«Модельный суперкомпилятор MSCP-A»

Неформальное описание структур и алгоритмов

Институт программных систем РАН

Лаборатория «Автоматизации программирования»

Декабрь, 2016 г.

Переславль-Залесский 2016

АННОТАЦИЯ

Данный документ содержит неформальное описание некоторых структур и алгоритмов модельного суперкомпилятора MSCP-A для алгоритмически полного подмножества базисного Рефала, синтаксис и семантика которого существенно используют ассоциативный конструктор построения данных. В MSCP-A для описания параметризованных состояний преобразуемой программы используются уравнения в словах. Модельный суперкомпилятор MSCP-A является единственным суперкомпилятором, использующим для анализа программ алгоритмы решения уравнений в свободном моноиде.

СОДЕРЖАНИЕ

Введение

Данный документ содержит неформальное описание некоторых структур и алгоритмов модельного суперкомпилятора MSCP-A для алгоритмически полного подмножества базисного Рефала (далее – язык L), синтаксис и семантика которого существенно используют ассоциативный конструктор построения данных. В MSCP-A для описания параметризованных состояний преобразуемой программы используются уравнения в словах. Модельный суперкомпилятор MSCP-A является единственным суперкомпилятором, использующим для анализа программ алгоритмы решения уравнений в свободном моноиде.

В языке описаний параметризованных состояний программы (и свойств этих состояний), преобразуемой суперкомпилятором MSCP-A, введено понятие системы уравнений на моноиде множества данных Рефала, сужающей множество возможных значений параметров, от которых зависит данное множество состояний. Таким образом, свойства значений параметров, соответствующие (не)принадлежности к множествам решений систем уравнений в словах, в том случае когда не удается явно решить эти системы уравнений, описываются в терминах самих этих уравнений: язык уравнений является частью языка описания параметризованных состояний L-машины. Эти системы уравнений отражают семантику и синтаксис L-программ и, следовательно, алгоритмы преобразования и анализа L-программ одновременно можно использовать для анализа свойств параметризованных состояний дерева развертки. Данное свойство языка описания параметризованных состояний L-машины является одним из ключевых элементов разработки суперкомпилятора MSCP-A и является оригинальной разработкой Проекта.

Нижеследующие классические алгоритмы суперкомпиляции расширены для возможности обработки впервые введенных вышеуказанных языковых конструкций: алгоритм прогонки; алгоритм проверки вложения множества состояний, описанного одним параметризованным состоянием, в множество состояний, описанного другим параметризованным состоянием; алгоритм обобщения двух данных параметризованных состояний. Эти языковые конструкции обрабатываются суперкомпилятором MSCP-A с помощью методов решения систем уравнений в словах: алгоритмов разрешения квадратичных уравнений в словах, леммы Леви и решения диофантовых уравнений на длины значений переменных.

Разработанный и реализованный алгоритм обобщения параметризованных состояний может

порождать уравнения в словах, использующиеся при анализе обобщенного состояния и хранящие дополнительную информацию о данных этого состояния.

Разработаны и реализованы алгоритмы обобщения и вложения параметризованных стеков функций, использующие свойства множеств решений уравнений в свободном моноиде, основанные на представлении элементов соответствующих стеков функциональными схемами в базисе имен функций суперкомпилируемой программы.

Алгоритмы решения уравнений в словах адаптированы к множеству данных, содержащему дополнительный свободный конструктор построения данных. В частности, эти алгоритмы стали исключать уравнения, для существования решения которых необходимо, чтобы переменная уравнения выражалась через повторное вхождение себя на другом скобочном уровне указанного конструктора. Реализован алгоритм конечного порождения уравнений по транзитивности равенства. Алгоритмы решения уравнений в словах также применены для решения задачи порождения компактной остаточной программы – результата суперкомпиляции исходной программы.

Пусть даны два параметризованных состояния (конфигураций) C_1, C_2 некоторой суперкомпилируемой программы. Алгоритмы обобщения конфигураций C_1, C_2 и вложения множества конкретных состояний, описанное посредством C_1, в соответствующее множество, описанное посредством C_1, согласованы со структурой данных «слоистый стек», представляющей стек вызовов функций и являющейся линейным случаем классического понятия функциональной схемы. Использования структуры данных «слоистый стек» для описания параметризованного стека вызова функций преобразуемой программы иногда позволяет улучшить временную сложность (в худшем случае) этой программы.

Оригинальное отношение Турчина, определенное на элементах последовательности параметризованных стеков вызовов функций вдоль потенциального пути (следа) вычисления программы, обобщено на понятие элементов последовательности параметризованного «слоистого стека» вызовов функций вдоль следа вычислений. В результате отношение Турчина стало возможно использовать для построения более точных обобщений конфигураций «слоистого стека».

Получил развитие алгоритм обобщения структуры уравнений в свободном моноиде, сужающих множества допустимых значений параметров, описывающих параметризованные состояния суперкомпилируемой программы. В частности, удалось частично решить проблему потери информации при обобщении символа и пустой строки до параметра, включающего оба эти понятия, что позволило расширить класс решаемых суперкомпилятором задач верификации программных моделей, спецификации которых используют ассоциативный конструктор

приписывания данных.

Предложен алгоритм разрешения неоднозначности построения декомпозиции параметризованной конфигурации стека вызовов функций, основанный на применении отношения Турчина. При необходимости декомпозиции параметризованного стека для свертки пути развития вычислений в цикл этот алгоритм однозначно строит искомую декомпозицию с *минимально возможной* потерей информации о свойствах исходного стека, основываясь на глобальных свойствах следа вычислений к которому принадлежит декомпозируемый стек. Предложенный алгоритм может быть применен в суперкомпиляторах функциональных языков программирования, использующих для построения данных как ассоциативный, так и не ассоциативный конструктор приписывания.

О терминалогии:

В данном документе используется терминология функционального языка программирования Рефал [27]. Также используется терминология из работ [23], [25] В.Ф. Турчина.

1. Структуры данных модельного суперкомпилятора MSCP-A

1.1. Об объектном (входном) языке и внутреннем языке преобразований модельного суперкомпилятора MSCP-A

Входным языком суперкомпилятора MSCP-A является подмножество базисного Рефала [27] (далее язык L). В данном разделе дается неформальное введение в это подмножество функционального языка программирования Рефал и в рабочий язык преобразований суперкомпилятора MSCP-A.

Множество данных языка Рефал является свободным моноидом относительно операции приписывания, которая обозначается пробелом, с дополнительным унарным свободным конструктором, который обозначается без имени – только своими скобками. Единицей моноида данных является пустая последовательность, которая обозначается *ничем*.

```
 data ::= data1 data2 || (data) || symbol || empty
Empty ::=
symbol::= CHAR || IDENTIFIER || NUMBER
```

Например, `('a') ('b' A)` – рефал-данное построенное посредством двух конструкторов приписывания и двух унарных конструкторов – скобок.

Аналогично определяется множество рефал-термов, которые в Рефале называются выражениями.

```
  Expr ::= Expr1 Expr2 || (Expr) || Call || Var || symbol || empty
  Call ::= <Function-Name Expr>
   Var ::= s.Var-Name || t.Var-Name || e.Var-Name
 empty ::=
symbol ::= CHAR || IDENTIFIER || NUMBER
```

В этом функциональном программирования языке есть три типа переменных: e-переменные (принимающие значение выражение), t-переменные (принимающие значение символ либо выражение в скобках) и s-переменные (принимающие значение символ). Здесь под символами понимаются буквы латинского алфавита, идентификаторы и натуральные числа. Рефал-переменная типа p с именем `Name` записывается как `p.Name`. Конкатенация двух выражений

[Expr$_1$] и [Expr$_2$] в Рефале записывается как [Expr$_1$] [Expr$_2$]. Вызов функции F с аргументом [Expr] записывается как <F [Expr]>. Т.е. все функции в Рефале являются унарными – имеют ровно один аргумент, которым может быть и пустое выражение.

Семантика Рефала основана на переписывании термов (выражений) и отождествлении по образцу.

Определение функции в языке L есть упорядоченный набор правил переписывания вида

[Образец] = [Правая часть];

Образец может содержать скобки, переменные и символы. Он не может содержать кратных вхождений e-переменных либо вхождений нескольких e-переменных на одном скобочном уровне. Правая часть есть выражение, множество переменных которого является подмножеством множества переменных образца.

При исполнении вызова функции F с аргументом [data] все ее правила просматриваются согласно их упорядочеванию – сверху вниз. Первое из правил, образец которого успешно сопоставится с данным [data], будет применено для замены вызова <F [data]> правой частью этого правила.

Далее везде в данном Приложении рассматриваемое подмножество базисного Рефала (язык L) будем также называть Рефалом.

Программа на языке Рефал, поданная на вход суперкомпилятору MSCP-A, переводится в его рабочий внутренний язык преобразований. Рефал-выражения кодируются посредством нижеследующей функции **μ**, где мета-понятия выделены жирным шрифтом.

```
1  μ(Expr1 Expr2) = μ(Expr1) μ(Expr2)
2  μ((Expr)) = ('*' μ(Expr))
3  μ( <F Expr> ) = (call (F t.timestamp)(args (arg Expr)))
4  μ( Type.var-Name ) = (var Type var-Name)
5  μ( Symbol ) = Symbol
6  μ(  ) =
```

Внутренний язык суперкомпилятора MSCP-A содержит новые сущности – параметры, которые типизированы аналогично переменным. В отличие от переменных, которые *могут* принимать значения, согласованные с их типом, параметры являются семантическими объектами. Их значения *уже заданы*, согласно с их типами, но *неизвестны*. Другими словами, параметры являются мета-сущностями относительно данных. Можно считать, что они связаны квантором всеобщности.

Переменные записываются в кодировке MSCP-A (т.е. в его внутреннем языке) как (var

`[Type] [Name])`, параметры – как `(par [Type] [Name])`.

Кроме того, во внутреннем языке имеются словесные переменные, обозначаемые как `(weval [Type] [Name])` (от word evaluation) – эти переменные используются при работе с уравнениями в словах, порождаемыми суперкомпиляцией. Тип словесной переменной может быть `e` или `t`.

Чтобы отличить конструктор структурных скобок самого Рефала, используемый суперккомпилятором MSCP-A, от конструктора структурных скобок преобразуемой программы последние помечаются дополнительным символом `'*'` сразу после открывающей скобки. Из соображений краткости мы не всегда пишем этот символ при записи внутренних структур суперкомпилятора, если это не приводит к недоразумениям.

Вызов функции <`F` **`e.Arg`**> в MSCP-A кодируется выражением

`(call (F` **`s.timestamp`**`)(args (arg` **`e.Arg`**`)))`

Число **`s.timestamp`** – уникальная метка, которая присваивается вызову сразу же, как только он появляется в конфигурации (не обязательно в активной части стека вызовов функций!) – параметризованном состоянии преобразуемой программы. Она необходима, чтобы хранить дополнительную информацию о глобальных свойствах вызовов, нужную для построения более точных обобщений.

Факт равенства параметра или переменной некоторому выражению `[Expr]` записывается в MSCP-A как `(assign [Значение параметра или переменной]([Expr]))`.

Уравнения в моноиде параметризованных рефал-выражений, возникающие в процессе преобразования входной программы, в MSCP-A записываются как

`(AreEqual ([ExprData1]) ([ExprData2])),`

где блоки `[ExprDataX]` могут содержать не только сами выражения, равенство которых утверждается, но и вспомогательную информацию о них.

1.2. Структура дерева вычислений программы, преобразуемой суперкомпилятором MSCP-A

Узел дерева вычислений программы в MSCP-A хранит информацию о своём статусе, о сужениях параметров, породивших его, уравнениях, ограничивающих множество значений его параметризованного состояния – конфигурации, своей конфигурации и структуре своего стека, и описывается следующей структурой, в определении которой используется рефал-терминология:

[Node]::=	(Node t.Status t.Name ([Restrictions])([LetExpressions]) ([Equations])([Expression])([Stack])) {(Children [Node]*) || empty })
t.Status::=	Driven || Undriven || Ready || Finished || (Looped t.NodeName [Assignments]) || ReadyForGenCheck || (Generalized t.NodeName (Looped [Assignments]) ([Equations])([Expression]) ([Stack])([Assignments])([Node]+)) || (Generalized t. NodeName (UpperGeneralization) ([Expression])([Stack]) ([Assignments])([Node]+))
[Assignments]::=	empty || (assign (par t.type t.name)([Expression])) [Assigments]
[Restrictions]::=	empty || (assign (par t.type t.name)([Expression])) [Assigments]
[LetExpressions]::=	empty || Let [Assignments] In [Expression]
[Stack]::=	--- см. описание в Разделе 1.3 данного Приложения.

Статусы узлов можно пояснить следующим образом:

- `Driven` – узел прогнан, но у него еще возможно есть потомки, не прошедшие прогонку;
- `Undriven` – узел только что порожден;
- `Ready` – прогонка показала, что требуется вытолкнуть очередной вызов;
- `Finished` – узел и все его потомки прогнаны;
- `(Looped t.NodeName [Assignments])` – конфигурация, хранящаяся в данном узле, вкладывается в некоторый узел с именем `t.NodeName` посредством подстановки `[Assignments]`;
- `ReadyForGenCheck` – конфигурация из данного узла прошла проверку на точные вложения (см. Раздел 5.1 данного Приложения) в конфигурации узлов-предков с отрицательным результатом и готова к проверке на возможные обобщения;
- `(Generalized t.NodeName (Looped [Assignments]) ([Equations])([Expression])([Stack])([Assignments])([Node]+)))`

– конфигурация из данного узла была обобщена с конфигурацией узла-предка `t.NodeName`, при этом полученное обобщение – выражение `[Expression]` – совпадает с конфигурацией узла `t.NodeName` по модулю переименования переменных. При таком обобщении конфигурация в узле-потомке разрезается на выражение `[Expression]`, содержащее параметры после обобщения, и выражения в блоке `[Assignments]`, содержащие назначения на эти параметры для узла-потомка. Подробнее об обобщении см. Раздел 5.2 данного Приложения.

- `(Generalized t.NodeName (UpperGeneralization) ([Expression])([Stack])([Assignments])([Node]+))` – конфигурация из данного узла была обобщена с конфигурацией узла-предка `t.NodeName` так, что выражение, являющееся обобщением обоих конфигураций, не совпадает с конфигурацией узла-предка. Конфигурация в узле-предке при этом разрезается на выражение `[Expression]`, содержащее параметры после обобщения, и выражения в блоке `[Assignments]`, содержащие назначения на эти параметры для узла-предка.
- Структура `(Children [Node]+)` описывает узлы-потомки данного узла. Если данный узел имеет статус `Looped`, либо имеет статус `Finished` и не имеет вызовов в стеке, структура `Children` у такого узла отсутствует.

1.3. Структура параметризованного стека вызов функций программы, преобразуемой суперкомпилятором MSCP-A

В MSCP-A используется модель стека, представляющая собой линейный вариант функциональной схемы. Участниками Проекта эта модель названа слоистым стеком. Слоистый стек имеет вид

```
e.Stack ::= (e.StackLevel) e.Stack || (e.StackLevel)
e.StackLevel ::= e.VarDef || e.VarDef e.StackLevel
                
e.VarDef ::= (assign (var l t.varname)(e.StackVal))
             || (assign (var l t.varname)
                        ((call (t.fname t.timestamp)
                               (args (arg e.Expr))
                        ))
                )
```

```
e.StackVal ::= empty || ('*' e.StackVal) e.StackVal
               || Symbol e.StackVal || Parameter e.StackVal
               || (var l t.varname) e.StackVal
```

Объект `(var l t.varname)` – это переменная специального типа «стековая переменная слоя» (l – от layer). Она принимает значение, равное либо вызову функции, либо выражению, не содержащему вызовов функций.

Выражение `e.Expr` внутри вызова функции может содержать вызовы функций, параметры и стековые переменные, определенные в слое стека непосредственно над данным слоем. Поэтому вызов, стоящий в верхнем (первом) слое стека не должен содержать стековых переменных вообще.

Декомпозиция выражения в слоистый стек происходит следующим образом. Сначала из выражения выделяются все вызовы, не содержащиеся внутри других вызовов. Они формируют верхний уровень стека, в котором эти вызовы заменены на стековые переменные (l-переменные). Затем происходит попытка вычислить первый вызов на верхнем уровне. Если она успешна, то результат вычисления вместе со всеми остальными переменными того же уровня подставляется в следующий уровень стека, и опять происходит декомпозиция. Иначе порождается следующий уровень стека таким же образом, как происходила декомпозиция исходного стека вызовов функций.

Таким образом, поведение слоистого стека аналогично поведению стека при нормальном порядке вычислений, за исключением того, что слоистый стек хранит больше информации.

Может возникнуть вопрос, зачем необходимо подставлять все переменные в стек и переформировывать его полностью, если изменения произошли лишь в одной переменной. Нижеследующий пример показывает, почему в некоторых случаях это целесообразно.

Пример 1.3.1

Пусть дана конфигурация

```
<F <G e.x> <H e.x>> <F e.x e.x <H e.x>>
```

И пусть

`<G e.x> = e.x e.x,`

а вычисление вызова функции `F` невозможно без знания первого и последнего символов ее аргумента.

После начальной декомпозиции стек будет выглядеть так:

верхний слой:

```
((assign (var l 3)((call (g 3)  (args (arg (par e x))))))
(assign (var l 4)((call (h 4) (args (arg (par e x)))))))
```

средний слой (декомпозиции подвергался лишь первый его член!):

```
((assign (var l 1)((call (f 1) (args (arg (var l 3) (var l 4)))))))
 (assign (var l 2)((call (f 2) (args (arg (par e x) (par e x)
                                          (call h (args (arg (par e x)
                                  ))
                    ))
 )))))
```

нижний слой:

```
((assign (var l 0)((var l 1)(var l 2))))
```

Затем вычислится вызов `(call (g 3) (args (arg (par e x))))`. После подстановки соответствующих значений в выражение для `(var l 1)` выяснится, что значения переменных `(var l 1)` и `(var l 2)` полностью совпадают, и стек примет вид:

верхний слой:

```
((assign (var l 2)((call (h 4) (args (arg (par e x)))))))
```

средний слой:

```
((assign (var l 1) ((call (f 1) (args (arg (par e x)(par e x)
                                          (var l 2)))))))
```

нижний слой:

```
((assign (var l 0)((var l 1)(var l 1))))
```

Аналогичное совпадение аргументов вызовов может произойти сколь угодно далеко в стеке. Поскольку главная задача этой модификации стека – выявление равных дочерних вызовов узлов конфигурации, целесообразно проводить проверку на их наличие у всех предков узла, подвергшегося изменению.

Из-за того, что слоистый стек подвергается попыткам перестройки после развития любого из его элементов, временные метки назначаются вызовам функций не в процессе появления их в стеке, а сразу как только вызовы появятся в конфигурации.

Слоистый стек, описанный выше, позволяет разворачивать сразу несколько равных вызовов, стоящих на одном уровне в дереве вызовов конфигурации. Таким образом, дерево вызовов превращается в граф, возможно имеющий кратные ребра, а в остальном соответствующий дереву. Такое преобразование позволяет эффективно использовать ранее разработанные в рамках Проекта алгоритмы вложения и обобщения древесных конфигураций, при этом избегая чрезмерного их усложнения, как, например, в случае, когда дерево заменяется орграфом (см. [21]). В то же время, хотя слоистый стек формально получается более глубокой декомпозицией, чем стек при обычных нормальных (ленивых) вычислениях, первые элементы его слоев, содержащие вызовы, описывают такой же стек, какой получился бы при декомпозиции в семантике нормальных (ленивых) вычислений. Все остальные элементы слоев, кроме первых, выделяются лишь для того, чтобы иметь возможность быстро обнаружить равенство двух вызовов, дочерних к одному и тому же вызову функции, и в дальнейшем разворачивать их совместно.

2. Основной шаг суперкомпилятора MSCP-A

Основная функция суперкомпиляции `UnfoldMain` получает на вход дерево вычислений, состоящее только корня – входной точки программы (возможно, параметризованной). Эта вершина получает статус `Unready`. После чего осуществляется следующий цикл.

1. Корневая вершина не имеет потомков с непрогнанными конфигурациями – порождение графа развертки программы завершено, можно порождать остаточную программу.
2. Нашлось противоречие в корневой конфигурации – программа не может быть вычислена, все дерево удаляется.
3. Нашлось противоречие в некорневой конфигурации – удаляем узел, в котором было противоречие, и переходим к его родителю.
4. Некорневой узел получил статус `Finished` – переходим к его родителю.
5. Если узел прогнан и имеет единственный дочерний узел с вытолкнутым вызовом, заменяем данные родительского узла данными дочернего, при этом родительский узел получает статус `Ready`.
6. Если узел прогнан или оказалось, что его конфигурация является повторной – вставляем результат прогонки в дерево вычислений и ищем ближайший узел без прогонки (см.

Раздел 5.1 данного Приложения, точное вложение).

7. Если узел повторяет конфигурацию узла-предка – помечаем его статусом `Looped` и ищем ближайший непрогнанный узел. Если узел подвергся проверке на эквивалентности, и она не выявила повторений – он помечается как готовый для прогонки.
8. После прогонки проверяем, является ли узел кандидатом на обобщение (см. Раздел 5 данного Приложения).
9. Если выяснилось, что конфигурация данного узла обобщается, причём обобщение происходит только в этом узле (а узел-предок остается неизменным), удаляем всех потомков данного узла, расщепляем конфигурацию данного узла и помечаем узел как прогнанный.
10. Если конфигурация данного узла обобщается, причем обобщение затрагивает и узел-предок, то конфигурация узла-предка расщепляется и все прежние его потомки удаляются.
11. Если узел имеет статус `Ready`, начинаем его прогонку (см. Раздел 1.2 данного Приложения).
12. Если узел не содержит вызовов функций, помечаем его как завершенный.

Таким образом, после завершения работы функции `UnfoldMain` она выдает дерево вычислений, все вершины которого либо являются объектными выражениями, либо повторяют конфигурацию какого-либо своего предка – либо пустое дерево, если оказалось, что программа никогда не может быть вычислена.

3. Алгоритмы прогонки

3.1. Конфигурация прогонки

Общий вид конфигурации прогонки `e.Config` в MSCP-A следующий.

```
e.Config ::= ((e.equations)(e.restrictions)
              (e.assignments)(e.Clashes)(s.Log))
             )
s.Log ::= 'T' | 'F' | 'N'
```

Структура сужений имеет следующий синтаксис.

```
e.restrictions ::= (assign (par s.param_type t.param_name)
                           (e.parametrized_expression))*
                   )
```

Его описание совпадает с данным В.Ф. Турчиным в работе [25].

Структура назначений переменным образца имеет следующий синтаксис.

```
e.assignments ::= (assign (var s.variable_type t.variable_name)
                          (e.parametrized_expression)
                  )*
```

Его описание совпадает с данным В.Ф. Турчиным в работе [25].

Структура сопоставлений имеет следующий синтаксис.

```
e.Clashes ::= ((e.parametrized_expression) to (e.pattern))*
              (Delayed ((s.Log)(e.parametrized_expression)
                        to (e.pattern))*
              )
```

Добавлена структура отложенных сопоставлений `Delayed`, поскольку теперь столкновение с невозможностью разрешить сопоставление немедленно не обязательно означает невозможность разрешить сопоставление в течение текущего шага прогонки. Сопоставления в блоке `Delayed` отличаются от активных тем, что содержат флаг внесенных изменений. После завершения первого этапа прогонки отложенные сопоставления в блоке `Delayed` переводятся в уравнения и обрабатываются вместе с элементами блока `e.equations`.

Структура уравнений имеет следующий синтаксис.

```
e.equations ::= (AreEqual ((s.Log) e.FreeExpression) ((s.Log)  e.FreeExpression))*
                (Desired
                        (AreEqual ((s.Log) (call (t.function_name t.timestamp)
                                  (args (arg e.FreeExpression)))) ((s.Log) e.FreeExpression))*
                )
```

Эта структура является новым элементом MSCP-A по сравнению с другими реализациями суперкомпиляторов. В него записываются все равенства, возникающие в сопоставлении, которые невозможно разрешить на данном этапе прогонки. В структуре `e.equations` есть специальная под-структура `Desired`, содержащая условия на значения вызовов функций. Каждая часть уравнения структуры `e.equations` содержит флаг изменений `s.Log`. Если этот флаг принимает значение `'T'`, значит, в уравнение вносились изменения, после которых ещё не было проверки корректности уравнения. Если флаг есть `'F'`, значит, таких изменений не было. Если этот флаг принимает значение `'N'`, значит, данное уравнение принадлежит к классу чисто негативных – оно не может порождать новых подстановок в параметры и используется только для поиска возможных противоречий в описании конфигурации.

3.2. Общий алгоритм прогонки

Основные функции прогонки `ClashLeft` (сопоставление слева) и `ClashRight` (сопоставление справа) на входе получают конфигурацию, а на выходе – гроздь – дерево ветвлений параметров, листья которых содержат новые конфигурации. Прогонка начинается с вызова сопоставления слева.

Если удалось сопоставить образец в определении функции с выражением внутри вызова функции, вызывается алгоритм обработки уравнений.

Если к концу сопоставления остались отложенные сопоставления, производится проверка, нашлись ли среди них те, которые существенно уточнялись. Эти сопоставления переносятся в обычную структуру сопоставлений, и для них снова вызывается `ClashLeft`.

Если обе части сопоставления пусты, удаляем его из структуры `e.Clashes`.

Если в образце осталась только e-переменная, сохраняем ее значение в назначениях и удаляем текущее сопоставление. Свойства образца гарантируют, что назначение e-переменной может порождаться лишь одно.

Если в сопоставляемом выражении (далее – левой части) остался единственный e-параметр, сопоставляемый пустому выражению, добавляем в структуру сужений пустое сужение на e-параметр, после чего подставляем его в структуру назначений, уравнений и сопоставлений.

Сопоставление символов успешно только при их равенстве.

Сопоставление символа `s.Sym` в образце s- или t-параметру

```
(par s.par_type t.par_name)
```

влечёт порождение сужения `(assign (par s.par_type t.par_name)(s.Sym))` и затем подстановку его в структуры назначений, уравнений и сопоставлений.

Сопоставление символа `s.Sym` в образце e-параметру `(par e t.par_name)` влечёт расщепление конфигурации на две: с подстановкой в структуру сужений сужения

```
(assign (par e t.par_name)(s.Sym (par e t.new_par_name)))
```

и с подстановкой в структуру сужений сужения `(assign (par e t.par_name)())`. Новое сужение подставляется также в структуры `(e.assignments)`, `(e.equations)` и `(e.clashes)`. Похожим образом обрабатывается сопоставление s- и t-переменных образца e-параметру, а также выражения в скобках. При сопоставлении `(var s t.var_name)` или `(var t t.var_name)` параметру `(par e t.par_name)` в конфигурации, соответствующей непустому сужению, в сужении

```
(assign (par e t.par_name)
        ((par s.par_type t.new_par_name) (par e t.new_par_name2)))
```

`s.par_type` совпадает с типом переменной образца, после чего в назначения подставляется

```
(assign (var s.var_type t.var_name)(par s.var_type  t.new_par_name))
```

и производится проверка назначения на единственность.

Сопоставление s- или t-переменной образца символу либо s-параметру порождает назначение этой s- или t-переменной. Производится проверка назначения на единственность, после чего (если не обнаружено противоречия) назначение сохраняется в `e.assignments`. Если оказалось, что назначение этой переменной уже было на другой s- или t-параметр, этот параметр сужается до нового параметра или символа, и производится подстановка сужения в блоки `e.assignments`, `e.clashes`, `e.equations`.

Сопоставление t-переменной образца выражению в скобках `('*' e.Val)` отличается от предыдущего случая тем, что если эта t-переменная ранее была сопоставлена также с выражением в скобках `('*' e.OldVal)`, в структуру равенств будет добавлено уравнение

```
(AreEqual (('F')e.NewVal)(('F')e.Val))
```

, если `e.Val` либо `e.NewVal` представляют собой единичный вызов, то равенство добавляется в блок `Desired`.

Сопоставление s-переменной t-параметру порождает сужение t-параметра до s-параметра с новым именем. Далее сужение обрабатывается как обычно.

Сопоставление образца с единственным вызовом `((call e.call)) to (e.Pattern)`

порождает уравнение в структуре `Desired` вида

```
(('F') (call e.call)) (('F') e.Pattern)).
```

Сопоставление терма, символьной или термовой переменной `t.TermOrSym` в образце с вызовом порождает две конфигурации: в одну в структуру `Desired` помещается уравнение

```
(AreEqual (('F')(call e.call))(('F'))),
```

в другую – уравнение

```
(AreEqual (('F')(call e.call))(('F')t.TermOrSym (weval e t.New_Index))),
```

где `(weval e …)` – это псевдопараметры типа строка, использующиеся только в структуре решения уравнений. Во второй конфигурации замены `(call e.call)` на

```
t.TermOrSym (weval e t.New_Index)
```

в текущем сопоставлении не происходит; вместо этого сопоставление переключается на разбор справа (если оно осуществлялось слева) или перемещается в отложенные (если уже осуществлялось справа).

Сопоставление выражения в скобках `('*'e.InBracks)` t-параметру влечёт порождение сужения на t-параметр, в котором все вхождения переменных образца в `('*'e.InBracks)` заменены вхождениями новых параметров соответствующих типов, а сами переменные образца получают в качестве назначений эти параметры. Далее сужения и назначения обрабатываются как обычно.

Сопоставление выражения в скобках `('*' e.InBracks)` e-параметру влечёт расщепление на две конфигурации: с пустым сужением на e-параметр и с сужением на e-параметр вида

```
(assign (par e t.OldParameter)
        (('*' (par e t.NewParameter1)) (par e t.NewParameter2)),
```

после чего сужение обрабатывается как обычно.

Сопоставление выражения в скобках выражению в скобках порождает два сопоставления в рамках этой же конфигурации: отдельно сопоставляются внутрискобочные и внескобочные выражения.

Если e-переменная в образце (не единственная) сопоставляется с произвольным выражением – при разборе справа сопоставление перемещается в структуру `Delayed`, а разбор слева переключается на разбор справа.

Все остальные сопоставления невозможны и влекут удаление конфигурации из грозди

прогонки.

После разбора всех сопоставлений конфигурации происходит разбор структуры уравнений. Для этого все уравнения, а также сопоставления, оказавшиеся отложенными, порождают чистые уравнения в словах (см. Раздел 4.1 данного Приложения). Алгоритмы решения уравнений в словах пытаются построить на базе этих и переданных из конфигурации выше уравнений либо противоречие, либо новые сужения на параметры.

После завершения обработки уравнений производится обратная замена строковых переменных на данные программы, подстановка всех полученных алгоритмами обработки уравнений сужений, после чего все элементы грозди прогонки просматриваются, и:

1) если элемент грозди прогонки содержит противоречие, он удаляется;
2) если он содержит непустую структуру `Desired` (напомним, что в структуре `Desired` хранятся условия на вызовы функций, содержащихся в пассивной части стека исходной конфигурации), все вызовы, стоящие в блоке `Desired` и приравнивающиеся к чему-либо, отличному от свободной переменной (отсутствующей в блоке уравнений), добавляются в мультимножество кандидатов на обработку;
3) если блок `Desired` элемента грозди прогонки пуст, оставляем гроздь прогонки в ее исходном виде.

Если мультимножество вызовов-кандидатов на обработку непусто, из него выбирается элемент, имеющий наибольшую кратность. Этот элемент становится новым активным вызовом.

Если мультимножество вызовов-кандидатов на обработку пусто, порождается столько дочерних конфигураций к исходной, сколько осталось элементов грозди прогонки с различными блоками сужений – причем каждая из них помечена соответствующей структурой сужений. В каждой из них производится подстановка соответствующих назначений в правую часть определения функции.

4. Модуль алгоритмов решения уравнений в свободном моноиде рефал-данных

4.1. Общий алгоритм решения уравнений

Все алгоритмы решения уравнений оперируют уравнениями, свободными от структур

данных прогонки – для этого структуры данных прогонки заменяются в них на строковые переменные. *Чистое уравнение в словах* может содержать термы, а также строковые переменные двух типов: `(weval e …)` и `(weval t …)`. В первые переводятся: e-параметры, e-переменные, а также ими являются псевдопараметры структуры `Desired`. Во вторые: t-, s-переменные, t-, s-параметры.

Общий вид чистого уравнения в словах имеет вид структуры

```
(AreEqual ((s.Log)(e.Complexity)(e.Expression))((s.Log)
          (e.Complexity)(e.Expression)))
```

Здесь `s.Log` – флаги изменений; `e.Complexity` – мультимножество строковых переменных соответствующей части уравнения, `e.Expression` – выражение, стоящее в левой или правой части уравнения. Мультимножество `e.Complexity` есть набор вида

`(t.multiplicity (weval e t.name))`* `(t.multiplicity_const const)`,

где `t.multiplicity` – кратность переменной `(weval e t.name)` в части уравнения, `t.multiplicity_const` – суммарное количество всех термов и вхождений переменных типа `(weval t t.name)` в эту часть уравнения. Отсчёт кратностей e-переменных ведется с 1, отсчёт констант – с 0 (т.е. структура `(t.multiplicity_const const)` обязательно присутствует в `e.Complexity`).

В 2016 году в рамках работ по Проекту реализован алгоритм анализа уравнений этой структуры с учетом транзитивности равенства. Обработка уравнений начинается с того, что уравнения в блоке упорядочиваются следующим образом.

1. Все повторные (с учётом коммутативности равенства) уравнения удаляются. Все оставшиеся подвергаются процедуре расщепления на подуравнения (см. Раздел 4.1.2 данного Приложения) и из результатов также удаляются повторные. Затем все полученные уравнения проверяются на противоречие по алгоритму решения квадратичного уравнения и помещаются в множество `Eqs_All`.
2. Происходит порождение новых уравнений по транзитивности равенства. А именно, если множество `Eqs_All` может быть удачно сопоставлено с образцом

```
e.00 (AreEqual e.11 (t.Log1 t.Complex1 (e.CommonPart)) e.12)
e.01 (AreEqual e.21 (t.Log2 t.Complex2
                          (e.X e.CommonPart e.Y)) e.22) e.02,
```

а выражение `e.11 e.12` есть `(t.Log1x t.Complex1x (e.EqualPart))`, то порождается уравнение

```
(AreEqual e.21 (t.Log2 t.Complex2x
                       (e.X e.EqualPart e.Y) e.22).
```

Причем, если `t.Log1` есть `'N'`, то все флаги изменений в порожденном уравнении также заменяются на `'N'`.

Здесь `t.Compex2x` – мультимножество, полученное вычитанием мультимножества `t.Complex1` из `t.Complex2` и объединением результата с мультимножеством `t.Complex1x`. Новое уравнение подвергается расщеплению и проверке на противоречивость алгоритмом решения квадратичных уравнений, и все полученные им уравнения, если они отсутствуют в блоке `Eqs_All`, помещаются в его конец.

Шаг 2 можно пояснить на следующем примере.

Пример 4.1.1

Пусть в прогонке сопоставляются образец

```
(var t 1) (var t 1) (var t 2) (var t 2)
```

и выражение

```
('*' (par e x) 'A' (par e y)) ('*B' (par e y) (par e z))
  ('*' (par e x)(par e x) 'A' (par e y))
  ('*' (par e y) 'B' (par e z)(par e z)).
```

Тогда изначально структура уравнений будет содержать лишь два уравнения. Из соображений краткости мы опускаем запись мультимножеств параметров.

```
(AreEqual ((weval e 1) 'A' (weval e 2))('B' (weval e 2)(weval e 3)))

(AreEqual ((weval e 3) (weval e 1) 'A' (weval e 2))
          ((weval e 2) 'A' (weval e 3)(weval e 3)))
```

На шаге 2 будет замечено, что первая часть первого уравнения есть подслово первой части второго уравнения. Будет порождено уравнение

```
(AreEqual ((weval e 3) 'B' (weval e 2)(weval e 3))
          ((weval e 2) 'A' (weval e 3)(weval e 3))),
```

которое упростится расщеплением до уравнения

```
(AreEqual ((weval e 3)'B' (weval e 2))((weval e 2) 'A' (weval e 3))).
```

Это уравнение принадлежит к классу квадратичных уравнений, причем оно не имеет корней. Таким образом, исходная конфигурация противоречива. Эту информацию невозможно было извлечь из двух исходных уравнений ни расщеплением, ни алгоритмом решения квадратичных уравнений.

4.1.1. Доказательство завершаемости работы алгоритма порождения уравнений по транзитивности

Конечность времени работы алгоритма порождения уравнений по транзитивности обосновывается леммой Хигмана [8]. Предположим, что алгоритм не завершает свою работу. Тогда существует такое уравнение, для правой части которого найдутся эквивалентные подстановки какой угодно длины. Обозначим эти эквивалентные правые части (упорядоченные по возрастанию длины) R_1, …, R_n, …. Поскольку количество weval-e-переменных в блоке уравнений конечно и может только уменьшаться, по лемме Хигмана среди этих равных правых частей найдутся две такие R_i и R_j, что R_i есть подпоследовательность R_j. Применим алгоритм расщепления к уравнению R_i=R_j. Если он не приводит к противоречию, значит, выражение R_j содержит, помимо термов R_i, только некоторое количество weval-e-переменных. Алгоритм расщепления присвоит всем этим переменным значение пустое слово и подставит его во все остальные уравнения.

Если оказалось, что после этого больше не существует неограничено растущих наборов попарно равных правых частей, алгоритм породит лишь конечное множество добавочных уравнений. Иначе возьмем любой такой набор R'_1, …, R'_n, … и выберем в нем такие две правые части, одна из которых является подпоследовательностью другой. Расщепление равенства, их содержащего, приведет либо к противоречию, либо к замене хотя бы одной weval-e-переменной пустым словом. Поскольку количество weval-e-переменных ограничено, и новых переменных в блоке уравнений не порождается, этот процесс не сможет продолжаться бесконечно.

После того как все возможные уравнения будут построены, из них выделяются уравнения вида назначение, то есть такие уравнения, в левой или правой части которых стоит

единственная строковая переменная. Эти уравнения переносятся в структуру назначений алгоритма решения уравнений, а в основной структуре вместо этой строковой переменной во все уравнения, содержащие ее, подставляется равное ей выражение, после чего все измененные уравнения по возможности расщепляются и анализируются на противоречия (алгоритмами, описанными в Разделах 4.1.2 и 4.2 данного Приложения). Так продолжается до тех пор, пока в основном блоке не останется уравнений вида назначение, либо не будет найдено противоречие.

В структуре решений уравнений могут порождаться только новые weval-переменные типа t (алгоритмами решения диофантовых уравнений). На каждую такую переменную может приходиться, самое большее, одно назначение. При этом появляется назначение на переменную типа e, и она более не может встречаться в основной структуре уравнений. Поэтому с подстановкой каждого очередного назначения в основную структуру количество переменных типа e в основной структуре будет уменьшаться. Это гарантирует конечность времени работы этой части алгоритма обработки уравнений в словах.

После того как работа с уравнениями завершена, полученный в ее результате блок назначений переводится обратно в структуры прогонки.

4.1.2. Алгоритм разделения уравнения на подуравнения

Может оказаться так, что одно длинное уравнение можно разделить на несколько коротких исходя из его структуры. Поскольку процессы решения уравнений существенно замедляются с ростом длины уравнения, имеет смысл осуществлять простую проверку возможности разбивки уравнения на подуравнения как можно чаще.

Предварительно от двух частей каждого уравнения отщепляются равные термы и переменные слева и справа. Таким образом, все уравнения, подающиеся на вход функции выделения подуравнений, имеют вид

```
(AreEqual (('F') (e.Complexity1)
      (t.1 e.Expression1))(('F')(e.Complexity2)(t.2 e.Expression2)))
```

Причем `t.1` не совпадает с `t.2`, либо

```
(AreEqual (('F')(e.Complexity)
           ((weval e t.name) e.Expression))(('F') ((0 const))())).
```

Если переменных типа `(weval e t.name)` в уравнении не нашлось, значит, оно может быть расщеплено на подуравнения длины 1, если `e.Complexity1=e.Complexity2`, либо

противоречиво.

На каждом шаге функция разделения уравнения на подуравнения в первую очередь проверяет, не оказались ли одинаковы мультимножества переменных уже просмотренных частей левого и правого выражения. Если они одинаковы (и не равны `(0 const)`), то из соотношений на длины левых и правых частей следует, что исходное уравнение

```
e.ProcessedLeft e.NonProcessedLeft
                                = e.ProcessedRight e.NonProcessedRight
```

образует два уравнения:

```
e.ProcessedLeft = e.ProcessedRight
```

и

```
e.NonProcessedLeft=e.NonProcessedRight,
```

второе из которых опять можно попробовать расщепить.

Если хотя бы одна из непросмотренных частей уравнения пуста, переходим к оценке этого уравнения. Иначе отщепляем от каждой части уравнения спереди по терму, переносим его в просмотренную часть и добавляем в мультимножество кратностей просмотренных частей, после чего опять проводим процедуру попытки разделения.

Если оказалось, что в полученном уравнении одна из частей пуста, а другая содержит константы – порождаем противоречие. Если другая содержит только переменные типа e – присваиваем им всем пустое значение.

Если одна из частей содержит только константы, значит, имеется линейное диофантово уравнение на длины значений переменных, имеющее конечное количество решений в натуральных числах. Разрешая его, получаем, возможно, несколько вариантов сужений вида

```
(assign (weval e t.Name) ((weval t t.NewName1)…
                                (weval t t.NewNameN)))
```

на все e-переменные другой стороны. Если существует хотя бы один такой набор решений, расщепляем текущую конфигурацию на гроздь конфигураций, каждая из которых содержит один вариант набора сужений, и продолжаем разбор в каждой из них по отдельности; иначе возвращаем противоречие.

В 2016 году был добавлен также следующий шаг алгоритма. Если обе части содержат e-

переменные, но только с кратностью больше 1 – уравнение также передается модулю разрешения диофантовых уравнений со специальным флагом, и там проводится проверка, не является ли это уравнение противоречивым. Если не является, значит, множество его решений бесконечно, и модуль диофантовых уравнений далее это уравнение не обрабатывает. Диофантовы уравнения на длины значений переменных, соответствующие уравнениям вида

```
(AreEqual (e.1 (weval e [Name1]) e.2) (e.3 (weval e [Name1]) e.4))
```

с кратностью хотя бы одной weval-переменной, равной единице, всегда имеют решения, поэтому такие уравнения не рассматриваются этим шагом анализа изначально.

Если с одной стороны оказалась одна переменная типа e - присваиваем ей значение, стоящее с другой стороны (если бы эта же переменная имела вхождения в это значение, выполнился бы один из предыдущих случаев). Во всех прочих случаях возвращаем полученное уравнение без изменений.

Такая процедура расщепления уравнения на подуравнения проводится суперкомпилятором MSCP-A слева и справа.

4.2. Обобщенный алгоритм разрешения квадратичного уравнения

Пусть дано уравнение

```
(AreEqual ((s.Log) (e.Complexity1)(e.LHS))
          ((s.Log) (e.Complexity2)(e.RHS))),
```

где кратность каждой переменной в `(e.Complexity1)U(e.Complexity2)` не больше 2. Мощность `e.Complexity1=N`, мощность `e.Complexity2=M`. Тогда установить, имеются ли у этого уравнения корни, можно с помощью следующей процедуры.

Рассматриваем два первых терма `e.LHS` и `e.RHS`, Считаем, что они неравны и что хотя бы один из них – e-переменная (это гарантируется алгоритмом разделения уравнения на подуравнения). Пусть первый терм `e.LHS` – это переменная `(weval e.X)`. Тогда:

1. если первый терм `e.RHS` – это `(weval e.Y)`, по лемме Леви [12] существует три возможности.
   1.1. `(weval e.X) = (weval e.Y)`
   1.2. `(weval e.X) = (weval e.Y)(weval e.X1)`

1.3. `(weval e.Y) = (weval e.X)(weval e.Y1).`

2. если первый терм `e.RHS` – это t.1, по лемме Леви [12] существует две возможности.

   2.1. `(weval e.X) = empty`

   2.2. `(weval e.X) = t.1 (weval e.X1)`

Производим соответствующие замены и удаляем равные префиксы дочерних уравнений. Совокупные длины их левых частей будут в случае 1.1 равны M+N-2, в случае 1.2 равны M+N-2+multiplicity(`e.X`), в случае 1.3 равны M+N-2+multiplicity(`e.Y`), в случае 2.1 равны M+N-multiplicity(`e.X`), в случае 2.2 равны M+N-2+multiplicity(`e.X`). В любом случае эти совокупные длины не превысят M+N в силу исходных ограничений на кратности переменных. Переименуем `e.X1` в `e.X`, `e.Y1` в `e.Y` и рассмотрим дочерние уравнения, пользуясь тем же самым алгоритмом. Назовём его прогонкой уравнения.

Рано или поздно всякая конфигурация в вершине дерева прогонки исходного уравнения либо повторит конфигурацию в какой-нибудь вершине-предке (в этом случае прогонка вершины более не осуществляется), либо будет содержать тождественно истинное уравнение, либо будет содержать противоречие.

Если хотя бы одна конфигурация в дереве прогонки уравнения содержит тождественно истинное уравнение, исходное уравнение имеет решения. В таком случае переходим к рассмотрению следующего уравнения в конфигурации. Иначе исходное уравнение решений не имеет, и содержащая его конфигурация противоречива.

Случай 2.2 алгоритма решения квадратичных уравнений выше может содержать в себе скрытое противоречие, связанное со структурой данных входного языка. А именно, если `t.1` есть выражение в скобках, содержащее значение, которому была сопоставлена строковая переменная `(weval e X)`, то равенство 2.2 невозможно. Поэтому на каждом шаге развертки квадратичного уравнения по лемме Леви [12] производится проверка на противоречия такого рода. Ранее, в 2015 году, эти проверки осуществлялись в рамках «первичного анализа уравнения на противоречия», который мог выявить подобные противоречия лишь в редких случаях.

Термы не квадратичных уравнений могут подвергнуться подобной проверке лишь в крайней левой и крайней правой позиции. А именно, если переменная `(weval e X)` может быть сопоставлена только с пустым словом либо выражением, в качестве подвыражения имеющим `(weval e X)`, не противоречив лишь первый вариант. Алгоритм расщепления не может распознать второй случай, если выражение, соответствующее переменной `(weval e X)`,

заключено в скобки, как, например, в равенстве

```
(AreEqual (('*' (weval e X)) 'A' (weval e X))
          ((weval e X)(weval e Y) (weval e Z))).
```

Поэтому такие проверки осуществляются независимо от него, в том случае, если уравнение не принадлежит к квадратичным.

5. Алгоритмы поиска вложения и обобщения параметризованных конфигураций

5.1. Алгоритм поиска вложений на полной конфигурации узла при развертке

В 2016 году в рамках работ по Проекту существенно уточнены алгоритмы вложения на слоистых стеках (таким образом, чтобы отношение Турчина на этом типе стека согласовывалось с его определением, данным В.Ф. Турчиным [24], [25]). В результате отношение Турчина стало возможно использовать не только для поиска вложений, но и для построения более точных обобщений конфигураций.

Конфигурация узла в дереве развертки программы есть

```
(Node [Status] [Name] (([Restrictions]) ([letExpressions])
                       ([equations]) ([term])[Stack])) [Children]).
```

Если блок `[letExpressions]` непуст, значит, узел описывает уже совершенное обобщение либо вложение. Такие узлы алгоритмами вложения и обобщения не рассматриваются. Поэтому вложение осуществляется только на оставшихся структурах данных: а именно сужениях, уравнениях, стеке и терме конфигурации.

Конфигурация C_1 узла *точно вкладывается* в конфигурацию C_0 его предка, если она совпадает с C_0 с точностью до временных меток и имен (но не типов) параметров, причем существует подстановка имен параметров конфигурации C_0 в имена параметров конфигурации C_1 такая, что каждый параметр конфигурации C_0 отображается в единственный параметр конфигурации C_1.

Пример 5.1.1

Конфигурация `<F e.x e.x>` точно вкладывается в конфигурацию `<F e.x e.y>`, но не наоборот. В конфигурацию `<F t.x e.x>` не вкладывается точно ни одна из перечисленных выше, из-за несовпадения типов параметров в аргументе вызова.

Определение 5.1.1 Конфигурация C_0 узла-предка *вкладывается по Крускалу* в конфигурацию C_1 узла-потомка, если выполнен один из следующих случаев:

1. C_1 точно вкладывается в C_0.
2. C_1 можно представить как конкатенацию выражений C_{11}' C_{12}' C_{13}'(возможно пустых), причем конфигурация C_0 вкладывается по Крускалу в C_{12}'.
3. C_1 есть вызов функции с аргументом C_{11}', причем C_0 вкладывается по Крускалу в C_{11}'.
4. $C_1 = (C_{11}')$, C_0 вкладывается по Крускалу в C_{11}'.
5. $C_1 = C_{11}'C_{12}'$, $C_0 = C_{11}'C_{01}'$, причем C_{01}' вкладывается по Крускалу в C_{12}'.
6. $C_1 = C_{11}'C_{12}'$, $C_0 = C_{01}'$ C_{12}', причем C_{01}' вкладывается по Крускалу в C_{11}'.
7. C_1 = <[FunctionName] C_{11}'>, C_0 = <[FunctionName] C_{01}'>, причем C_{01}' вкладывается по Крускалу в C_{11}'. В MSCP-A вызов функции, помимо имени, содержит еще уникальную временную метку [TimeStamp], но при проверке вложения по Крускалу временные метки не учитываются.
8. $C_1 = (C_{11}')$, $C_0=(C_{01}')$, причем C_{01}' вкладывается по Крускалу в C_{11}'.

Данное выше определение вложения по Крускалу полностью игнорирует имена параметров, поскольку два любых параметра всегда точно вкладываются друг в друга, а шаги 5-8 позволяют свести проверку на вложение двух конфигураций, совпадающих по модулю параметров, к проверке на вложение единичных параметров.

Определение 5.1.2 Конфигурация S_0 стека в узле *вкладывается по Турчину* в конфигурацию стека S_1 узла-потомка, если выполнено следующее условие.

1. Выделяется максимальная общая часть стеков S_0 и S_1 – это все слои стека S_1, содержащие первым элементом присваивание вида

```
(assign (var l [VarName])
        ((call [FunctionName TimeStamp] e.Arg))),
```

имеющие метку `[TimeStamp]` такую же, как и некоторый элемент стека S_0, и наоборот. Эта общая часть далее не рассматривается.

2. Во всех оставшихся слоях отбрасываются все присваивания, кроме первого, а также те первые их элементы, которые не имеют вид
   `(assign (var l [VarName])((call [FunctionName Time] e.Arg))).`
   Все оставшиеся присваивания представляются лишь именами функций, вызовы которых они содержат.
3. Если строка из имен функций стека S_0, полученная таким образом, есть префикс строки из имен функций стека S_1, то стек S_0 вкладывается в стек S_1 по Турчину.

Пример 5.1.2

Пусть узел-потомок имеет слоистый стек

```
((assign (var l 3)
   ((call (f 6)(args (arg (call (f 7)(args (arg (par e y))))))))
 (assign (var l 4)((call (g 3) (args (arg 'A' (par e y)))))))
((assign (var l 2)((var l 3)(par e x)(var l 4))))
((assign (var l 1)((call (f 1)(args (arg (var l 2)))))))),
```

узел-предок имеет стек

```
((assign (var l 2)((call (g 2) (args (arg (par e x)))))))
((assign (var l 1)((call (f 1)(args (arg (var l 2)))))))),
```

что соответствует конфигурациям `<F <F e.y> <G 'A' e.y>>` и `<F <G e.x>>`. Вторая вкладывается в первую по Крускалу, но не по Турчину, поскольку ее представление в виде строки имен активных функций (без учета общего контекста, то есть внешнего вызова функции `F`) – есть `G`, а строковой представление стека первой (без учета общего контекста) есть `F`.

Чтобы конфигурации из двух узлов в дереве вычисления программы считались кандидатами на обобщение, необходимо, чтобы структура `[term]` узла-предка вкладывалась в структуру `[term]` узла-потомка по Крускалу, а структура `[stack]` узла-предка в структуру `[stack]` узла-потомка – по Турчину. Анализ отношения Турчина в рамках предложенной в рамках Проекта модели многослойных префиксных грамматик верифицирует наличие таких узлов на любом бесконечном пути в дереве вычислений программы.

Для того, чтобы определить *вложения в структуре уравнений*, будем считать, что уравнение имеет слишком высокую сложность вложения, если оно содержит какие-либо строковые переменные, отличные от параметров, либо количество всех термов в нем на всех уровнях (т.е.

включая термы, стоящие в скобках) превышает некоторое заранее заданное N, устанавливаемое эмпирически.

Определение 5.1.3 Скажем, что структура уравнений `e.Equations1` вкладывается в структуру уравнений `e.Equations2`, если для всякого уравнения $A_1=B_1$ из `e.Equations1` с не слишком высокой сложностью вложения найдётся уравнение $A_2=B_2$ из `e.Equations2` (произвольной сложности) такое, что A_1 является подпоследовательностью A_2 и B_1 является подпоследовательностью B_2 с точностью до переименования параметров, либо A_1 является подпоследовательностью B_2, а B_1 – подпоследовательностью строки A_2, либо наоборот. В любом случае обобщением этих двух уравнений (возможно, не минимальным) будет исходное уравнение $A_1=B_1$.

Величина N варьируется в зависимости от свойств программы. В частности, если допустить N=1, структура уравнений будет игнорироваться вообще (поскольку все уравнения такого вида попали бы в структуру сужений).

Поскольку количество уравнений, имеющих не слишком высокую сложность вложения, конечно, определяемое отношение вложения не допускает бесконечно длинных плохих последовательностей.

Сужения также могут рассматриваться как уравнения вида

```
(AreEqual ([Parameter])([Expression])).
```

Хотя блок сужений описывает уравнения на параметры, которые подверглись замене, рассматривать его как продолжение блока уравнений при поиске вложений может быть целесообразно в целях построения более точных аппроксимаций циклов. Эта возможность опциональна и работает лишь при назначении значения N>1.

5.2. Алгоритм обобщения параметризованных конфигураций

Пусть оказалось, что конфигурация из узла-предка вкладывается в конфигурацию из узла-потомка по Крускалу и по Турчину. От суперкомпилятора требуется построить обобщение этих конфигураций.

Даже в языках без ассоциативной конкатенации обобщение в смысле построения наиболее узкого шаблона для двух термов неоднозначно, если иметь ввиду декомпозицию конфигураций при обобщении. Например, термы `f(g(f(x)))` и `f(x)` могут породить либо подстановку `Let g(f(x))=u in f(u)`, либо подстановку `Let f(x)=w in f(g(w))`. Эту неопределенность

в рамках реализации MSCP-A удалось решить в 2016 году, используя отношение Турчина. Отношение Турчина дает подсказку о размере общего контекста, которой используемся в следующем смысле.

Пусть вызовы функций в N слоях стеков конфигурации-потомка сохранили временные метки вызовов конфигурации-предка. Заменив аргумент первого вызова на N-ом слое стека свежей свободной переменной в стеках вызовов функций потомка и предка, построим две пары термов: термы-контексты, содержащие свежие свободные переменные, и термы-аргументы, являющиеся значениями этих переменных. Далее строим обобщения для этих двух пар параллельно.

Пример 5.2.1

Причина, по которой контексты могут также потребовать обобщения – пассивная развертка значений параметров. Кроме того, намеренно не считаются общими контексты соседние к аргументу первого вызова на N-ом уровне стека вызовы. Поясним это на примере. Рассмотрим вычисление числа Фибоначчи с помощью функции

```
1 F {
2  'SS' e.1 = <F 'S' e.1> <F e.1>;
3  'S' = 'S';
4      = 'S';
5 }
```

Пусть вычисляется терм `<F <F e.x> <F e.x>> e.x`. Поскольку мы используем слоистый стек, прогонка по первому правилу определения функции `F` породит терм

`<F <F 'S' e.x1> <F e.x1> <F 'S' e.x1> <F e.x1>> 'SS' e.x1.`

Верхний вызов функции `F` является общим контекстом, поэтому отношение Турчина подскажет следующее разбиение:

`Let w=<F e.x> <F e.x> in <F w> e.x;`

`Let u=<<F 'S' e.x1> <F e.x1><F 'S' e.x1><F e.x1>> in <F u> 'SS' e.x1,`

После чего будут обобщаться отдельно значения `w` и `u` и содержащие их термы. В конце полученные нетривиальные обобщения для `w` и `u` будут подставлены в обобщение для контекста.

В примере выше факт наличия пар равных вызовов в конфигурациях удалось сохранить при разрезании ее на контекст и активную часть. Если бы эти равные вызовы возникли случайно при пассивной развертке на одном из слоев ниже, чем N-ый, информация о них была бы потеряна.

После того как выделен общий контекст, временные метки при обобщении далее не учитываются.

В языках с ассоциативной конкатенацией неоднозначность обобщения не разрешается одним только выделением общего контекста. Например, обобщение двух объектных термов `'ABA'` и `'ABABA'` может быть записано как `'A' e.1 'B' e.1 'A'`, так и как `'ABA' e.1` – ни одно из них не может быть уточнено еще больше. Поэтому суперкомпилятор MSCP-A последовательно строит возможные варианты обобщений и сравнивает их между собой с помощью оценочной функции, задаваемой эмпирически. То обобщение, оценка которого оказалась выше, рассматривается как рабочий вариант обобщения. При этом используются следующие допущения:

1. Каждое совпадение подстановок в обобщении (что означает сохранение информации о равенстве термов) сильно повышает оценку этого обобщения.
2. Каждое обобщение до e-параметра выражения, не являющегося вызовом или e-параметром, ухудшает оценку обобщения.
3. Обобщение же двух параметров до параметра того же типа повышает оценку обобщения.

Пример 5.2.2

Выражения

`<F 'I' e.x> <F e.x> <F 'I' e.x>` и

`<F 'I' e.y> <F e.y> <F 'I' e.y> <F 'I' e.y> <F e.y>`

могут быть обобщены до выражения `<F 'I' e.w> <F e.w> <F 'I' e.w> e.u`, где выражение `e.u` пусто для первого выражения и равно `<F 'I' e.y> <F e.y>` в случае второго. А могут быть обобщены до выражения `e.u <F e.w> e.u`, где выражение `e.u` равно `<F 'I' e.x>` в случае первого терма и `<F 'I' e.y> <F e.y>` в случае второго. Все вызовы в этих выражениях находятся в одном слое стека, поэтому уточнение по контексту не дает никакой информации. Но применение оценочной функции показывает, что второе обобщение несколько более предпочтительно не только потому, что в нем есть совпадение подстановок слева и справа от выражения `<F e.w>`, но и потому, что в нем отсутствуют подстановки, заменяющие параметром константное выражение (заменяются только вызовы или параметры).

После того как построено обобщение в структуре `[term]` узла, строится обобщение в его

структуре уравнений. Рассматриваются лишь уравнения, имеющие сложность не более, чем заранее заданная сложность N. Каждое такое уравнение может быть обобщением не более, чем конечного числа уравнений в другой конфигурации. Если подстановка, порожденная алгоритмом обобщения на предыдущем шаге, сводит хотя бы одно уравнение из второй группы к рассматриваемому простому уравнению, такое уравнение помещается в структуру уравнений обобщенной конфигурации после соответствующей подстановки. Иначе уравнение отбрасывается.

Кроме того, что в связи с использованием слоистых стеков выбор обобщения теперь больше ориентируется на наличие в его подстановках равных, в 2016 году алгоритм обобщения был модифицирован так, чтобы существенно взаимодействовать со специальной структурой данных MSCP-A – структурой уравнений. В частности, удалось частично решить проблему потери информации при обобщении символа и пустой строки до свежего e-параметра, что позволило расширить класс решаемых суперкомпилятором задач верификации ассоциативных данных (см. примеры ниже).

Пусть структура `[letExpressions]` обобщенной конфигурации содержит следующую пару let-выражений

```
Let (par e x) = s.Sym in e.w,
Let (par e x) = empty in e.w,
```

где `(par e x)` – свежий параметр. Уравнение `(AreEqual (('T')(par e x) s.Sym)(('T') s.Sym (par e x)))` помещается в структуру уравнений конфигурации, содержащей выражение, включающее `(par e x)`. Это уравнение становится тождеством тогда и только тогда, когда `(par e x)` принимает в качестве значения строку из одних символов `s.Sym` (возможно, нулевого их числа).

Такое преобразование позволяет сохранить информацию о возможных значениях параметра `(par e x)` в обобщенной конфигурации, но оно не дает суперкомпилятору права использовать эту информацию иначе, чем для отсечения заведомо недостижимых ветвей вычисления, потому что уравнение помечено флагом `'N'`.

Пример 5.2.3

Пусть требуется доказать, что строка, порождаемая функцией

```
1 F {
2  s.1 e.1 = 'A' <F e.1>;
3          = ;
4 }
```

не содержит в конце букву 'B'. Построим вспомогательную функцию

```
1 Check {
2  e.1 'B' = False;
3      e.Z = True ;
4 }
```

Осуществим суперкомпиляцию параметризованного вызова <Check <F e.x>>. Для краткости здесь пишем вызовы функций не во внутреннем представлении MSCP-A, а в виде $<F_{TimeStamp}$ e.Arg>.

Элементы исходного вызова получают временные метки, и начальная конфигурация (номер 0) есть

$<Check_0$ $<F_1$ (par e x)>>, начальный стек вызовов функций есть

((assign (var l 1)($<F_1$ (par e x)>)))

((assign (var l 0)($<Check_0$ (var l 1)>))).

Развертка по первому правилу в определении F порождает узел номер 1 со следующими данными:

Restrictions:	(assign (par e x)((par s 1)(par e 1)))
Equations:	empty
Term:	$<Check_0$ 'A' $<F_2$ (par e 1)>>
Stack:	((assign (var l 2)($<F_2$ (par e 1)>)))
	((assign (var l 1)('A' (var l 2))))
	((assign (var l 0)($<Check_0$ (var l 1)>)))

Конфигурация узла номер 0 успешно вкладывается в конфигурацию данного узла. Отношение Турчина задает следующее разрезание стека вызовов функций:

Let (par e u) = $<F_1$ (par e x)> in $<Check_0$ (par e u)>,

Let (par e u) = 'A' $<F_2$ (par e x)> in $<Check_0$ (par e u)>.

После чего варианты конфигураций для (par e u) обобщаются до конфигурации

(par e w) $<F_3$ (par e z)>

и предложенное обобщение уточняется до $<Check_0$ (par e w) $<F_3$ (par e z)>>.

Информация об этом обобщении записывается в узел номер 0, и разбирается узел номер 1 со следующими данными:

Restrictions:	empty
Equations:	(AreEqual ((par e w) 'A')('A' (par e w)))
Term:	<Check$_0$ (par e w) <F$_3$ (par e z)>>
Stack:	((assign (var l 2)(<F$_3$ (par e z)>)))
	((assign (var l 1)((par e w)(var l 2))))
	((assign (var l 0)(<Check$_0$ (var l 1)>)))

В случае, если значение (par e z) пусто, значение (par e w) не сопоставится с левой частью e.1 'B' успешно: результат сопоставления сразу же выявит противоречие в блоке уравнений.

В случае, если значение (par e z) начинается с символа, получаем узел номер 2 с данными:

Restrictions:	(assign (par e z)((par s 2)(par e 2)))
Equations:	(AreEqual ((par e w) 'A')('A' (par e w)))
Term:	<Check$_0$ (par e w) 'A' <F$_4$ (par e z)>>
Stack:	((assign (var l 2)(<F$_4$ (par e z)>)))
	((assign (var l 1)((par e w) 'A' (var l 2))))
	((assign (var l 0)(<Check$_0$ (var l 1)>)))

Эти данные успешно обобщаются с данными узла номер 1. На сей раз обобщение происходит снизу: данные узла номер 1 не меняются, а данные узла номер 2 подвергаются следующему разрезанию:
(assign (par e v)((par e w) 'A')) in <Check$_0$ (par e v)<F$_4$ (par e z)>>. После чего конфигурация <Check$_0$ (par e v) <F$_4$ (par e z)>> точно вкладывается в конфигурацию узла 1, и развертка заканчивается.
Ни одна ветвь дерева развертки этой программы не содержит константы False, что верифицирует исходное условие.

Пример 5.2.4

Можно усложнить предыдущий пример, заменив определение функции Check на следующее:

```
1 Check {
2  e.1 'B' = False;
3      e.Z = True ;
4  e.1 'A' = <Check e.1>;
5 }
```

Вплоть до построения узла 2 суперкомпилятор будет работать так же. Однако теперь добавится нетривиальная дочерняя узлу 1 ветвь на сужении `(par e z)` до пустого слова. После вычисления вызова `F` с пустым аргументом имеем следующую конфигурацию (узел 3):

Restrictions:	(Always)
Equations:	(AreEqual ((par e w) 'A')('A' (par e w)))
Term:	<Check$_0$ (par e w)>
Stack:	((assign (var 1 0)(<Check$_0$ (par e w)>)))

На правиле определения функции `Check`, содержащем образец `e.1 'B'`, она породит противоречие, на правиле с пустым образцом – породит значение `True`. А на правиле с образцом `e.1 'A'` будет порождена следующая конфигурация (узел 4):

Restrictions:	(assign (par e w)((par e 2) 'A'))
Equations:	(AreEqual ((par e 2) 'AA')('A' (par e 2) 'A'))
Term:	<Check$_4$ (par e 2)>
Stack:	((assign (var 1 0)(<Check$_4$ (par e 2)>)))

Уравнение в соответствующем блоке упростится до вида

`(AreEqual ((par e 2) 'A')('A' (par e 2))),`

и конфигурация узла 4 точно вложится в конфигурацию узла 3.

В остаточной программе опять будет отсутствовать константа `False`, что верифицирует искомое свойство.

Список использованных источников

[1] С.М. Абрамов, А.И. Адамович, Л.А. Гайдар, Г.В. Гузилова, М.Р. Коваленко, А.П. Немытых, А.Н. Непейвода, С.М. Пономарева, Е.В. Шевчук, Ю.В. Шевчук. Исследование методов функционального моделирования распределённых недетерминированных вычислительных систем и верификации функциональных моделей посредством суперкомпиляции. // Приложение к промежуточнуму отчету по Проекту РФФИ 14-07-00133-а, 2014.

[2] С.М. Абрамов, А.И. Адамович, Л.А. Гайдар, Г.В. Гузилова, М.Р. Коваленко, А.П. Немытых, А.Н. Непейвода, С.М. Пономарева, Е.В. Шевчук, Ю.В. Шевчук. Исследование методов функционального моделирования распределённых недетерминированных вычислительных систем и верификации функциональных моделей посредством суперкомпиляции. // Приложение к промежуточнуму отчету по Проекту РФФИ 14-07-00133-а, 2015.

[3] С.М. Абрамов, А.И. Адамович, Л.А. Гайдар, Г.В. Гузилова, М.Р. Коваленко, А.П. Немытых, А.Н. Непейвода, С.М. Пономарева, Е.В. Шевчук, Ю.В. Шевчук. Исследование методов функционального моделирования распределённых недетерминированных вычислительных систем и верификации функциональных моделей посредством суперкомпиляции. // Технический итоговый отчет по Проекту РФФИ 14-07-00133-а, 2016.

[4] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proiett. Proving Horn Clause Specifications of Imperative Programs, 2015.

[5] N. Bjorner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009), vol. 5505 of LNCS, pp: 307-321. Springer Berlin Heidelberg, 2009.

[6] Г.С. Маканин. Проблема разрешимости уравнений в свободной полугруппе. // *Матем. сб.*, 103(145):2(6) (1977), Стр. 147–236, 1977.

[7] Greibach, Sheila. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. // *Journal of the ACM* **12** (1), January 1965.

[8] G. Higman. Ordering by divisibility in abstract algebras. // Proc. London Math. Soc. 2(7), pp: 326-336, 1952.

[9] J.E. Hopcroft, J. D. Ullman. Introduction to Automata Theory, Languages, and Computation. // Addison-Wesley, 1979.

[10] A. Jez. One-Variable Word Equations in Linear Time, in the Proc. of Automata, Languages, and Programming, LNCS, Vol. 7966, pp: 324-335, 2013.

[11] A. Jez. Recompression: Word Equations and Beyond. In the Proc. of Developments in Language Theory, LNCS, Vol. 7907, pp: 12-26, 2013

[12] J.Karkhumaki. Combinatorics of words, p.24 // https://www.utu.fi/en/units/sci/units/math/staff/Documents/karhumaki/combwo.pdf

[13] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. // Trans. Amer. Math. Society. Vol. 95, pp: 210-225, 1960.

[14] A. Lisitsa and A.P. Nemytykh. Verification as a Parameterized Testing (Experiments with the SCP4 Super-compiler) // J. Programming and Computer Software , Vol. 33, No.1, pp: 14-23, 2007.

[15] A.P. Lisitsa, A.P. Nemytykh. Reachability Analysis in Verification via Supercompilation // International Journal of Foundations of Computer Science (IJFCS), 2008, Vol. 19, No. 4, pp: 953-969, August 2008.

[16] A.P. Lisitsa, A.P. Nemytykh. Experiments on verification via supercompilation // Интернет-ресурс http://refal.botik.ru/protocols/ , 2007-2009.

[17] Модельный суперкомпилятор MSCP-A. // Интернет-ресурс: http://refal.botik.ru/mscp/ , 2016.

[18] А.П. Немытых Суперкомпилятор SCP4: общая структура // М.: УРСС, 152 стр., 2007.

[19] Антонина Н. Непейвода. Отношение Турчина и аппроксимация циклов при анализе программ. // Сборник трудов по функциональному языку программирования Рефал. Том №1, Стр. 170-192. / Переславль-Залесский: Изд-во «Сборник», 2014, ISBN 978-5-9905410-1-6. / http://refal.botik.ru/library/refal2014_issue-I.pdf

[20] Antonina Nepeivoda. Turchin's Relation for Call-by-Name Computations: A Formal Approach. // In: “Proceedings of the Fourth International Workshop on Verification and Program Transformation” / Electronic Proceedings in

Theoretical Computer Science (EPTCS), Vol. 216, pp. 137-159, 2016, DOI: http://dx.doi.org/10.4204/EPTCS.216.8 , ISSN: 2075-2180

[21] J. P. Secher. Driving in the Jungle. Second Symposium, PADO2001 Aarhus, Denmark, May 21–23, 2001 Proceedings // LNCS, vol. 2053, pp. 198-217, 2001.

[22] V. F. Turchin. The Language REFAL, the Theory of Compilation, and Metasystem Analysis. Courant Institute Report #20, New York, 1980

[23] V.F.Turchin. The concept of a supercompiler // ACM Transactions on Programming Languages and Systems, 8, pp.292-325, 1986.

[24] V.F. Turchin. The algorithm of generalization in the supercompiler // Partial Evaluation and Mixed Computation, pages 341-353, 1988.

[25] V. F. Turchin. *The School "Metacomputation in the Language Refal"*. // Obninsk, July 11-23, 1990.

[26] V.F. Turchin. Program Transformation with Metasystem Transitions. The Journal of Functional Programming, Vol. 3, N. 3, pp:283-313, 1993

[27] V.F. Turchin. Refal-5, Programming Guide and Reference Manual // New England Publishing Co., 1989. (electronic version: http://www.botik.ru/pub/local/scp/refal5/ ,2000)