УДК 004.05

Проект системы предикатного программирования

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описываются исходные требования, схема реализации и модель программы транслятора с **языка предикатного программирования** P [1].

Исходные требования к разработке следующей версии системы предикатного программирования

В описании языка P [1] зафиксированы почти все изменения в языке, накопленные за последние несколько лет. Не удалось включить лишь операцию модификации поддеревьев [2]. Адекватное представление таких возможностей в языке возможно лишь в рамках аппарата метапрограммирования. Возможно, в будущем появятся предпосылки для реализации такого аппарата в системе предикатного программирования.

В настоящий момент имеется экспериментальная **система предикатного программирования** с ограниченного подмножества языка P, эксплуатируемая при генерации формул корректности на PVS на задачах студентов НГУ в рамках курса «Формальные методы в программной инженерии». В трансляторе не реализованы параметрические типы и списки, что доставляет серьезные неудобства. Система используется также как полигон для студенческих работ.

Состояние текущей версии системы предикатного программирования. Полностью реализован лексический и синтаксический анализ. Семантический анализ (анализ типов конструкций) реализован для ограниченного подмножества. Написан генератор с внутреннего представления на императивное расширение языка P. Реализован и работает генератор формул корректности на PVS и SMT-решатель CVC4. Реализована проверка части условий семантической корректности программы через SMT-решатель CVC4. Реализация оптимизирующих трансформаций на начальной стадии.

Необходимо определить, где реально может использоваться система предикатного программирования после того, как ее разработка будет завершена и система будет готова для производственной эксплуатации. Разумеется, применение в целях обучения студентов важное, но не главное назначение создаваемой системы. **Основное назначение системы предикатного программирования – это разработка программ повышенной надежности и безопасности с применением дедуктивной верификации, которая более чем в пять раз дешевле по сравнению с общедоступной дедуктивной верификацией для императивного программирования.**

В нашей стране потребность в разработке программ повышенной надежности и безопасности существует уже сейчас. А других инструментов, подобных системе предикатного программирования, для этого в России нет. Такие инструменты есть во Франции, Германии, США. Не в свободном доступе. Например, во Франции есть фирма Prove&Run, работающая на языке Smart. Им можно заказать любую разработку. Описание языка Smart недоступно. По ряду специальных возможностей (например, гиперфункции) совпадение с языком P. Ранее не наблюдалось подходов даже со слабой корреляцией к предикатному программированию.

Разумеется, система предикатного программирования может также использоваться в обычном стиле программирования, без дедуктивной верификации и написания предусловий и постусловий. Такое применение представляется менее приоритетным. Приоритетными являются следующие задачи: реализация трансляции с полного языка P, доработка подсистемы дедуктивной верификации, реализация оптимизирующей трансформации. Для сокращения затрат на дедуктивную верификацию целесообразна реализация собственного **специализированного решателя**, работающего интерактивно в контексте создаваемой программы и набора теорий [3, см. Заключение].

Еще один аспект. Если система предикатного программирования будет использоваться, скажем, в ОАО «Авиадвигатель», то в соответствии со стандартом DO178C необходимо будет проводить **сертификацию** используемых инструментов: транслятора, подсистемы дедуктивной верификации, оптимизирующих трансформаций. В частности, гарантировать, что применение трансформаций не приведет к появлению ошибки в программе.

В отношении полноты реализации языка P необходимо внести уточнения. Есть конструкции и особенности языка со сложной реализацией, полезные, но не обязательные. Без них можно обойтись. Предлагается отложить их реализацию, исключив из следующего релиза. Это неявные и обобщенные типы. Предлагается исключить на время описатель **var.**

Есть особенности, реализация которых обязательна. Это качественная диагностика ошибок программы. Это возможность описывать программы, метки и формулы после их вызовов. При этом предварительное описание для рекурсивных программ становится необязательным. В существующем релизе описание предшествует использованию. Это неудобно, потому что программа разрабатывается сверху вниз. Читать программу также удобнее сверху вниз, начиная с главной программы.

Планируемые релизы системы предикатного программирования

Здесь представлен другой набор релизов по сравнению с предыдущей версией Проекта. Опыт применения системы верификации Why3 [10] в работе [9] определяет новые возможности и принципиально меняет систему приоритетов в планируемой разработке системы предикатного программирования. Дедуктивная верификация в системе Why3 в автоматическом режиме с применением большого набора SMT-решателей существенно быстрее ручного доказательства в системе PVS. Поэтому генерация формул корректности на язык спецификаций Why3 является более приоритетной задачей по сравнению с генерацией формул корректности для системы PVS.

Система Why3 в принципе позволяет выходить на PVS. Однако двойная трансляция формул корректности порождает ряд серьезных проблем и неудобств. Поэтому независимый генератор формул на PVS, по-видимому, остается значимой компонентой в системе предикатного программирования. Эксплуатация Why3 вместе с независимым выходом на PVS порождает необходимость реализации собственного менеджера целей доказательства.

Технология предикатного программирования при отсутствии инструмента, системы предикатного программирования, предполагает построение вручную формул корректности программы применением системы правил доказательства корректности для разных видов операторов. Результатом является теория на языке P. Кодирование этой теории на языке Why3 также реализуется вручную. В работе [9, раздел 8] описаны особенности сертификации результатов применения такой технологии. Несмотря на очевидные недостатки такого стиля, технология будет иметь преимущество по времени перед дедуктивной верификацией императивных программ. Однако появляется большой объем нетривиальной работы для оценщика.

Релиз 1. Трансляция теорий языка P на языки спецификаций Why3 и PVS

Язык теорий – это подмножество языка P, содержащее языки формул, типов и выражений. Лишь часть языка выражений. Не нужны предикатные типы. Необходима часть транслятора для данного подмножества языка, реализующая синтаксический и семантический разбор теорий на входном языке с переводом во внутреннее представление программы. Транслятор теорий переводит теории во внутреннем представлении на язык спецификаций Why3.

В языке Why3 нет подтипов и параметрических типов. Поэтому предстоит систематическое вынесение формул, определяющих подтипы, в предусловия и постусловия. Есть специфика и проблемы при трансляции операций над массивами.

Перевод теории на язык спецификаций PVS реализуется проще. Можно использовать соответствующий модуль из программы текущего релиза системы предикатного программирования.

Релиз 2. Трансляция с подмножества языка P, ориентированная на дедуктивную верификацию

Транслятор с языка теорий в Релизе 1 расширяется до минимального подмножества языка P с набором ограничений, перечисленных ниже. Для программы во внутреннем представлении разрабатывается генератор формул корректности на собственный язык теорий. Через доработку генератора текущей версии системы предикатного программирования. Реализуется трансляция теорий на Why3 и PVS. Система предикатного программирования должна быть реализована в рамках некоторого редактора, предположительно Eclipse, хотя возможны и другие варианты. С качественной диагностикой ошибок.

Ограничения на язык P следующие.

1. Неограниченные типы **nat**, **int**, **real**. Без типов с фиксированной разрядностью.

2. Без классов.

3. Запрещается использование **var**. Типы переменных всегда должны быть явно описаны.

4. Без модулей. Транслируемый текст – последовательность описаний. В следующих релизах будут модули без параметров как независимые области локализации. Что такое модули с параметрами и как можно было бы их использовать, остается непонятным.

5. Без процессов. Реализация процессов и всего с ними связанного (время, аппарат сообщений) откладывается.

6. Откладывается реализации специальных конструкторов для оптимизации работы со списками и строками [1, Разд. 7.3 и 7.4].

Сначала предстоит доработать проект внутреннего представления программы, представленный ниже. Далее следует разработать детальную схему трансляции. Разрешить при этом все принципиальные вопросы.

Генератор формул корректности должен быть существенно доработан и покрывать все виды языковых конструкций реализуемого подмножества языка P. В частности, гиперфункции. Есть недостатки в техническом оформлении файла для PVS.

Релиз 3. Реализация оптимизирующих трансформаций предикатных программ

Концептуальный проект реализации оптимизирующих трансформаций представлен в статье [6]. Это доработанный текст дипломной работы И.В. Каблукова. Трансформация операций со списками и строками – в работе [7]. Программная реализация оптимизирующих трансформаций на начальной стадии.

Принципиальные вопросы реализации оптимизирующих трансформаций решены. Необходим детальный рабочий проект. Необходимо продолжение исследований.

Аналогов в мире нет.

Структура транслятора

Программа транслятора состоит из front-end'а, преобразующего предикатную программу на языке P во внутреннее представление транслятора, двух middle-end'ов и нескольких back-end'ов.

Преобразование предикатной программы во внутреннее представление (front-end) реализуется следующей последовательностью действий:

- Лексический анализ
- Синтаксический анализ с построением дерева внутреннего представления
- Идентификация (сопоставление имен и описаний) и семантический анализ
- Нахождение заместителей переменных предикатного типа. Построение графа вызовов.

Два middle-end'а:

- Дедуктивная верификация с генерацией формул корректности в виде набора теорий
- Применение набора оптимизирующих трансформаций с получением эффективной программы на императивном расширении языка P

В составе back-end'ов:

- Генерация текста на выходном языке (C, C++, …)
- Конвертация формул корректности для системы PVS и набора SMT-решателей.

Модель внутреннего представления программы

Полная предикатная программа во внутреннем представлении (ВП) транслятора определяется в виде дерева[1] структур. Полная предикатная программ состоит из набора предикатных программ (определений предикатов). В данном разделе дается описание различных видов структур, используемых для программы во ВП. Описание структур абстрагировано от конкретной реализации в языке C++.

Далее **set** определяет непустой набор структур одного вида. Обозначение **seq** определяет непустую последовательность структур одного вида. Последовательность упорядочена. Набор не задает порядка. Обозначения **set*** и **seq*** используется в случае, когда допускается пустой набор или пустая последовательность. Обозначение **ptr** Имя структуры используется для указателя на представленную структуру.

1. Программа

Программа = Описания, Область локализации

Программа является структурой самого верхнего уровня – образом во ВП текста некоторого файла с расширением «.psrc». Описания содержат список описаний программы. Область локализации содержит объекты, определяемые Описаниями программы.

Область локализации = Scope(**set*** Объект локализации)

Объект локализации = Имя, Описание

Объект локализации определяет соответствие между Именем объекта, используемым в тексте программы, и объектом, представленным структурой Описание, полученной из соответствующего описания в тексте программы. В Области локализации можно получить доступ к объекту по его Имени.

Область локализации определяет набор имен и обозначаемых ими объектов, представленных Описаниями. В Области локализации каждому Имени соответствует ровно один объект. Исключение: одному Имени может соответствовать несколько полиморфных предикатных программ, различающихся числом и/или типами параметров.

Область локализации прикрепляется к языковым конструкциям некоторых видов. Набор различных Областей локализации определяет дерево областей локализации в соответствии с иерархией языковых конструкций, которым принадлежат Области локализации. Корнем дерева является Область локализации, принадлежащая Программе.

Имеются *автономные* Области локализации, прикрепленные к объекту класса, структуры или объединения. Они не входят в дерево областей локализации. Для автономных областей локализации возможна независимая иерархия, например, между классом и суперклассом.

[1] Точнее, это граф.

2. Описания

Описания = **set*** Описание

Именованный объект = N = Имя

Описание определяет Имя и объект, обозначаемый Именем. Поле Имя является общим у всех Описаний. Далее поле Имя представлено префиксом (N). Фактически структура Именованный объект наследуется во всех структурах, определяющих Описания.

Описание = Описание типа | Описание переменной |
Описание предикатной программы | Описание формулы |
Описание класса | Описание процесса | Описание сообщения |
Описание теории | Описание метки

Виды объектов, определяемых описаниями, соответствуют [1, разд.4], за исключением последнего.

Описание типа =
TypeName(N)(Описания аргументов, Указатель типа, Область локализации)

Указатель типа = **ptr** Типовый терм

Описание типа вводит обозначение – имя N для Типового терма (см. Разд. 6). Описания аргументов определяют параметры описываемого типа. Область локализации не пуста при наличии параметров типа.

Описание переменной = VarDecl(N)(Указатель типа)

Отметим, что описатель **type** (TypeType, Разд.6) также может быть значением Указателя типа.

Описание предикатной программы =
ProgSpec(N)(Заголовок программы, Область локализации) |
Prog(N)(Заголовок программы, Область локализации, Тело программы)

Тело программы может отсутствовать, когда программа представлена только своей спецификацией. Область локализации объединяет объекты, определенные в Заголовке программы и в Теле программы.

Описание формулы = FormulaDecl(N)(Описания аргументов, Тип результата,
Мера, Формула, Область локализации)

Область локализации содержит объекты, определяемые Описаниями аргументов. Задание Меры необходимо лишь в случае рекурсивно определяемой формулы.

Тип результата = Указатель типа

Тип результата **bool** используется для формулы исчисления предикатов. Для других типов Формула определяет функцию указанного типа.

Описание класса =

Class(N)(Описания аргументов, Суперкласс, Описания, Область локализации)

Суперкласс = **ptr** Описание класса

Набор объектов (полей и методов) класса определяется Описаниями. Среди них – конструкторы класса являются предикатными программами, имена которых совпадают с именем класса. Область локализации является автономной. Она объединяет объекты, определенные в Описаниях аргументов и в Описаниях. Суперкласс определяет ссылку на описание суперкласса.

Описание процесса = Process(N)(Заголовок процесса, Состояние процесса,
Тело процесса, Область локализации)

Заголовок процесса = Заголовок программы

Заголовок процесса такой же, как для предикатной программы, за исключением случая бесконечного процесса, когда нет ветвей результатов. Область локализации объединяет объекты, определенные в Заголовке процесса и Состоянии процесса, а также сдержит метки (набор управляющих состояний) и локальные переменные, определенные в Теле процесса.

Состояние процесса = Описания

Главным образом здесь описывается набор переменных, определяющих состояние процесса, однако здесь могут быть также описания сообщений и в принципе любые другие описания.

Описание сообщения = Message(N)(Типы параметров сообщения)

Типы параметров сообщения = **seq*** Указатель типа

Сообщение идентифицируется именем и типами параметров сообщения.

Описание теории =

Theory(N)(Описания аргументов, Утверждения теории, Область локализации)

Утверждения теории = **seq** Утверждение теории

Утверждение теории = Assertion(Имя утверждения, Формула, Вид утверждения)

Вид утверждения = (теорема | аксиома)

Область локализации объединяет объекты, определенные в Описаниях аргументов, и набор утверждений, идентифицируемых их именами.

Описание метки = Label(N)(Адрес)

Адрес определяет позицию метки, т.е. оператора, который метится этой меткой, в операторной структуре программы.

3. Заголовок программы

Заголовок программы =

ProgHead(Описание внешних аргументов, Описание аргументов,

Предусловие, Постусловие, Мера, Результаты программы)

Заголовок программы – единая структура для предикатной программы, в частности, для программы-гиперфункции, а также для программы-процесса.

Предусловие = Формула

Постусловие = Формула

Мера = Выражение

Задание Меры необходимо лишь для рекурсивной программы. Предусловие, Постусловие и Мера используются при дедуктивной верификации и могут отсутствовать, когда дедуктивная верификация не проводится.

Полное предусловие, конструируемое при построении формул корректности, состоит из Предусловия и набора утверждений принадлежности значений аргументов их типам, объявленных в Описании аргументов. Если типом аргумента является подтип, то соответствующее ограничение на значение аргумента должно быть включено в полное предусловие. Аналогичным образом, *полное постусловие* содержит утверждения принадлежности значений результатов их типам. Если тип результата является подтипом, то соответствующее ограничение должно быть включено в полное постусловие.

Описание аргументов = **seq*** Описание переменной

Описание внешних аргументов = Описание аргументов

Определяются имена и типы параметров-аргументов предикатной программы.

Результаты программы = Описание результатов | Ветви результатов

Первая альтернатива соответствует обычной программе – не гиперфункции. Вторая альтернатива – для гиперфункции при наличии не менее двух ветвей.

Ветви результатов = **seq** Ветвь результатов

Ветвь результатов = Branch(Описание результатов, Метка ветви,
Предусловие ветви, Постусловие ветви)

Описание результатов = **seq*** Описание переменной

Метка ветви = Имя

Исполнение гиперфункции должно завершаться оператором перехода на Метку ветви для какой-либо ветви гиперфункции.

Для предикатной программы, не гиперфункции, недопустим пустой список результатов. По меньшей мере, одна переменная в качестве результата должна присутствовать.

Предусловие ветви определяет ограничение на значения переменных для данной ветви в рамках полного предусловия. Т.е., если P1 – предусловие ветви, а P – полное предусловие, то соответствующее ограничение на аргументы ветви есть P & P1.

Предусловие ветви отсутствует для одной из ветвей. В этом случае оно определяется как дополнение по отношению к предусловиям остальных ветвей в рамках полного предусловия. Пусть P – полное предусловие, а P1 и P2 – предусловия для первой и второй ветви. Тогда предусловие для последней третьей ветви определяется как P & ¬P1 & ¬P2.

Если ветвь содержит непустой список результатов, то эта ветвь может содержать Постусловие ветви, по которой строится *полное постусловие ветви* с учетом типов результатов данной ветви.

4. Операторы

Тело программы = **set** Сегмент

Локальные переменные и метки, определенные в Теле программы принадлежат Области локализации из Заголовка программы. Блоки внутри Тела программы не имеют собственной области локализации. Поэтому переменные и метки, определенные в любом месте Тела программы принадлежат единой Области локализации. За исключением конструкций, содержащих собственную Область локализации.

Множество Сегментов программы образует *дерево сегментов*. Исполнение программы начинается с *корневого сегмента*, находящегося в начале программы. Остальные сегменты являются образами конструкций вида:

M **:** Оператор

Здесь M – метка Оператора.

Сегмент = Segment(Входная метка, Оператор, Выходы)

Выходы = **seq*** Метка

Входная метка = Метка

Метка = Имя

Входная метка определяет начало Сегмента. Она является локальной меткой Тела программы и находится перед Оператором данного сегмента. Два сегмента связаны дугой в дереве сегментов, если в первом сегменте есть оператор перехода на Входную метку второго сегмента. Корневой сегмент Входной метки не имеет.

Выходы содержат входные метки других сегментов, а также метки ветвей гиперфункции, на которые имеются переходы в текущем Сегменте.

Некорневые сегменты, в частности, появляются при отображении ОПЕРАТОРА-ОБРАБОТКИ-ВЕТВЕЙ. Например, фрагмент программы:

A(i**:** u**:** y #1**:** #2) **case** 1**:** B(y, i**:** u) **case** 2**:** C(i**:** u); D(u**:** z)

преобразуется во ВП в следующие четыре сегмента:

A(i**:** u #3**:** y #1**:** #2); 1**:** B(y, i**:** u); #3; 2**:** C(i**:** u); #3; 3**:** D(u**:** z)

Оператор = Присваивание | Мультиприсваивание | Оператор перехода |
Вызов программы-функции | Вызов гиперфункции |
Условный оператор | Суперпозиция операторов |
Параллельная композиция |
Описание локальной переменной | Описание локальной программы

Описание локальной переменной = Описание переменной

Описание переменной для локальной переменной можно вынести в начало программы. За одним исключением, когда тип переменной – параметрический, а значения параметров зависят от другой локальной переменной. В этом случае Описание переменной должно остаться на своем месте.

Присваивание = Assign(Переменная, Выражение)

Мультиприсваивание = MultiAssign(**seq** Переменная, **seq** Выражение)

В Мультиприсваивании типы Переменных должны быть совместимы с типами соответствующих Выражений.

Оператор перехода = Goto(Метка)

Вызов программы-функции =

Call(Вызываемая программа, **seq*** Аргумент вызова, **seq** Результат вызова)

Вызываемая программа = Выражение //в частности, имя предикатной программы

Аргумент вызова = Выражение

Результат вызова = Переменная

Вызов гиперфункции =

HyperCall(Вызываемая программа, **seq*** Аргумент вызова, **seq** Ветвь вызова)

Ветвь вызова = **seq*** Результат вызова, Метка продолжения

Метка продолжения = Метка

Отметим, что Метка продолжения имеется на каждой ветви вызова. При отсутствии метки для одной из ветвей в исходной программе, эта метка вставляется. Например, фрагмент:

H(i**:** x**:** y #M); A1;

будет преобразован во ВП следующим образом:

H(i**:** x #1**:** y #M); 1**:** A1;

Условный оператор = If(**seq** Ветвь условного, Иначе оператор)

Ветвь условного = Условие, Оператор

Условие = Выражение

Иначе оператор = Оператор

ОПЕРАТОР-ВЫБОРА преобразуется в Условный оператор. Например, фрагмент

switch (a[i]){ **case** 1**:** A1 **case** 2**:** A2 **default :** A3}

преобразуется во ВП в эквивалентный условный оператор:

nat t = a[i]; **if** (t=1) A1 **elsif** (t=2) A2 **else** A3

Фрагмент обработки объектов алгебраических типов, в частности, списков:

switch (s[i]){ **case** nil**:** A1 **case** cons(h, t)**:** A2}

преобразуется в следующий условный оператор:

list(**nat**) b = s[i]; **if** (nil?(b)) A1 **else** { {h=b.car || t=b.cdr}; A2}

Отметим, что проверка истинности распознавателя cons?(b) для последнего конструктора опущена, поскольку nil?(b) $\vee$ cons?(b) тождественно истинна. Вместо nil?(b) обычно используется эквивалентное отношение b = nil.

Суперпозиция операторов = Superposition(**seq** Оператор)

Параллельная композиция = Parallel(**seq** Оператор)

Описание локальной программы = Описание предикатной программы

В теле Описания локальной программы доступны все объекты из области локализации программы, в теле которой встречается Описание локальной программы.

5. Выражения

Типизированный объект = T = Указатель типа

Типизированный объект – объект, для которого тип является одним из основных атрибутов. Далее тип объекта представляется префиксом (T). Фактически структура Типизированный объект наследуется во всех структурах, определяющих Выражения и Переменные.

Переменная = Указатель переменной | Элемент массива | Поле структуры |

Переменная класса | Вырезка массива

Указатель переменной = **ptr** Описание переменной

Вхождение переменной в исходной программе определяется Именем переменной. Во ВП вместо Имени используется Указатель переменной, отсылающий к Описанию переменной.

Элемент массива = ArrElem(T)(Выражение, **seq** Индекс)

Индекс = Выражение

Поле структуры = StructField(T)(Выражение, Номер поля)

Переменная класса = ClassVar(T)(Выражение, Указатель переменной)

Вырезка массива = ArrSlice(T)(Выражение, Суженный набор типов индексов)

Суженный набор типов индексов = **seq** Индекс или диапазон

Индекс или диапазон = Индекс | Диапазон

Диапазон = Range(Выражение, Выражение)

Элемент массива, поле структуры, вырезки массива могут находиться в правой части оператора присваивания или в качестве результата вызова программы. Такое вхождение переменной интерпретируется как операция модификации **with** для полной структуры, компонентом которой является переменная.

Выражение = Value(T)(Переменная) | Вызов функции | Константа |
Указатель типа | Указатель программы | Метод |
Агрегат | Определение массива | Определение массива по частям |
Модификация агрегатом | Модификация частями |
Конструктор | Распознаватель |
Поле объединения | Элемент списка | Вырезка списка | Сообщение |
Условное выражение | Унарная операция | Бинарная операция |
Приведение

Вычисление Выражения поставляет значение, используемое при исполнении конструкции, содержащей Выражение. Вхождение Переменной в позиции Выражения определяет взятие значения Переменной. Однако во ВП операция Value опускается, хотя подразумевается.

Вызов функции = Superposition(Вызов программы-функции, Value(T)(Переменная))

Во ВП вызов функции вида f(x) преобразуется в вызов предиката f(x**:** t), где t – новая локальная переменная. Значением Вызова функции является оператор суперпозиции {f(x**:** t); Value(t)} в стиле Алгол-68. Задача корректного вынесения вызова предиката из выражения, содержащего Вызов функции, в общем случае нетривиальна. Отметим, что такое вынесение необходимо для дедуктивной верификации.

Константа = Const(T)(Значение константы)

Только для примитивных типов.

Указатель программы = **ptr** Описание предикатной программы

Отметим, что Указатель программы также определяет литерально задаваемую программу конструкцией ГЕНЕРАТОР-ПРЕДИКАТА.

Метод = Method(T)(Выражение, Указатель программы)

Указатель программы ссылается на программу, запускаемую в качестве метода класса. Объект класса представлен Выражение. Тип T есть предикатный тип, соответствующий программе.

Агрегат = Aggregate(T)(**seq*** Элемент агрегата)

Элемент агрегата = Выражение | Номер элемента, Выражение

Номер элемента = Выражение

Номер элемента – номер поля в структуре (константа) или индекс элемента массива. Агрегат должен задавать структуру или массив полностью. В структуре должны

присутствовать все поля. Для массива должны быть представлены все его элементы. Неполное задание элементов допустимо, когда Агрегат используется в операции модификации. Тип агрегата T определяется по виду агрегата и набору Элементов агрегата, если этот тип не был указан явно в начале агрегата.

Определение массива = ArrayDef(T)(Заголовок определения массива, Выражение)

Заголовок определения массива = **seq** Индекс-переменная, Область локализации

Индекс-переменная = Указатель переменной

Индекс-переменная соответствует индексу элемента массива по очередному измерению массива. Длина списка переменных совпадает с числом измерений. Набор Индекс-переменных определяет новую Область локализации, действующую в пределах данного Определения массива. Тип T есть тип определяемого массива.

Выражение определяет элемент массива для набора индексов, указанных в заголовке.

Определение массива по частям =

ArrayDefCase(T)(Заголовок определения массива, Определение частей массива)

Определение частей массива = **seq** Определение части массива, Часть по умолчанию

Часть по умолчанию = Выражение

Определение части массива = **seq**(Суженный набор типов индексов), Выражение

В каждой части Определения массива по частям свое Выражение для элемента массива в рамках заданного множества наборов индексов. Отметим, что данная конструкция содержит Область локализации.

Отметим сложную трехъярусную структуру множеств наборов индексов. Это множества наборов, которые не должны пересекаться между собой. При этом каждое Определение части массива может определять несколько множеств. Каждое множество определяется структурой Суженный набор типов индексов. Набор в множестве соответствует набору индексов элемента массива. По каждому измерению это либо одиночный индекс, либо диапазон.

Часть по умолчанию определяет значение элемента массива для набора индексов, не покрываемых множествами наборов индексов во всех Определениях части массива. Часть по умолчанию может отсутствовать. В этом случае множества наборов индексов по всем Определениям части массива должно покрывать любые допустимые наборы индексов.

Модификация агрегатом = WithAgr(T)(Выражение, Агрегат)

Модификация частями = WithCase(T)(Выражение, Определение массива по частям)

Выражение определяет структуру или массив в операции WithAgr. В Агрегате указывается часть элементов, которые модифицируются в исходной структуре или массиве. Агрегат не может быть полным.

В Определении массива по частям должна отсутствовать Часть по умолчанию. Покрытие наборов индексов должно быть неполным.

Конструктор = Constructor(T)(Номер конструктора, Поля конструктора)

Поля конструктора = **seq*** Выражение

Конструктор определяет объект типа объединения T заданием имени конструктора (здесь идентифицируется Номером конструктора) и значениями всех Полей конструктора.

Распознаватель = Recognizer(T)(Номер конструктора, Выражение)

Реализуется проверка, что объект, представленный Выражением типа **union**, соответствует конструктору, номер которого указан параметром. Тип T есть **bool**.

Поле объединения = UnionField(T)(Выражение, Номер поля)

Номер поля объединения определяется по всему типу **union**.

Элемент списка = ListElem(T)(Выражение, Индекс)

Индекс = Выражение

Вырезка списка = ListSlice(T)(Выражение, Диапазон)

Проверка корректности указанных операций со списком, а также корректность обращения к полю объединения – все это функции семантического анализа предикатных программ.

Сообщение = MessageGet(T)(Указатель сообщения, Параметры сообщения)

Указатель сообщения = **ptr** Описание сообщения

Параметры сообщения = **seq*** Указатель переменной

Тип T есть **bool**. Истинное значение при получении сообщения в момент исполнения данной конструкции.

Условное выражение = IfExpr(T)(Условие, Выражение, Выражение)

Условие = Выражение

Типы альтернатив в Условном выражении должны быть согласованы.

Унарная операция = Унарная(Т)(Выражение)

Бинарная операция = Бинарная(Т)(Выражение, Выражение)

Унарная = PlusUnaryInt | PlusUnaryReal | MinusUnaryInt | MinusUnaryReal |
Negate | ComplementSet | ComplementInt

Здесь представлены операции для всех унарных операций: **+, -, !, ~**. Есть сомнения в полезности унарного плюса. Остается вопрос относительно операции **~** дополнения для целых. Для отрицательных. Типы **int** и **nat** не ограничены. А ограниченные типы мы пока не рассматриваем.

Бинарная = AddNat | AddInt | AddReal | AddSet | AddLists | AddArrays |
SubtractNat | SubtractInt | SubtractReal |
MultiplyNat | MultiplyInt | MultiplyReal |
Divide | Reminder | Power |
Less | Greater |LessOrEquals | GreaterOrEquals | Equals | NotEquals |
And | Or | Xor | Implies | Iff | ShiftLeft | ShiftRight | In

Соответствующий набор бинарных операций: **+, -, *, /, %, ^, <, >, <=, >=, =, !=, &, or, xor, =>, <=>, <<, >>, in**. Есть пробелы в исходном языке. Непонятно, для каких типов и комбинаций типов допустимы операции Equals и NotEquals. Для операций сравнения Less и т.д. допустимо ли сочетание типов операндов **nat** и **real** ?

Приведение = Cast(T)(Выражение)

Значение Выражения преобразуется к типу Т. Необходимо будет точно определить, для каких комбинаций типов проводится Приведение.

6. Типы

Указатель типа = **ptr** Типовый терм

Типовый терм = Примитивный тип | Вызов типа | Неизвестный тип как параметр |
Тип массива | Подтип | Тип структуры | Тип объединения |
Тип перечисления | Тип множества | Предикатный тип | Указатель класса

Указатель класса = **ptr** Описание класса

Примитивный тип = Nat(N) | Int(N) | Real(N) | Bool(N) | Char(N)

Здесь N – имя примитивного типа. Определены отношения и Nat **<:** Int и Int **<:** Real. Напомним, что в текущем релизе нет разрядности типов. Числовые типы потенциально бесконечные.

Вызов типа = TypeCall(S)(Указатель описания типа, **seq*** Выражение)

Указатель описания типа = **ptr** Описание типа

Вызов типа есть вхождение имени типа с возможными параметрами.

Неизвестный тип как параметр = TypeType(N)

Это тип, являющийся параметром программы или типа и определяемый описанием: **type** ИМЯ-ТИПА. N = **type**.

Тип массива = Array(S)(Тип элемента, **seq** Тип по измерению)

Тип элемента = Указатель типа

Тип по измерению = Указатель типа

Указатель типа = Номер типового терма

Подтип = Subtype(S)(Указатель переменной, Формула, Область локализации)

Область локализации состоит из единственной переменной, определяемой первым параметром. Для подтипа **subtype**(тип x**:** формула) Область локализации содержит описание переменной x.

Тип диапазона преобразуется в Подтип.

Тип структуры = Struct(S)(**seq*** Описание поля, Область локализации)

Описание поля = Номер поля, Описание переменной

Каждое Описание переменной представляет описание поля структуры: имя переменной – это имя поля, тип переменной – тип поля. Имена полей ассоциируются со структурой. Они образуют автономную Область локализации. Число полей структуры не меньше двух. Меньшее число полей возможно при использовании внутри конструктора объединения.

Тип объединения = Union(S)(**seq** Описание конструктора, Область локализации)

Описание конструктора = Constructor(N)(Номер конструктора, Тип структуры)

Здесь N – имя конструктора. Поля конструктора определяются Типом структуры. Имя конструктора и имена полей конструктора определяют автономную Область локализации, прикрепленную к Типу объединения.

Тип списка list является Типом объединения. Описание типа списка представлено в библиотечной части языка P, которая транслируется перед началом трансляции программы. Тип **string**, определяемый как **type string** = list(**char**), также транслируется перед началом трансляции программы из библиотечной части.

Тип перечисления = Enum(S)(Число элементов, **seq** Имя элемента перечисления)

Число элементов = Натуральная константа

Имя элемента перечисления = Имя

Тип перечисления является Типом объединения с набором конструкторов без полей. Однако здесь используется другое представление. Элемент перечисления идентифицируется в программе его номером.

Тип множества = Set(S)(Тип базового множества)

Тип базового множества = Указатель типа

Предикатный тип = PredicateType(S)(Заголовок программы)

Имена аргументов и результатов в структуре Заголовок программы могут быть не заданы при отсутствии предусловий и постусловий.

7. Формулы

Формула = Выражение | Вызов формулы | Формула с кванторами |
Условная формула

Формула имеет тип. Тип **bool** используется для формулы исчисления предикатов. Для других типов Формула определяет функцию указанного типа.

В качестве Формулы допустимы следующие виды Выражений: константы, переменные, Условное выражение, унарные и бинарные операции. В качестве переменных везде в формулах допустимы лишь простые переменные, поля структур, поля объединений и элементы массивов. Все остальные конструкции Выражений, в частности вызов функции, не могут использоваться в составе Формул. Вместо вызова функции допустимо использовать Вызов формулы.

Вызов формулы = FormulaCall(T)(Указатель описания формулы, **seq*** Выражение)

Указатель описания формулы = **ptr** Описание формулы

Формула с кванторами =

QuantorFormula(**seq** Квантор, Формула, Область локализации)

Квантор = Указатель переменной, Вид квантора

Вид квантора = для всех | существует

Набор кванторных переменных составляет Область локализации, прикрепленную к Формуле с кванторами.

Условная формула = IfExpr(T)(Условие, Формула, Формула)

Аналогично Условному выражению. Альтернативами в Условной формуле здесь являются Формулы.

Некоторые алгоритмы трансляции

Рассматривается однопроходная схема трансляции при реализации front-end'а. Многопроходные схемы возможны. Но они сложнее.

1. Идентификация

Идентификация – одна из стадий трансляции, при которой все вхождения имен в программе заменяются указателями на описания соответствующих объектов.

Описания имен должны предшествовать их использующим вхождениям в программе. За исключением имен предикатных программы. Описание предикатной программы может находиться позже ее вызова. При описании рекурсивных типов возможно придется использовать предописания вида "**type** T(...);".

Соответствие между именами и описаниями поддерживается структурой:

Текущий контекст = **set*** Объект контекста

Объект контекста = Имя, Описание, Предыдущий объект контекста

Предыдущий объект контекста = **ptr** Объект контекста

В Текущем контексте имеется набор Имен. Каждое Имя ссылается на соответствующее Описание. Для некоторого Имени в объемлющей Области локализации может существовать другой Объект контекста, который фиксируется как Предыдущий объект контекста.

В языке P допускаются полиморфные программы – возможность определить несколько программ с одинаковым именем и разными параметрами. В рамках каждой Области локализации строятся цепочки полиморфных программ, описанных в этой Области локализации. И только первая программа из каждой цепочки попадает в Текущий контекст.

Если для некоторого вхождения имени программы или вызова программы не обнаружено соответствующего описания, такое вхождение запоминается в отдельном списке с фиксацией текущей Области локализации. Идентификация для таких вхождений реализуется по завершению трансляции всего текста программы. Описание имени программ ищется по иерархии Областей локализации от текущей и далее по цепочке объемлющих.

2. Отношения на типах

Семантический анализ программы определяет типы конструкций программы и реализует проверку множества ограничений на типы смежных конструкций. Система ограничений на типы формулируется с использованием трех видов отношений на типах.

Тип T является *совместимым* с типом U (*вложенным* в тип U), если любое значение типа T являются также значением типа U. Будем использовать обозначение T <: U. Если рассматривать типы как множества значений, то T <: U эквивалентно $T \subseteq U$.

Тип T *тождественен* типу U, если T <: U и U <: T. Используется обозначение T ≐ U.

Тип S является *мажорантой* типов T и U, S = maj(T, U), если T <: S и U <: S. Далее будем рассматривать наименьшую мажоранту. Типы T и U *согласованы*, если они имеют общую мажоранту.

В процессе семантического анализа программы необходимо уметь проверять отношения совместимости, тождества и согласованности для произвольных типовых термов T и U, а также вычислять их мажоранту. Предлагаемый ниже алгоритм базируется на системе правил, полученных переработкой правил в магистерской работе А. Зубарева [8, разд.2].

Чтобы не писать одинаковые правила для <: и ≐, далее в правилах будем использовать знак ♦, означающий <: или ≐.

Следующие правила применяются неявно при работе алгоритма.

maj(T, U) |- maj(U, T);
T ≐ U ≅ T <: U & U <: T;
T <: U, U ♦ W ⊢ T <: W;
T ♦ U ⊢ maj(T, U) = U;
maj(T, T) = T;

Далее правила для примитивных типов.

nat <: **int**; **int** <: **real**; **nat** <: **real**;
nat ♦ **nat**; **int** ♦ **int**; **real** ♦ **real**;
maj(**nat**, **nat**) = **nat**; maj(**nat**, **int**) = **int**; maj(**nat**, **real**) = **real**;
maj(**int**, **nat**) = **int**; maj(**int**, **int**) = **int**; maj(**int**, **real**) = **real**;
maj(**real**, **nat**) = **real**; maj(**real**, **int**) = **real**; maj(**real**, **real**) = **real**;

Следующая группа правил для подтипов.

T <: U ⊢ **subtype**(T x: P(x)) <: U;
subtype(T x: P(x)) <: T;
T ≐ **subtype**(T x: **true**);
T <: U, x = y & P(x) ⇒ Q(y) ⊢ **subtype**(T x: P(x)) <: **subtype**(U y: Q(y));
T ≐ U, x = y => (P(x) ⇔ Q(y)) ⊢ **subtype**(T x: P(x)) ≐ **subtype**(U y: Q(y));
maj(**subtype**(T x: P(x)), **subtype**(U y: Q(y))) = **subtype**(maj(T, U) x: P(x) **or** x=y & Q(y));
maj(**subtype**(T x: P(x)), U) = maj(T, U);

В данных правилах предполагается, что подтипы приведены к *нормальному виду*, т.е. базовый тип подтипа не является подтипом. В частности, подтип **subtype**(**nat** x**:** P(x)) должен быть нормализован к виду: **subtype**(**int** x**:** x ≥ 0 & P(x)).

Правило вида A, B ├ C подразумевает, что C сводится к A и B. Т.е. отношение для подтипов сводится к отношению для базовых типов и проверке истинности формулы B.

Используется формула x = y & P(x) ⇒ Q(y) вместо P(x) ⇒ Q(x), поскольку формулы Q(x) нет во ВП программы. Ее, конечно, можно построить по Q(y), но это не лучшее решение. Истинность формулы x = y & P(x) ⇒ Q(y) реализуется сравнением деревьев для P(x) и Q(y) с учетом равенства x = y. В первом релизе предполагается использование лишь простейших эвристик. Во втором релизе планируется использование SMT-решателя в ситуации, когда собственный решатель не дает ответа.

Пусть K⟨T⟩ обозначает типовый терм K, в котором имеется подтерм T. Тогда K⟨U⟩ – типовый терм K, в котором в позиции терма T вместо T находится U.

Допустим, имеется типовый терм K⟨T⟩, в котором позиция T является одной из следующих: тип поля структуры или конструктора объединения, тип базового множества типа **set**, тип элементов в типе массива или тип аргумента предикатного типа. Тогда имеют место следующие правила:

T ♦ U ├ K⟨T⟩ ♦ K⟨U⟩;
maj(K⟨T⟩, K⟨U⟩) = K⟨maj(T, U)⟩;

Допустим, имеется типовый терм K⟨T⟩, в котором позиция T является одной из следующих: Тип по измерению в описании типа массива или тип результата Предикатного типа. Тогда имеют место следующие правила:

T ≐ U ├ K⟨T⟩ ≐ K⟨U⟩;
T ≐ U ├ maj(K⟨T⟩, K⟨U⟩) = K⟨U⟩;

Последние две группы правил предопределяют следующий алгоритм вычисления отношений на типовых термах T и U. Если T подтип, а U не подтип, то проверка отношения T <: U переносится на базовый тип для T и терм U. Случай, когда T или U – вызов типа, будет рассмотрен ниже. В остальных случаях необходимо проверять деревья термов T и U на совпадение везде кроме позиций вложенных подтипов. Например, для случая, когда T и U типы структур, исходное отношение последовательно проверяется для всех полей. При несовпадении имен полей или их числа вычисляемое отношение на T и U становится ложным.

Пусть K⟨**pre** P⟩ – предикатный тип с позицией предусловия **pre** P. Тогда имеют место правила:

P ⇒ R ├ K⟨**pre** P⟩ <: K⟨**pre** R⟩;
P ⇒ R ├ maj(K⟨**pre** P⟩, K⟨**pre** R⟩) = K⟨**pre** R⟩;

Пусть K⟨**pre** P⟩ – предикатный тип с позицией предусловия **pre** P. Тогда имеют место правила:

Аналогично, K⟨**post** Q⟩ – предикатный тип с позицией постусловия **post** Q.

Q ⇒ S ├ K⟨**post** Q⟩ <: K⟨**post** S⟩;
Q ⇒ S ├ maj(K⟨**post** Q⟩, K⟨**post** S⟩) = K⟨**post** S⟩;

Допустим, имеется описание типа **type** A(x) = T, где x – список параметров, возможно пустой. Z – список фактических параметров в вызове типа A(Z).

x = Z, T ♦ U ├ A(Z) ♦ U;
x = Z, U ♦ T ├ U ♦ A(Z);
x = Z ├ maj(A(Z), U) = maj(T, U);
Y = Z ├ A(Y) ♦ A(Z);

Правила определяют переход от вызова типа A(Z) к определению типа A в виде типового терма T. Таким образом, надо будет сравнить термы T и U при наличии системы равенств x = Z. Причем в списке Z не только простые переменные – параметры могут быть в виде поддеревьев.

Если тип A – рекурсивный, то лишь последнее правило может быть применимо.

Список литературы

1. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Новосибирск, 2018. 42с. http://persons.iis.nsk.su/files/persons/pages/plang14.pdf
2. Шелехов В.И. Предикатная программа вставки в АВЛ-дерево // Системная информатика, № 9. — Новосибирск, 2017. — С. 23-42. http://persons.iis.nsk.su/files/persons/pages/avl_insert.pdf
3. Шелехов В.И. Синтез операторов предикатной программы // Труды конф. «Языки программирования и компиляторы ’2017», Ростов-на-Дону — 2017 — С.258-262. http://persons.iis.nsk.su/files/persons/pages/sintr.pdf
4. Тумуров Э.Г., Шелехов В.И. Технология автоматного программирования на примере программы управления лифтом. — ИСИ СО РАН, Новосибирск, 2016. – 18с. http://persons.iis.nsk.su/files/persons/pages/lift1.pdf
5. Шелехов В.И. Разработка автоматных программ на базе определения требований // Системная информатика, №4, 2014. — ИСИ СО РАН, Новосибирск. — С. 1-29. http://persons.iis.nsk.su/files/persons/pages/req_tech.pdf
6. Каблуков И.В., Шелехов В.И. Реализация оптимизирующих трансформаций в системе предикатного программирования // Системная информатика, № 11. — Новосибирск, 2017. — С. 21-48. http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf
7. Булгаков К.В., Каблуков И.В., Тумуров Э.Г., Шелехов В.И. Оптимизирующие трансформации списков и деревьев в системе предикатного программирования // Системная информатика, № 9. — Новосибирск, 2017. — С. 63-92. http://www.system-informatics.ru/files/article/105.pdf
8. Зубарев А.Ю. Анализ типов в трансляторе с языка предикатного программирования // Выпускная квалификационная работа магистра. НГУ, Новосибирск, 2018. — 54с. http://persons.iis.nsk.su/files/persons/pages/diplom_14.pdf

9. Шелехов В.И. Дедуктивная верификация и оптимизация предикатной программы конкатенации строк // Системная информатика, № 12. — Новосибирск, 2018. — С. 61-84. http://persons.iis.nsk.su/files/persons/pages/strcat.pdf
10. Why 3. Where Programs Meet Provers. URL: http://why3.lri.fr