности учитывать не нужно). Левая и правая части правил представляют собой термы или логические выражения, к которым добавлены аннотации.

Аннотации представлены в системе как мета-функции wf, wfin, wfout, wh, nw. К каждому подвыражению и терму аннотируемого предложения добавляется одна из таких мета-функций, означающих следующее:

wf(X) - X находится внутри волнового фронта (направление волнового фронта не определено);

wfin(X) - X содержится в волновом фронте, направленном внутрь;

wfout(X) – X содержится в волновом фронте, направленном наружу;

wh(X) – X находится в волновой дыре;

nw(X) – выражение X не входит ни в какой волновой фронт.

В системе рассматриваются только волновые фронты, содержащие один терм. Если по смыслу волновой фронт должен окружить несколько вложенных термов, то появляется несколько вложенных волновых фронтов. Приведем пример представления аннотаций: выражение $(X+Y)^{\uparrow} + Z$ записывается как nw(wfout(wh(X)+wfout(Y)) +nw(Z)).

Процесс доказательства шага индукции с использованием волновых правил начинается с расстановки волновых фронтов в заключении индукции. Для автоматического решения этой задачи в системе реализована функция

annotate_concl, которая получает в качестве входных параметров гипотезу и заключение индукции, а возвращает аннотированное заключение.

Затем к аннотированному заключению по очереди начинают применяться волновые правила. Этот процесс продолжается до тех пор, пока либо не будет получен пример гипотезы индукции в заключении (эта ситуация считается успешным завершением доказательства), либо ни одно из правил нельзя будет применить. В последнем случае предпринимается попытка вернуться назад на тот шаг доказательства, где была возможность применения нескольких правил, и выбрать другой вариант.

Согласно найденному пути проводятся преобразования строки дедуктивной таблицы, содержащей заключение индукции. На завершающем этапе доказательства применяется правило резолюции для формирования текста функции из рекурсивной и нерекурсивной ветвей.

4.9 Результаты синтеза в системе АЛИСА

Описанный метод автоматического синтеза был реализован в системе АЛИСА и использован для синтеза некоторых функций. Наиболее простые функции, алгоритм вычисления значения которых фактически явно был задан в спецификации, синтезировались без использования правила индукции. Ниже приведены два примера спецификаций таких функций и результаты синтеза.

1. Спецификация функция конкатенации двух заданных списков:

```
<append(l,k)><=for list(l) and list(k) find <z> such that
    if emptylist(l) then z=k
     else z=addfirst(head(l),append(tail(l),k))
```

Результат синтеза:

```
(DEFUN append(l1,l2) (COND ((AND (LISTP l1) (LISTP l2) )
 (COND ((NULL l1) l2) (T (CONS (CAR l1) (append (CDR l1) l2)))))))
```

2. Спецификация функции вычисления квадрата заданного числа:

```
<sqr(b)> <= for number(b) find <z> such that z=b*b
```

Результат синтеза:

```
(DEFUN sqr(b) (COND ((NUMBERP b) (* b b))))
```

Эти функции являются не очень показательными для автоматического синтеза, но полезными, так как они использовались в дальнейшем для синтеза других функций.

Результаты синтеза программ, для построения которых требовалось проводить доказательство по индукции, приведены в таблице 1. В этой таблице для простых примеров приведено сравнение количества шагов синтеза с использованием эвристик и синтеза с выполнением полного перебора, при котором все возможные дедуктивные правила применялись ко всем строкам таблицы по очереди до получения тождественно истинной цели. Также в таблице указано минимально возможное количество шагов, за которое можно построить заданную функцию.

На простых примерах видно, что применение волновых правил позволяет сократить перебор при доказательстве на два порядка. Синтезировать за приемлемое время более сложные функции не представляется возможным без использования эвристик, поэтому количество правил, примененных для синтеза функции sort при полном переборе вариантов, было оценено только теоретически. Наиболее интересным синтезированным примером является программа сортировки, так как в известной литературе встречались только описания дедуктивного синтеза такой программы "вручную". При построении функции sort потребовалось использовать несколько различных гипотез индукции (и сама функция содержит несколько рекурсивных веток), поэтому функция не подходила под заранее заданные схемы программ (которые использовались в системе Periwinkle [Kraan et al., 1996]). Автоматический синтез методом дедуктивных таблиц был невозможен без дополнительных эвристик, направляющих доказательство, так как порождал огромный перебор вариантов. Использование волновых правил для планирования доказательства позволило сдать этот синтез направленным и получить текст функции sort автоматически за 4947 шагов.

Спецификации и синтезированные функции	Количество примененных правил		
	полный перебор	с использованием эвристик	кратчайший путь
<div(i,j),mod(i,j)><=for number(i) and number(j) find <y,z> such that if (i≥ 0) and (j>0) then (i=y*j+z) and (z ≥ 0) and (z<j) Вычисление результата деления нацело и остатка **(DEFUN div(i,j) (COND** ((**AND** (**NUMBERP** i) (**NUMBERP** j)) **(COND** ((<= j i) (+ (**div** (- i j) j) 1)) (**T** 0))))) **(DEFUN mod(i,j) (COND** ((**AND** (**NUMBERP** i) (**NUMBERP** j)) **(COND** ((<= j i) (**mod** (- i j) j)) (**T** i)))))	30300	39	20
<front(a),last(a)><= for list(a) find <h,t> such that if ¬(a =()) then (tail(t)=()) and (a=append(h,t) Разделение списка на "начало" и последний элемент **(DEFUN front(a)(COND ((LISTP** a)**(COND ((NULL a) NIL)(T (CONS (HEAD** a) **(front(TAIL** a))))))) **(DEFUN last(a)(COND ((LISTP** a)**(COND ((NULL tail**(a)) **a)(T (last (TAIL** a))))))	7200	28	15
<sqrt(b)> <= for number (b) find <z> such that if b≥ 0 then (sqr(z)<=b) and (b<sqr(z+1)) and (z ≥ 0) Целочисленный квадратный корень (**DEFUN sqrt(b) (COND** ((**NUMBERP** b) (**COND** ((< b 1) 0) (**T** (**COND** ((<= (**sqr** (+ (**sqrt** (+ b -1)) 1)) b) (+ (**sqrt** (+ b -1)) 1)) (**T** (**sqrt** (+ b -1))))))))	31000	76	19
<sort(b)> <= for list(b) find <z> such that perm(b,z) and ord(z) Сортировка заданного списка (**DEFUN sort(b) (COND** ((**LISTP** b) (**COND** ((**NULL** b) b) (**T** (**COND** ((**NULL** (**sort** (**CDR** b))) b) (**T** (**COND** ((<= (**CAR** (**sort** (**CDR** b))) (**CAR** b)) (**CONS** (**CAR** (**sort** (**CDR** b))) (**sort** (**CONS** (**CAR** b) (**CDR** (**sort** (**CDR** b))))))) (**T** (**COND** ((**NULL** (**CDR** b)) b) (**T** (**CONS** (**CAR** b) (**sort** (**CDR** b))))))))))))	> 1000000	4947	56

Таблица 1. Некоторые результаты синтеза.

Следует отметить, что в диссертации не ставилась задача анализа эффективности построенного алгоритма или поиска различных программ,

соответствующих заданной спецификации. Система прекращает свою работу, как только удалось построить какую-либо программу, удовлетворяющую заданной спецификации.

Заключение

Основные результаты диссертационной работы состоят в следующем:

- Предложен новый метод синтеза функциональных программ по их формальным спецификациям, позволивший расширить возможности автоматического синтеза программ за счёт существенного сокращения перебора.
- На основе предложенного метода разработана и реализована программная система автоматического синтеза функциональных программ.

Эксперименты с системой продемонстрировали достоинства предложенного метода и возможность синтеза за приемлемое время функциональных программ, дедуктивный синтез которых до этого удавалось реализовать только вручную.

Литература

[Адаменко и Кучуков, 2003] Адаменко А., Кучуков А. Логическое программирование и Visual Prolog. – Санкт-Петербург: БХВ-Петербург, 2003.

[Большакова и Мальковский, 1987] Большакова Е.И., Мальковский М.Г. Автоматический синтез программ. – М.: МГУ, 1987.

[Братко, 1990] Братко И. Программирование на языке Пролог для искусственного интеллекта, М.: Мир, 1990.

[Кахро и др., 1981] Кахро М.И., Калья А.П., Тыугу Э.Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ). – М.: Финансы и статистика, 1981.

[Тыугу, 1984] Тыугу Э.Х. Концептуальное программирование. – М.: Наука, 1984.

[Хювенен и Сеппянен, 1990] Хювенен Э., Сеппянен Й. Мир Лиспа. В 2 томах. – М.: Мир, 1990 – т.1-2.

[Чень и Ли, 1983] Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. – М.: Наука, 1983.

[Юэ и Оппен, 1991] Юэ Ж., Оппен Д. Равенства и правила переписывания. Обзор.// В сборнике статей: «Математическая логика в программировании» перевод с англ. (ред. М.В.Захарьящев и Ю.И.Янов) – М.: Мир, 1991, с. 176 – 232.

[Armando et al., 1999] Armando, A. Smaill, A., Green, I.: Automatic Synthesis of Recursive Programs: The Proof-Planning Paradigm. // Automated Software Engineering, vol.6, 1999, p. 329-356.

[Ayari and Basin, 2001] Ayari, A., Basin D.: A Higher-Order Interpretation of Deductive Tableau.// Journal of Symbolic Computation, 31 (5), (2001), p. 487-520

[Baader and Nipkow, 1999] Baader F., Nipkow T.: Term Rewriting and All That Cambridge University press, 1999

[Basin and Walsh, 1992] Basin D., Walsh T. Difference Matching. //Lecture Notes in Computer Science, vol.607, Springer, 1992, p. 295-309

[Basin and Walsh, 1993] Basin D., Walsh T. Difference Unification // – Ruzena Bajcsy (Ed.): Proceedings of the 13th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1993, p. 116-122

[Basin and Walsh, 1996] Basin, D., Walsh, T.: A Calculus for and Termination of Rippling.// Journal of Automated Reasoning, vol. 16 (1996) 147-180

[Boyer and Moore, 1979] Boyer R.S., Moore J.S. A Computational Logic. -- New York: Academic Press, 1979.

[Bundy et al., 1990] Bundy, A., van Hamerlen, F., Horn C., Smaill, A. The Oyster-Clam system. // Lecture Notes in Computer Science, vol. 449, Springer, 1990, p. 647-648.

[Bundy et al., 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A. and Smaill, A. Rippling: A Heuristic for Guiding Inductive Proofs.// Artificial Intelligence, vol. 62, 1993, p. 185-253.

[Bundy, 1999] Bundy A. A Survey of Automated Deduction. // Lecture Notes in Computer Science, vol.1600, Springer, 1999, pages 153-174.

[Bundy, 2001] Bundy, A.: The Automation of Proof by Mathematical Induction. // Handbook of automated reasoning, vol.1, Elsevier Science Publishers B.V, 2001.

[Bundy, 2005] Bundy A., Basin D., Hutter D., Ireland A. Rippling: Meta-level Guidance for Mathematical Reasoning. Cambridge University Press, 2005

[Burback et al., 1990] Burback Ron, Manna Z., Waldinger R. et al. Using the Deductive Tableau System. MacIntosh Educational Software Collection, Chariot Software Group, 1990

[Constable et al., 1986] Robert L. Constable, Stuart F. Allen, H. M. Bromley, Walter Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith Implementing Mathematics with the Nuprl Proof Development system. Prentice Hall, 1986.

[Dershowitz and Jouannaud, 1990] Dershowitz N., Jouannaud J.-P. Rewrite Systems. In Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics. – North-Holland Amsterdam, 1990.

[Harper, 1989] Harper R. Introduction to Standard ML. – University of Edinburgh, Laboratory for Foundation of Computer Science, revised edition, 1989.

[Hutter and Sengler, 1996] Hutter D., Sengler C. INKA: the next generation. // In McRobbie, M.A. and Slaney, J.K., (eds.), 13th International Conference on Automated Deduction, Springer-Verlag, 1996, p. 288 – 292.

[Kraan et al., 1996] Kraan, I., Basin, D., Bundy, A.: Middle-Out Reasoning for Synthesis and Induction. Journal of Automated Reasoning, vol.16, 1996, p. 113-145

[Kreitz, 1998] Kreitz, C. Program Synthesis //Chapter III.2.5 of Automated Deduction – A Basis for Applications, Kluwer, 1998, p. 105-134.

[Lacey et al., 2000] Lacey, D., Richardson, J., Smaill, A. Logic Program Synthesis in a Higher-Order Setting.// Proceedings of the First International Conference On Computational Logic (CL2000) LNAI, vol.1861, Springer-Verlag Berlin Heidelberg, 2000, p. 912-925.

[Lee et al., 1974] Lee R.C.T., Chang C.L., Waldinger R.J. An improved program-synthesizing algorithm and its correctness.// Communications of ACM Vol.17(4), 1974, p. 211-217.

[Manna and Waldinger, 1980] Manna, Z., Waldinger R. A deductive approach to program synthesis.// ACM Transactions. Programming languages and systems 2(1), 1980, p. 91-121.

[Manna and Waldinger, 1992] Manna, Z., Waldinger, R. Fundamentals of Deductive Program Synthesis.// IEEE Transactions on Software Engineering, vol.18(8), 1992, p. 674-704.

[Manna and Waldinger, 1993] Manna, Z., Waldinger, R. Deductive Foundations of Computer Programming. Addison-Wesley, 1993.

[McCarthy, 1960] McCarthy, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine.// Communications of ACM, Vol.3, N 4, 1960, p. 184-193

[Nardi, 1989] Nardi, D. Formal Synthesis of a Unification Algorithm by the Deductive Tableau Method. // Journal of Logic Programming 7, 1989, 1-43

[Paulson, 1987] L. Paulson, Logic and Computation: Interactive Proof with Cambridge LCF. // Cambridge Tracts in Theoretical Computer Science 2, Cambridge University Press, 1987

[Paulson, 1988] Paulson L.C. A preliminary user's manual for Isabelle. Technical report 133, Computer Laboratory, University of Cambridge, 1988

[Paulson, 1989] Paulson L.C. The Foundation of a Generic Theorem Prover. Journal of Automated Reasoning, vol.5, 1989, p. 363-397.

[Pientka and Kreitz, 1999] Pientka, B., Kreitz, C. Automating inductive specification proofs. // Fundamenta Informatica, vol. 39(1-2), IOS Press, 1999, p. 181-209.

[Richardson, 2002] Richardson, J.D.C. Proof Planning and Program Synthesis: A Survey. // Proceedings of AAAI'2002 Stanford, California, 2002

[Robinson, 1965] Robinson, J.A. A machine-oriented logic based on resolution principle. // Journal of the ACM, 12, 1965, p. 23-41.

[Traugott, 1989] Traugott, J.: Deductive Synthesis of Sorting Programs. // Journal of Symbolic Computation, vol.7, 1989, p. 533-572

[Waldinger and Lee, 1969] Waldinger, R., Lee, R. PROW: A step towards automatic program writing. //1st International Joint Conference on Artificial intelligence, Morgan Kaufman, 1969, p.241-252.

[Корухова и Пильщиков, 2002] Корухова Ю.С., Пильщиков В.Н. Система дедуктивного синтеза программ. // Научно-теоретический журнал искусственный интеллект, № 2 (ISSN 1561-5359), Донецк, Наука і освіта, 2002, с. 451-459.

[Корухова и Пильщиков, 2002а] Корухова Ю.С., Пильщиков В.Н. Система дедуктивного синтеза программ. // Международная научная конференция "Интеллектуализация обработки информации" ИОИ-2002, тезисы докладов, Симферополь, 2002, с.124-125.

[Korukhova, 2004] Korukhova Y.S. Planning Proof in the Deductive Tableau Using Rippling. // Proceedings of the 5th International Conference on Recent Advances in Soft Computing (RASC'2004), Nottingham Trent University, 2004, p. 384-390

[Korukhova, 2005] Korukhova Y.S. Automation of Program Synthesis from Logic-Based Specifications in the Deductive Tableau**.** ICCL Workshop on Logic-Based Knowledge Representation, TU Dresden, 2005
www.computational-logic.org/content/events/iccl-ss-2005/talks/YuliaKorukhova.pdf

[Korukhova, 2005a] Korukhova Y.S. An Approach to Automation of Program Synthesis in the Deductive Tableau. // Proceedings of the 10th ESSLLI Student Session, Heriot-Watt University, Edinburgh, 2005, p. 122-133 http://www.sissa.it/~gervain/ProceedingsFinalVersion.pdf

Приложение 1. Описание языка спецификаций, используемого в системе АЛИСА

В данном приложении приводится полное описание языка, на котором записываются спецификации синтезируемых функций.

<спецификация> ::= **<**заголовки определяемых функций **> <=**

for <ограничения**> find <**выходы функций**> such that** <логическое выражение>

В левой части спецификации (до знака <=) указывается имена функций, которые мы собираемся синтезировать, и для каждой функции – список имен ее формальных параметров.

<заголовки определяемых функций **> ::=**

<имя>(<параметры>){,<имя>(<параметры>)}

<параметры> ::=<имя> {, <имя>}

<выходы функций> ::=<имя> {, <имя>}

<имя> ::= <идентификатор>

В качестве примера приведем несколько заголовков функций:

<append(x,y)> - требуется синтезировать функцию с названием append, которая имеет два параметра.

<div(i,j),mod(i,j)> - требуется получить две функции div и mod, обе они имеют по два параметра. Функции могут зависеть как от одних и тех же параметров, так и от разных.

Конструкция **for** <ограничение> введена в язык для определения ограничений на входные параметры синтезируемой функции.

<ограничения> ::= <ограничение> (<имя>) | <ограничения> **and** <ограничения>

| <ограничения> **or** <ограничения> | **not** (<ограничения>)

<ограничение> ::= **list** | **number**

В ограничениях можно указать, какие параметры синтезируемых функций являются числами, используя предикат number, а какие - списками list. Пример ограничений: number(B) and list(L)

После ключевого слова **find** в спецификации указаны имена переменных, которыми обозначены результаты описываемых функций (они записываются в том же порядке, что и имена функций, к которым они относятся). Логическое выражение, идущее следом за **such that,** задает ограничения (условия) на эти переменные, то есть на результаты функций. Логическое выражение выглядит следующим образом:

<логическое выражение > ::= <предикат> | <отношение> | <if-выражение> |
<квантор><переменная> <логическое выражение> |
< логическое выражение > **and** < логическое выражение > |
< логическое выражение > **or** < логическое выражение> |
not < логическое выражение >

<квантор>::= **ForAll** | **Exist**

<if-выражение>::= **if** <логическое выражение> **then** <логическое выражение> |
if <логическое выражение> **then** <логическое выражение> **else**
<логическое выражение>

<предикат> ::= <имя встроенного предиката>(<параметры>)

<отношение> ::=
<числовое выражение><операция сравнения><числовое выражение> |
<логическое выражение> = <логическое выражение>

<числовое выражение> ::= <функция> | <константа> | (<числовое выражение>)

<константа> ::= <целое число> | <имя входной переменной>

<операция сравнения> ::= <= | >= | < | > | <> | =

<функция > ::= <имя функции>(<параметры>) |
<числовое выражение><арифметическая операция><числовое выражение>

<арифметическая операция> ::= + | - | *

<имя функции> ::= <имя синтезируемой функции> |
<имя встроенной функции> |
<имя ранее синтезированной функции>

Для записи спецификаций используются следующие предикаты и функции, которые заранее встроены в систему АЛИСА.

Предикаты:

number(*x*) – истина для *x*, являющегося целым числом,

list(*x*) – истина для *x*, являющегося списком,

emptylist(*x*) – истина для *x*, являющегося пустым списком,

elem(*c,L*) – истина, если *c* является элементом списка *L*,

в остальных случаях значения предикатов - ложь.

Функции:

addfirst(*c,x*) – добавляет элемент *c* первым в список *x*, результат – новый список,

head(*x*) – выдает в качестве результата первый элемент списка *x*,

tail(*x*) – выдает список *x* без первого элемента.

Пример логического выражения:

`if not(emptylist(a)) then a=addfirst(head(a),tail(a))` – любой непустой список состоит из "головы" (первого элемента) и "хвоста" (остальной части списка).

Помимо встроенных предикатов и функций в систему могут быть добавлены новые предикаты и функции, для использования которых нужно задать аксиомы. Синтезированные функции могут быть сохранены в системе и использоваться как известные

Приложение 2. Встроенные знания системы АЛИСА

2.1 Набор аксиом

В данном параграфе приведены встроенные аксиомы системы, а также аксиомы, введенные дополнительно для синтеза некоторых функций. Дополнительные аксиомы были добавлены как утверждения в таблицы axiomsnum и axiomslist, содержащие аксиомы для чисел и списков соответственно. Приведенные таблицы аксиом можно расширять и дальше, а при добавлении любой новой аксиомы могут использоваться не только встроенные, а все известные в системе предикаты и функции.

Для наглядности в текстовом представлении аксиомы записываются в колонке утверждений таблицы. Во внутреннем представлении они содержатся (с добавлением отрицания) в колонке целей. Все свободные переменные в колонке утверждений подразумеваются связанными квантором всеобщности.

Аксиомы, содержащие конструкцию $<_{wf}$, представляют собой запись wf-отношений. Их добавление в систему производится аналогично добавлению обычных аксиом (знак $<_{wf}$ вводится в текстовом виде как <wf).

Аксиомы для списков (таблица **axiomslist**):

N	Assertions	Goals	Output	Reason
1	if not(x=nil)then x=addfirst(head(x),tail(x))			axiom
2	if emptylist(x)then append(x,y)=y			axiom
3	(y1=y2)=(addfirst(x,y1)=addfirst(x,y2))			axiom
4	if x=nil then ord(x)			axiom
5	perm(x,x)			axiom
6	perm(append(y,addfirst(u,z)),addfirst(u,x))=perm(append(y,z),x)			axiom
7	if not(emptylist(x)) and emptylist(tail(x)) then ord(x)			axiom
8	if not(x=nil)and ord(x)and(u< =head(x))then ord(addfirst(u,x))			axiom
9	if not(x=nil)then tail(x) $<_{wf}$ x			axiom
10	if perm(append(w,addfirst(u,x)),y)then append(w,x) $<_{wf}$ y			axiom
11	perm(w,y)=perm(y,w)			axiom
12	perm(addfirst(u,x),addfirst(u,z))=perm(x,z)			axiom
13	addfirst(a1,b1)=append(addfirst(a1,nil),b1)			axiom
14	(y=append(y1,y2))=(addfirst(x,y)=append(addfirst(x,y1),y2))			axiom

15	if ord(addfirst(h,a))and elem(e,a)then h< =e			axiom
16	if ord(a)and(forall e(if elem(e,a)then h< =e))then ord(addfirst(h,a))			axiom
17	elem(d,addfirst(h,t))=((d=h)or elem(d,t))			axiom
18	perm(addfirst(y,addfirst(u,z)),addfirst(u,x))=perm(addfirst(y,z),x)			axiom

Аксиомы для чисел (таблица **axiomsnum**):

N	Assertions	Goals	Output	Reason
1	if(0< =c)then b< =b+c			axiom
2	(a< b)=(a+c< b+c)			axiom
3	if(z>=0)then sqr(z)+1<= sqr(z+1)			axiom
4	(a>=b)=(a>b)or(a=b)			axiom
5	if(i>=0)then(i-1) $<_{wf}$ i			axiom
6	if(t=0)then sqr(t)=0			axiom
7	(a-b=c)=(a=b+c)			axiom
8	v*0=0			axiom
9	if(i>=0)and(k>0)and(k<=i) then i-k $<_{wf}$ i			axiom
10	(y+1)*c=y*c+c			axiom

2.2 Встроенные преобразования термов и логических выражений

При добавлении новой строки в дедуктивную таблицу выполнятся тождественные преобразования, которые упрощают дальнейшую работу с таблицей. Так как свободные переменные в различных строках считаются различными, в новой строке делается подстановка, дающая всем переменным новые имена, которые ранее не встречались в таблице. Эти имена имеют вид xN, где x – фиксированный символ, а N принимает одно за другим целочисленные значения, начиная с 0. В системе хранится информация о последнем использованном имени переменой (например, x12). Тогда в качестве следующего имени будет создано x13, затем x14 и т.д.

Приведенные ниже правила преобразования применяются для всех строк, кроме тех, которые были получены при восстановлении доказательства по найденному пути, так как в этом случае встроенные преобразования могут мешать выполнению специальных подстановок, выполненных при

доказательстве по индукции (например, выражение a-1+1 полученное в результате специальной подстановки в заключении индукции преобразовывать не нужно, а все необходимые преобразования будут проведены явно при помощи дедуктивных правил).

Для преобразования предложений в системе используются следующие тождества (коммутативность и ассоциативность операций конъюнкции, дизъюнкции, сложения и умножения неявно подразумевается):

для логических выражений:

T1: $F \vee \text{false} \equiv F$

T2: $F \vee \text{true} \equiv \text{true}$

T3: $F \wedge \text{false} \equiv \text{false}$

T4: $F \wedge \text{true} \equiv F$

T5: $F \wedge \neg F \equiv \text{false}$

T6: $F \vee \neg F \equiv \text{true}$

T7: $\neg (\neg F) \equiv F$

T8: $\neg (A \wedge B) \equiv \neg A \vee \neg B$

T9: $\neg (A \vee B) \equiv \neg A \wedge \neg B$

T10: $X \wedge (A \vee B) \equiv (X \wedge A) \vee (X \wedge B)$

T11: $A \wedge B \wedge A \equiv A \wedge B$

T12: $A \vee B \vee A \equiv A \vee B$

T13: $\neg \text{true} \equiv \text{false}$

T14: $\neg \text{false} \equiv \text{true}$

T15: $(X = X) \equiv \text{true}$

T16: $(X <> X) \equiv \text{false}$

T17: if A then B else C $\equiv (A \wedge B) \vee (\neg A \wedge C)$

T18: if A then B $\equiv (A \wedge B) \vee \neg A$

T19: if true then B $\equiv$ B

T20: if true then B else C $\equiv$ B

T21: if false then B $\equiv$ true

T22: if false then B else C $\equiv$ C

T23: if A then X else X $\equiv$ X

T24: if A then true $\equiv$ true

T25: if A then false $\equiv \neg A$

T26: $(A \equiv B) \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$

T27: $\neg (A \equiv B) \equiv (\neg A \wedge B) \vee (A \wedge \neg B)$

T28: emptylist(addfirst(x,y)) $\equiv$ false

T29: emptylist(NIL) $\equiv$ true

T30: $A > B \equiv B < A$

T31: $A \geq B \equiv B \leq A$

T32: $A+X < B+X \equiv A < B$

T33: $A+X \leq B+X \equiv A \leq B$

для термов:

T34: if A then "" else t ≡ t (здесь "" – обозначение пустого терма)

T35: if A then s else "" ≡ s

T36: if A then t else t ≡ t

T37: Если в сумму входят два числа (N_1 и N_2), то их сумма вычисляется и становится слагаемым вместо этих двух чисел: $a+\ldots+N_1+\ldots+N_2+\ldots+b \equiv a+\ldots+b+(N_1+N_2)$

T38: a+0 ≡ a

T39: a+N+(-1)*N ≡ a

T40: Если в произведение входят два числа (N_1 и N_2), то их произведение вычисляется и становится множителем вместо этих двух чисел: $a*\ldots*N_1*\ldots*N_2*\ldots*b \equiv a*\ldots*b*(N_1*N_2)$

T41: a*1 ≡ a

T42: (a+b)*c ≡ a*c+b*c

Используя эти тождества, система приводит все выражения в таблице к дизъюнктивной форме, которая представляет собой дизъюнкцию из конъюнкций литералов (атомов и отрицаний атомов). Условные логические выражения (`if A then B else C` или `if A then B`) преобразуются с помощью тождеств T17 и T18 в выражения, содержащие конъюнкцию, дизъюнкцию и отрицание, для вложенных подвыражений применяются правила де Моргана (T8, T9) и закон дистрибутивности T10. Для термов условный оператор остается. Используя тождества T30 и T31, мы приведем выражение к виду, содержащему только операции сравнения $<$ и $\leq$, с учетом ассоциативности + и * и свойства дистрибутивности T42 раскрываются скобки в арифметических выражениях. Тождества T34 и T35 применяются при записи терма в выходной колонки строки, добавляемой в таблицу.

Преобразования введены для того, чтобы к строкам дедуктивной таблицы, содержащим преобразованные выражения, проще было применять правила расщепления и проще проводить поиск подвыражений и внутренних термов при применении дедуктивных правил.

2.3 Используемый алгоритм унификации

Обычный алгоритм унификации двух предложений, может быть записан следующим образом.

Назовем множеством рассогласований непустого множества предложений W множество, получающееся выявлением первой в W позиции (слева), на которой не для всех предложений из W стоит один и тот же символ, и затем выписыванием из каждого предложения W внутреннего предложения, которое начинается с символа, занимающего эту позицию. Множество этих внутренних предложений и есть множество рассогласований в W. Например, для выражений P(a) и P(x), где P – некоторый предикат, множеством рассогласований является {a, x}.

Алгоритм содержит следующие шаги:

Шаг 1. Множество k=0, W_k=W и λ_k={}

Шаг 2. Если W_k – единичный дизъюнкт (либо один терм) то остановка, λ_k – наиболее общий унификатор для W. В противном случае найдем множество D_k рассогласований для W_k.

Шаг 3. Если существуют такие элементы v_k и t_k в D_k, что v_k – переменная, не входящая в t_k, то перейти к шагу 4. В противном случае – остановка, W – не унифицируемо.

Шаг 4. Пусть $\lambda_{k+1}=\lambda_k\{v_k \leftarrow t_k\}$ и $W_{k+1}=W_k\{v_k \leftarrow t_k\}$. Заметим, что $W_{k+1}=W_k\{v_k \leftarrow t_k\}$. Присваиваем k значение k+1 и переходим на шаг 2.

Подробно алгоритм унификации описан в [Чень и Ли, 1983], стр.81. В результате работы алгоритма либо получается наиболее общий унификатор, либо сообщается о невозможности унифицировать заданные предложения. В системе АЛИСА используется расширенный алгоритм, который имеет следующие дополнительные условия:

1. При унификации термов учитывается их тип. Подстановка терма, имеющего тип "список" (то есть list_(N)) не может быть выполнена вместо переменной, тип которой – число (то есть int_(M)) для любых неотрицательных M и N, и наоборот, переменная-список не может быть заменена на терм-число. Других ограничений на типы унифицируемых термов нет.

2. При унификации выражений, содержащих операции сравнения $<$ и $\leq$ учитываются соотношения

T43: A<B ≡ not(B≤A),

T44: A≤B ≡ not(B<A).

То есть если выражения X и Y унифицировать не удалось, а одно их них (например X) содержит операцию сравнения < или ≤ , то это выражение преобразуется согласно тождеству T43 или T44 в выражение X1 и проводится попытка унифицировать X1 и Y. Если она оказалась успешной, то исходное выражение X заменяется на X1 в рассматриваемой строке дедуктивной таблицы и унификация считается успешной. Это свойство позволяет унифицировать, например, выражения `i<j` и `not(k≤i)`, где i,j – константы, k – некоторая переменная. Так как в результате использования тождеств, приведенных выше, операции сравнения <, ≤, >,≥ , были сведены к операциям < и ≤, именно для них и рассматривается расширение алгоритма унификации.

3. Дополнительно при унификации учитывается свойство коммутативности операций конъюнкции, дизъюнкции, сложения и умножения. При унификации предложений X и У, оба из которых содержат как минимум одну из перечисленных операций, рассматриваются поочередно все перестановки аргументов этой операции. Если найдена перестановка, при которой унификация возможна, в соответствующей строке таблицы исходное выражение заменяется на выражение, в котором выполнена найденная перестановка аргументов.

4. При унификации арифметических выражений, содержащих + и * учитывается свойство дистрибутивности. Если термы X и Y не могут быть унифицированы приведенным выше алгоритмом, но один из них (например, X) содержит операции + и *, тогда делается попытка преобразовать его в X1, используя (справа налево) тождество

T42: (a+b)*c ≡ a*c+b*c.

Если найден наибольший общий унификатор для X1 и Y, то и исходные выражения являются унифицируемыми.

Приложение 3. Представление выражений и термов в системе АЛИСА

В данном приложении приводится описание на языке Пролог основных типов предложений, которые используются в системе Алиса при записи предложений в дедуктивной таблице и при применении волновых правил.

Терм:

```
IExpr = reference int(rinteger);var(rstring,VType);const(rstring,VType);
        sk(rstring,IExprList,VType);
        funct(PName,IExprList,VType);
        if_then_else(LExpr,IExpr,IExpr,VType); if_then(LExpr,IExpr,VType)
/*Термом может являться число, переменная, константа, скулемовская функция, обращение к известной или синтезируемой функции, а также условный терм. Для всех термов, кроме чисел хранится информация об их типе. Типом числа считается int_(0) */
```

Тип терма:

```
VType=unknown_;
      int_(rinteger);
      list_(rinteger)
```

Список термов: IExprList = reference IExpr*

Переменные, константы, списки переменных:

```
Variable=var(rstring,VType);const(rstring,VType);sk(rstring,IExprList,VType)
VarList=Variable*
 rinteger = reference integer
 rstring = reference string
```

Имена предикатов и функций:

```
PNAME = p(rstring);/*добавленный пользователем предикат, его имя хранится в файлах с системной информацией*/
          list; number; emptylist; /*встроенные предикаты*/
          less; great; lesseq; greateq; noteq; eq; wfLess; /*отношения*/
          newf(rstring);/*синтезируемая функция*/
          plus;mult;minus/*встроенные арифметические функции*/
```

f(rstring);/*известные функции*/

Логическое выражение:

LExpr = reference true; false; pred(PName,IExprList);

rel(PName,IExpr,IExpr);lrel(PName,LExpr,LExpr);

conj(LExprList);disj(LExprList); neg(LExpr);qex(BVarList,LExpr);

lif_then(LExpr,LExpr);lif_then_else(LExpr,LExpr,LExpr)

LExprList=reference LExpr*

/*Логическое выражение может представлять собой константу true или false, обращение к известному предикату, запись арифметического отношения или операции сравнения (=) для логических выражений, конъюнкцию, дизъюнкцию, отрицание выражения, выражение с кванторами, условное выражение */

Контроль соответствия имен предикатов и функций (функции участвуют в формировании термов, предикаты – в записи логических выражений) также встроен в систему.

Аннотированное предложение:

WExpr = reference wex(WAT, WExpr1)

WAT=wfin;wfout;wf;wh;nw /*волновой фронт*/

WExpr1 = reference true; false;int(rinteger);

var(rstring);/*переменная*/

const(rstring);/*константа*/

sk(rstring,WExprList);/*скулемовская функция*/

wex1(WFunction,WExprList)

WFunction = list; number; emptyList;

newf(rstring);f(rstring);p(rstring);

less; great; lesseq; greateq; noteq; eq;

lnoteq; leq;

plus;minus;mult;

conj;disj;neg;

if_then_else;if_then;lif_then;lif_then_else

/*В отличие от обычных выражений аннотированные выражения не разделены явно на термы и логические выражения, но это можно определить по имени операции (WFunction), записанной в аннотированном выражении.*/

Приложение 4. Синтез функции mod. Удаление скулемовской функции из доказательства

Рассмотрим синтез функции `mod(i,j)`, вычисляющей остаток от деления целого числа `i` на натуральное `j`. Её спецификация задана следующим образом:

```
<mod(i,j)><=for number(i) and number(j) find <z> such that
if (i>=0) and (j>0) then ∃ y ((i=y*j+z) and (z>=0) and(z<j))
```

В спецификации существует переменная y, связанная квантором существования. При добавлении в таблицу исходной цели этот квантор явно можно не записывать, так как выражение добавляется в колонку целей, а в ней все свободные переменные подразумеваются связанными кванторами существования :

	Assertions	Goals	mod(i,j)
G		if (i>=0) and (j>0) then (i=y*j+z) and (z>=0) and (z<j)	z

Применяем к цели G правило расщепления:

G1		not(i>=0)	
G2		not (j>0)	
G3		(i=y*j+z) and (z>=0) and (z<j)	z

Строки G1 и G2 имеют пустые выходные колонки, так как выходная переменная z в их целях не содержится.

Применяем правило эквивалентной замены к G3 и аксиоме A1

A1	0*v=0		

получим строку G4:

G4		(i=0+z) and (z>=0) and (z<j)	z

В строке можно провести упрощение (0+z==>0) и применить правило резолюции к G4 и G1. Получим новую цель G5:

G5		(i<j)	i

Эта цель означает, что если мы докажем, что i<j, то в качестве результата функции следует выдать i.

При формировании гипотезы индукции, в таблице возникает явный квантор ∃ в колонке утверждений:

H	if (i>=j) and (j>0) and (i≥0) then ∃u ((i-j=u*j+mod(i-j,j)) and (mod(i-j,j)>=0) and (mod(i-j,j)<j))		

Перенесем гипотезу в колонку целей

H1		(i-j>=0) and (j>0) and ∀u (not(((i-j=u*j+mod(i-j,j)) and (mod(i-j,j)>=0) and (mod(i-j,j)<j))))	

С учетом описанных в главе 1 преобразований выражений с кванторами (правило сколемизации), заменим переменную `u` на скулемовскую функцию `sk` (без параметров). При появлении скулемовских функций в доказательстве оно осложняется из-за дополнительных проверок: такая функция не должна появляться в выходной колонке, так как правила ее вычисления не известны, и, таким образом, она выводит синтезируемую функцию за класс вычислимых функций. Однако в данном случае возникновения скулемовской функции можно избежать, с помощью введения новой функции, спецификация которой задана одновременно с исходной. Будем считать переменную y результатом новой функции (по смыслу это функция div), спецификация которой задана одновременно со спецификацией функции mod. При одновременном синтезе этих функций рассматривается дедуктивная таблица, содержащая две выходных колонки. Нерекурсивная ветвь данных функций

	Assertions	Goals	div(i,j)	mod(i,j)i
G6		i<j	0	i

получается аналогично нерекурсивной ветви mod. Синтез рекурсивной ветви приведен в главе 3. В результате в дедуктивной таблице будет получена строка

G7		not(i<j)	div(i-j,j)+1	mod(i-j,j)

Применение правила резолюции к G6 и G7 завершает синтез, в выходных колонках полученной строки содержатся алгоритмы вычисления div и mod.

	true	if i<j then 0 else div(i-j,j)+1	if i<j then i else mod(i-j,j)

Приложение 5. Синтез функций front и last

Рассмотрим синтез функций front(s) и last(s), возвращающих для заданного непустого списка его начало – все элементы, кроме последнего, и последний символ соответственно. Один из вариантов синтеза данных функций рассмотрен в [Manna and Waldinger, 1992].

Спецификация функций:

```
<front(s),last(s)> <= for list(s) find <z1,z2> such that
if not(s=NIL) then (tail(z2)=NIL) and s=append(z1,z2)
```

Используемая в спецификации функция `append(x,y)` возвращает строку, являющуюся конкатенацией двух заданных строк `x` и `y`. Синтез этой функции был проведен в системе ранее, и она считается известной.

Рассмотрим два варианта синтеза функций front и last, приводящих к разным результатам. Во втором варианте в результате синтеза получена некорректная функция, так как не учитывались типы аргументов при выводе.

Первый вариант синтеза.

Исходная цель для доказательства – G1:

	Assertions	Goals	front(s)	last(s)
G1		if not(s=NIL) then tail(z2)=NIL and s=append(z1,z2)	z1	z2

Применяем правило расщепления:

G2	not(s=NIL)			
G3		tail(z2)=NIL and s=append(z1,z2)	z1	z2

Применим правило эквивалентной замены к аксиоме A1:

A1	append(NIL,v) =v			

и цели G3:

G4		tail(z2)=NIL and s=z2	NIL	z2

Для полученной цели G4 добавляем пример:

G5		tail(s)=NIL	NIL	s

Смысл строки G5 следующий: если мы докажем, что s - список из одного символа (то есть "хвост" такого списка является пустым), то front(s) = NIL, a last(s)=s. Полученная строка таблицы содержит нерекурсивную ветвь функции.

Теперь будем проводить синтез рекурсивной ветви. Для этого запишем гипотезу индукции по переменной s, используя для непустого списка s wf-отношение $tail(s) <_{wf} s$.

H	if not(tail(s)=NIL) then tail(last(tail(s)))=NIL and tail(s)=append(front(tail(s)),last(tail(s)))			

Применив правило эквивалентной замены к аксиоме

A2	append(addfirst(u,y1),y2)=addfirst(u,append(y1,y2))			

и цели G3, получим цель

G6		tail(y2)=NIL and (s=addfirst(u,append(y1,y2)))	addfirst(u,y1)	y2

Применяем теперь правило замены эквивалентных термов к H и G6 для термов `append(front(tail(s)),last(tail(s))` и `append(y1,y2)`:

G7		tail(last(tail(s))=NIL and (s=addfirst(u,tail(s))) and not(tail(s)=NIL)	addfirst(u, front(tail(s)))	last(tail(s))

Применим правило резолюции к цели G7 и гипотезе индукции H для общего подвыражения `tail(last(tail(s)))=NIL`:

G8		(s=addfirst(u,tail(s))) and not(tail(s)=NIL)	addfirst(u, front(tail(s)))	last(tail(s))

Теперь применим правило резолюции к аксиоме A3

A3	if not(a=NIL) then a=addfirst(head(a),tail(a))			

и цели G8. Получим цель:

G9		not(tail(s)=NIL) and not(s=NIL)	addfirst(head(s), front(tail(s)))	last(tail(s))

Получена строка, содержащая рекурсивную ветвь функции.

Заключительные шаги:

Применим правило резолюции, чтобы "собрать вместе" рекурсивную (G9) и нерекурсивную (G5) ветви функции.

G10		not(s=NIL)	if tail(s)=NIL then NIL else addfirst(head(s), front(tail(s)))	if tail(s)=NIL then s else last(tail(s))

Применение правила резолюции к строкам G2 и G10 позволяет получить заключительную строку доказательства:

G11		true	if tail(s)=NIL then NIL else addfirst(head(s), front(tail(s)))	if tail(s)=NIL then s else last(tail(s))

Алгоритм вычисления результата синтезированных функций содержится в выходных колонках строки G11:

```
front(s) = if tail(s)=NIL then NIL
 else addfirst(head(s), front(tail(s)))
last(s) = if tail(s)=NIL then s else last(tail(s))
```

Второй вариант синтеза

Рассмотрим другой вариант синтеза функций front и last, при котором дедуктивные правила применялись в другой последовательности, и не учитывались типы термов.

Исходная цель для доказательства – G1:

	Assertions	Goals	front(s)	last(s)
G1		if not(s=NIL) then tail(z2)=NIL and s=append(z1,z2)	z1	z2

Нерекурсивную ветвь функций получаем тем же способом, что и в варианте 1:

G5		tail(s)=NIL	NIL	s

Также аналогично варианту 1 получаем цель G6

G6		tail(y2)=NIL and (s=addfirst(u,append(y1,y2)))	addfirst(u,y1)	y2

Для синтеза рекурсивного обращения к функции запишем гипотезу индукции по переменной s, используя для непустого списка s wf-отношение $tail(s)<_{wf} s$.

H	if not(s=NIL) then if not(tail(s)=NIL) then tail(last(tail(s)))=NIL and tail(s)=append(front(tail(s)),last(tail(s)))				

Перенесем гипотезу в колонку целей и применим правило расщепления.

H1		not(s=NIL) and not(tail(s)=NIL) and not(tail(last(tail(s)))=NIL)		
H2		not(s=NIL) and not(tail(s)=NIL) and not(tail(s)=append(front(tail(s)),last(tail(s))))		

Применим правило резолюции к полученным строкам и цели G5 (заметим, что именно на этом шаге условие замены выражения с аргументом минимального типа не выполнено):

G7		not(s=NIL) and not(tail(last(tail(s)))=NIL)	NIL	s
G8		not(s=NIL) and not(tail(s)=append(front(tail(s)),last(tail(s))))	NIL	s

Заключительные шаги:

Применим правило резолюции для строк G6 и G8:

G10		not(s=NIL) and tail(y2)=NIL	if not(tail(s)=append (front(tail(s)),last(tail(s)))) then NIL else addfirst(head(s),front(tail(s)))	if not(tail(s)=append(front(tail(s)),last(tail(s)))) then s else y2

А затем еще раз применим резолюцию к полученной строке G10 и цели G8:

G11		not (s=NIL)	if not(tail(last(tail(s)))=NIL) then NIL else if not(tail(s)=append(front(tail(s)),last(tail(s)))) then NIL else addfirst(head(s),front(tail(s)))	if not(tail(last(tail(s)))=NIL) then s else if not(tail(s)=append (front(tail(s)),last(tail(s)))) then s else tail(last(tail(s)))

Наконец, резолюция полученной цели G11 и G2 дает истинную цель и другие тексты функций в выходных колонках.

G12		true	if not(tail(last(tail(s)))=NIL) then NIL else if not(tail(s)=append(front(tail(s)),last(tail(s)))) then NIL else addfirst(head(s),front(tail(s)))	if not(tail(last(tail(s)))=NIL) then s else if not(tail(s)=append (front(tail(s)),last(tail(s)))) then s else tail(last(tail(s)))

```
front(s) = if not(tail(last(tail(s)))=NIL) then NIL else
if not(tail(s)=append( front(tail(s)),last(tail(s)))) then NIL
else addfirst(head(s),front(tail(s)))
last(s) = if not(tail(last(tail(s)))=NIL) then s else
if not(tail(s)=append (front(tail(s)),last(tail(s)))) then s
else tail(last(tail(s)))
```

При попытке вычислить значения front и last для списка, содержащего один элемент, получаем ошибку, в отличие от функций, синтезированных в первом варианте, которые корректно обрабатывают такой аргумент.

Приложение 6. Синтез функции sort

В данном приложении приведена таблица, полученная в системе АЛИСА при синтезе программы сортировки заданного списка b по спецификации

`<sort(b)> <= find <z> such that perm(b,z) and ord(z)`

N	Assertions	Goals	sort(b)	Reason
1	if not(x=nil)then x=addfirst(head(x),tail(x))			axiom
2	if emptylist(x)then append(x,y)=y			axiom
3	(y1=y2)=(addfirst(x,y1) =addfirst(x,y2))			axiom
4	if x=nil then ord(x)			axiom
5	if x=y then perm(x,y)			axiom
6	perm(append(y,addfirst(u,z)), addfirst(u,x))= perm(append(y,z),x)			axiom
7	if x=addfirst(u,nil)then ord(x)			axiom
8	if not(x=nil)and ord(x)and(u< =head(x))then ord(addfirst(u,x))			axiom
10	if perm(append(w,addfirst(u,x)),y) then append(w,x) $<_{wf}$y			axiom
11	perm(w,y)=perm(y,w)			axiom
12	perm(addfirst(u,x),addfirst(u,z))= perm(x,z)			axiom
13		perm(b,z)and ord(z)	z	new
14		(emptylist(x41) **and** perm(b,x41))	x41	Res. 4,13
15		(emptylist(x42) **and** (b=x42)	x42	Res.5, 14
16		emptylist(b)	b	Ex.15
17		((x43 = addfirst(x44,nil)) **and** perm(b,x43))	x43	Res. 7,13
18		((b = x46) **and** (x46 = addfirst(x47,nil)))	x46	Res. 5,17
19		(b = addfirst(x48,nil))	b	Ex 18
24		(**not**(emptylist(b)) **and** (addfirst(head(b),tail(b)) = addfirst(x54,nil)))	b	Eq. 1,19
29		(emptylist(tail(b)) **and not**(emptylist(b)))	b	L.-Eq. 3,24
31	(**if** (x63 $<_{wf}$ b) **then** perm(x63,sort(x63)) **and if** (x63 $<_{wf}$ b) **then** ord(sort(x63)))			Ind. 13
32	**if** (x64 $<_{wf}$b) **then** perm(x64,sort(x64))			Split 31
33	**if** (x66 $<_{wf}$b) **then** ord(sort(x66))			Split 31
34		(**not**(emptylist(b)) **and not**(perm(tail(b),sort(tail(b)))))		Res. 9,32
35		(**not**(emptylist(b)) **and not**(ord(sort(tail(b)))))		Res.

				9,33
36		(perm(addfirst(head(b),tail(b)),x68) **and** ord(x68) **and not**(emptylist(b)))	x68	Eq. 1,13
40		(emptylist(sort(tail(b))) **and not**(emptylist(b)))	b	Rel.R. 29,34
41		(perm(tail(b),x77) **and** ord(addfirst(head(b),x77)) **and not**(emptylist(b)))	addfirst(head(b),x77)	L.-Eq. 12,36
43		(perm(tail(b),x81) **and not**(emptylist(b)) **and not**(emptylist(x81)) **and** ord(x81) **and** (head(b) < = head(x81)))	addfirst(head(b),x81)	Res. 8,41
45		(**not**(emptylist(sort(tail(b)))) **and** (head(b) < = head(sort(tail(b)))) **and not**(emptylist(b)))	addfirst(head(b),sort(tail(b)))	Res. 44,35
46		(**not**(emptylist(tail(b))) **and** (head(b) < = head(sort(tail(b)))) **and not**(emptylist(b)))	addfirst(head(b),sort(tail(b)))	Rel.R. 34,45
47		(perm(addfirst(head(b),sort(tail(b))),x84) **and** ord(x84) **and not**(emptylist(b)))	x84	Rel.R. 34,36
52		(perm(addfirst(head(b),addfirst(head(sort(tail(b))),tail(sort(tail(b))))),x87) **and** ord(x87) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))))	x87	Eq. 1,47
64	addfirst(a1,b1)=append(addfirst(a1,nil),b1)			new
167		(perm(append(addfirst(head(b),nil),addfirst(head(sort(tail(b))),tail(sort(tail(b))))),x477) **and** ord(x477) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))))	x477	Eq. 52,64
173		(perm(append(addfirst(head(b),nil),tail(sort(tail(b)))),x493) **and** ord(addfirst(head(sort(tail(b))),x493)) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),x493)	L.-Eq. 6,167
254		(ord(addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))) **and** (append(addfirst(head(b),nil),tail(sort(tail(b)))) $<_{wf}$ b))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 32,173
255		(ord(addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))) **and** perm(append(addfirst(head(b),nil),addfirst(x552,tail(sort(tail(b))))),b))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 254,10
269		(perm(append(addfirst(head(b),nil),sort(tail(b))),b) **and** ord(addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Eq. 1,255
274		(perm(addfirst(head(b),sort(tail(b))),b) **and** ord(addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Eq. 64,269

290		(perm(addfirst(head(b),tail(b)),b) **and** ord(addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))))) **and** **not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 34,274
302		(perm(b,b) **and** ord(addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))))) **and** **not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Eq. 1,290
303		(ord(addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))))) **and** **not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 5,302
306		(ord(addfirst(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))))) **and** **not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Eq. 64,303
307		(**not**(ord(addfirst(head(sort(tail(b))),tail(sort(tail(b)))))) **and not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))		Eq. 1,35
309		(**not**(emptylist(tail(b))) **and** **not**(ord(addfirst(head(sort(tail(b))),tail(sort(tail(b)))))) **and not**(emptylist(b)))		Rel.R. 34,307
313	if ord(addfirst(h,a))and elem(e,a)then h< =e			axiom
326	if ord(a)and(forall e(if elem(e,a)then h< =e))then ord(addfirst(h,a))			axiom
327	(**if** ((x606 < = sk_x602(x606,x607)) **and** ord(x607)) **then** ord(addfirst(x606,x607)) **and if** ord(x607) **then** (elem(sk_x602(x606,x607),x607) **or** ord(addfirst(x606,x607))))			Skol 326
328	**if** ((x608 < = sk_x602(x608,x609)) **and** ord(x609)) **then** ord(addfirst(x608,x609))			Split 327
329	**if** ord(x611) **then** (elem(sk_x602(x610,x611),x611) **or** ord(addfirst(x610,x611)))			Split 327
330	elem(d,addfirst(h,t))=((d=h)or elem(d,t))			new
332		(**not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)) **and** **not**(elem(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))))),sort(addfirst(head(b),tail(sort(tail(b)))))))) **and** ord(sort(addfirst(head(b),tail(sort(tail(b))))))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 306,329
333		(**not**(emptylist(tail(b))) **and** **not**(elem(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))))),sort(addfirst(head(b),tail(sort(tail(b)))))))) **and** ord(sort(addfirst(head(b),tail(sort(tail(b)))))))) **and**	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 34,332

		not(emptylist(b)))		
334		(**not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)) **and** (head(sort(tail(b))) < = sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))))) **and** ord(sort(addfirst(head(b),tail(sort(tail(b)))))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 306,328
335		(**not**(emptylist(tail(b))) **and** (head(sort(tail(b))) < = sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))))) **and** ord(sort(addfirst(head(b),tail(sort(tail(b)))))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 334,34
338		(perm(append(x621,addfirst(x622,x623)),b) **and** **not**(ord(sort(append(x621,x623)))))		Res. 10,33
339		(perm(append(x626,addfirst(x627,x628)),b) **and** **not**(perm(append(x626,x628),sort(append(x626,x628)))))		Res. 10,32
356		(perm(append(x709,addfirst(x710,x711)),addfirst(head(b),tail(b))) **and** **not**(perm(append(x709,x711),sort(append(x709,x711)))) **and not**(emptylist(b)))		Eq. 1,339
363		(perm(append(x743,addfirst(x744,x745)),addfirst(head(b),tail(b))) **and** **not**(ord(sort(append(x743,x745)))) **and** **not**(emptylist(b)))		Eq. 1,338
371		(perm(append(x778,addfirst(x779,x780)),addfirst(head(b),sort(tail(b)))) **and** **not**(perm(append(x778,x780),sort(append(x778,x780)))) **and not**(emptylist(b)))		Rel.R. 34,356
380		(perm(append(x807,addfirst(x808,x809)),addfirst(head(b),sort(tail(b)))) **and** **not**(ord(sort(append(x807,x809)))) **and** **not**(emptylist(b)))		Rel.R. 34,363
387		(perm(append(x841,addfirst(x842,x843)),append(addfirst(head(b),nil),sort(tail(b)))) **and** **not**(perm(append(x841,x843),sort(append(x841,x843)))) **and not**(emptylist(b)))		Eq. 64,371
397		(perm(append(x896,addfirst(x897,x898)),append(addfirst(head(b),nil),sort(tail(b)))) **and** **not**(ord(sort(append(x896,x898)))) **and** **not**(emptylist(b)))		Eq. 64,380
409		(perm(append(x957,addfirst(x958,x959)),append(addfirst(head(b),nil),addfirst(head(sort(tail(b))),tail(sort(tail(b)))))) **and** **not**(perm(append(x957,x959),sort(append(x957,x959)))) **and not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))		Eq. 1,387

423		(perm(append(x1025,addfirst(x1026,x1027)),append(addfirst(head(b),nil),addfirst(head(sort(tail(b))),tail(sort(tail(b)))))) **and** **not**(ord(sort(append(x1025,x1027)))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))		Eq. 1,397
431		(**not**(perm(append(x1064,x1066),sort(append(x1064,x1066)))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))) **and** (append(x1064,addfirst(x1065,x1066)) = append(addfirst(head(b),nil),addfirst(head(sort(tail(b))),tail(sort(tail(b)))))))		Res. 5,409
432		(**not**(ord(sort(append(x1067,x1069)))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))) **and** (append(x1067,addfirst(x1068,x1069)) = append(addfirst(head(b),nil),addfirst(head(sort(tail(b))),tail(sort(tail(b)))))))		Res. 5,423
433		(**not**(perm(append(addfirst(head(b),nil),tail(sort(tail(b)))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))		Ex 431
434		(**not**(ord(sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))		Ex 432
436		(**not**(perm(addfirst(head(b),tail(sort(tail(b)))),sort(addfirst(head(b),tail(sort(tail(b))))))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))		Eq. 64,433
438		(**not**(ord(sort(addfirst(head(b),tail(sort(tail(b)))))))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))		Eq. 64,434
439		(**not**(emptylist(tail(b))) **and** (head(sort(tail(b))) < = sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 335,438
440		(**not**(emptylist(tail(b))) **and** **not**(elem(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))))),sort(addfirst(head(b),tail(sort(tail(b))))))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 333,438
442		(**not**(emptylist(tail(b))) **and** **not**(elem(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))))),sort(addfirst(head(b),tail(sort(tail(b))))))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 34,440
444		(**not**(emptylist(tail(b))) **and** (head(sort(tail(b))) < = sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 34,439
445		(**not**(elem(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))))),addfirst(head(b),tail(sort(tail(b)))))) **and** **not**(emptylist(tail(b))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 436,442

446		(**not**(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))) = head(b)) **and** **not**(elem(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))),tail(sort(tail(b))))) **and not**(emptylist(tail(b))) **and** **not**(emptylist(b)) **and** **not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	L.-Eq. 330,445
448		(**not**(elem(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))),addfirst(head(b),tail(sort(tail(b)))))) **and not**(emptylist(tail(b))) **and not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 34,445
449		(**not**(emptylist(tail(b))) **and not**(emptylist(b)) **and** ord(addfirst(head(sort(tail(b))),x1070)) **and** elem(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b))))),x1070))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 313,444
450		(ord(addfirst(head(sort(tail(b))),tail(sort(tail(b))))) **and** **not**(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))) = head(b)) **and** **not**(emptylist(tail(b))) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 449,446
452		(ord(addfirst(head(sort(tail(b))),tail(sort(tail(b))))) **and** **not**(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))) = head(b)) **and** **not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 34,450
458		(ord(sort(tail(b))) **and** **not**(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))) = head(b)) **and** **not**(emptylist(tail(b))) **and not**(emptylist(b)) **and not**(emptylist(sort(tail(b)))))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Eq. 1,452
465		(ord(sort(tail(b))) **and** **not**(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))) = head(b)) **and** **not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Rel.R. 34,458
466		(**not**(sk_x602(head(sort(tail(b))),sort(addfirst(head(b),tail(sort(tail(b)))))) = head(b)) **and** **not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Res. 35,465
467		((head(sort(tail(b))) < = head(b)) **and** **not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b))))))	Eq. 466,444
472		(head(b)<=head(sort(tail(b)))) **and** **not**(emptylist(b)))	**if** (emptylist(tail(b))) **then** b **else** addfirst(head(b),sort(tail(b)))	Res. 29,46
473	(a< =z)=(a< z)or(a=z)			axiom

474		(head(b)<head(sort(tail(b)))) **and** **not**(emptylist(b)))	**if** (emptylist(tail(b))) **then** b **else** addfirst(head(b),sort(tail(b)))	L.-Eq. 473,472
475		(head(b)=head(sort(tail(b)))) **and** **not**(emptylist(b)))	**if** (emptylist(tail(b))) **then** b **else** addfirst(head(b),sort(tail(b)))	L.-Eq. 473,472
476		**not**(emptylist(sort(tail(b)))) **and** **not**(emptylist(b)))	**if** ((head(sort(tail(b))) < = head(b)) **then** addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))) **else** **if** (emptylist(tail(b))) **then** b **else** addfirst(head(b),sort(tail(b)))	Res. 467,474
473		**not**(emptylist(b))	**if** emptylist(sort(tail(b))) **then** b **else if** ((head(sort(tail(b))) < = head(b)) **then** addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))) **else if** (emptylist(tail(b))) **then** b **else** addfirst(head(b),sort(tail(b)))	Res. 40,476
474		**true**	**if** emptylist(b) **then** b **else if** emptylist(sort(tail(b))) **then** b **else if** ((head(sort(tail(b))) < = head(b)) **then** addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))) **else if** (emptylist(tail(b))) **then** b **else** addfirst(head(b),sort(tail(b)))	Res. 16,473

sort(b)= **if** emptylist(b) **then** b **else**

if emptylist(sort(tail(b))) **then** b **else**

```
if ((head(sort(tail(b))) < = head(b)) then
   addfirst(head(sort(tail(b))),sort(append(addfirst(head(b),nil),tail(sort(tail(b)))))) else
     if (emptylist(tail(b))) then  b  else addfirst(head(b),sort(tail(b)))
```

Построенный по выходному терму текст программы на языке Лисп:

```
(DEFUN sort(b)
(COND ((LISTP b)
        (COND ((NULL b) b)
            (T (COND ((NULL (sort (CDR b))) b)     475
                (T (COND ((<= (CAR (sort (CDR b))) (CAR b))
              (CONS (CAR (sort (CDR b))) (sort (CONS (CAR b) (CDR (sort (CDR   b)))))))
                (T (COND ((NULL (CDR b)) b) (T (CONS (CAR b) (sort (CDR b))))))))))))
```

Приложение 7. Алгоритм унификации различий.

В данном приложении сформулирован алгоритм унификации различий, использующийся для получения волновых правил из правил переписывания. Он позволяет расставить волновые фронты в правилах переписывания таким образом, чтобы получившиеся правила сохраняли основу при переписывании и уменьшали меру (согласно определению, приведенному в главе 2). Подробно алгоритм унификации различий описан в работе [Basin and Walsh, 1993].

В отличие от обычного алгоритма унификации, описанного в приложении 2, в результате применения приведенного алгоритма определяется не только возможность отождествления заданных предложений, но и фиксируются различия, которые нужно убрать, чтобы предложения могли быть унифицированы обычным способом.

При проведении унификации различий предложений s и t задача состоит в приведении четверки<{s=t/λ,λ},{},{},{}> к виду <{},Subst,As,At>, где Subst – множество подстановок, As,At – аннотированные предложения s и t, после символа / записываются позиции. Позиция – это способ записи пути внутреннего выражения (терма) в исходном предложении. Перед началом работы алгоритма в качестве внутренних предложений рассматриваются исходные предложения, их позиции пустые - λ). Множество позиций определяется следующим образом:

$Pos(f(s_1,\ldots,s_n)) = \{\lambda\} \cup \{i.p \mid 1<=i<=n \text{ and } p \in Pos(s_i)\}$,

где λ – пустая строка, . – оператор конкатенации строк. Предложение t, находящееся в позиции p, записывается как t/p, где

$t/\lambda = t$

$f(s_1,\ldots s_n)/i.p = s_i/p$

Для позиций введем операцию @ добавления номера в конец позиции:

λ@i = i.λ

(p.q)@i = p.(q@i)

Для проведения унификации применяются следующие правила:

DELETE: <S∪{t=t/p,q},Subst,As,At> ➔ <S,Subst,As,At>

DECOMPOSE: <S∪{f(s1,…sn)=f(t1,…,tn)/p,q},Subst,As,At>➔

<S∪{s1=t1/p@1,q@1,…sn=tn/p@n,q@n}, Subst, As, At>

$ELIMINATE_L$: <S∪{X=t/p,q},Subst,As,At> ➔

<S[X←t], Subst∪{X←t}, As, At>

Правило применимо, если X не входит в t.

$ELIMINATE_R$:

<S∪{s=X/p,q},Subst,As,At> ➔

<S[X←s]}, Subst∪{X←s}, As, At>

Правило применимо, если X не входит в s.

$IMITATE_L$: <S∪{X=f(t1,…,tn)/p,q},Subst,As,At>➔

<S[X←f(X1,…,Xn)]}∪{Xi=ti/p@i,q@i | 1<=i<=n},

Subst∪{X←f(X1,…,Xn)}, As, At>

Правило применимо, если q@i ∈ Pos(t) при i=1,…,n

$IMITATE_R$: <S∪{f(s1,…,sn)=X/p,q},Subst,As,At>➔

<S[X←f(X1,…,Xn)]}∪{Xi=si/p@i,q@i | 1<=i<=n},

Subst∪{X←f(X1,…,Xn)}, As, At>

Правило применимо, если p@i ∈ Pos(s) при i=1,…,n

$HIDE_L$: <S∪{f(s1,…,sn)=t/p,q},Subst,As,At> ➔

<S∪{si=t/p@i,q}, Subst, As∪{p@i}, At>

Правило применимо, если p@i ∈ Pos(s)

$HIDE_R$: <S∪{s=f(t1,…,tn)/p,q},Subst,As,At> ➔

<S∪{s=ti/p,q@i}, Subst, As, At∪{q@i}>

Правило применимо, если q@i ∈ Pos(t)

В качестве примера рассмотрим унификацию различий термов `f(X,a)` и `f(g(a),X)`, где `X` – переменная, `a` - константа. Первое различие встречается, начиная с 3-ей позиции, различающиеся термы – X и g(a). Применим к ним правило HIDE: функция g будет "спрятана" в волновой фронт. Следующее различие (между X и a) ликвидируется применением правила ELIMINATE: выполним подстановку {X←a}. Получим термы `f(a,a)` и `f(g(a),a)`. Больше различий в основах этих термов нет, в итоге выражения аннотированы следующим образом: `f(X,a)` и `f(g(a),X)`.

Следует отметить, что порядок применения правил не фиксирован, и возможны ситуации, в которых применимыми оказываются несколько правил, что дает несколько различных результатов применения алгоритма.