# Применение языка KAST для преобразования исходного кода и автоматического исправления дефектов

*Н. Л. Луговской (lugovskoy@ispras.ru),*
*С. В. Сыромятников (syrom@ispras.ru)*

**Аннотация.** В данной работе описывается расширение языка KAST для решения задачи трансформации исходного кода. В настоящее время язык KAST используется для поиска поддеревьев заданного в виде шаблона вида в синтаксических деревьях, построенных по коду на языках C/C++, Java и C#. В статье также рассматриваются некоторые существующие подходы к трансформации исходного кода и показываются преимущества использования для решения данной задачи языка KAST. Описывается метод, при помощи которого изменения в синтаксическом дереве преобразуются в изменения исходного кода.

**Ключевые слова:** трансформация кода; статический анализ; архитектура программ; язык шаблонов; KAST

## *1. Введение*

Необходимость в трансформации исходного кода возникает во многих областях программной инженерии. Целями трансформации могут быть оптимизация, рефакторинг; она может применяться в обратной инженерии. Хотя преобразование кода обычно может быть проведено вручную, удобнее использовать для него специальные автоматические инструменты, работающие в соответствии с некоторой формальной спецификацией. Как правило, в качестве подобной спецификации используется набор правил модификации для структур данных компилятора (например, синтаксического дерева). В данной работе описывается подход, в котором спецификации трансформации задаются шаблонами на языке KAST.

KAST, язык написания шаблонов для синтаксических деревьев, был описан в статье [4]. Мы не будем подробно останавливаться на синтаксисе этого языка и принципах работы соответствующего анализатора, поскольку эта информация может быть найдена в [4]. Напомним лишь, что KAST предназначен для поиска поддеревьев заданного в виде шаблона вида в синтаксических деревьях, построенных по коду на языках C/C++, Java и C#, и

в настоящее время используется для выявления синтаксических структур, представляющих собой дефекты исходного кода. Однако синтаксис языка KAST после небольшого расширения позволяет использовать его для написания процедур трансформации кода; при этом шаблоны будут задавать фрагменты кода, подлежащие замене, а описываемые в данной статье расширения языка позволят определять те конструкции, на которые их следует заменять. Подобный механизм представляется весьма полезным для разработчиков, так как, с одной стороны, опирается на знание синтаксической структуры программы, а с другой — является легко настраиваемым в соответствии с потребностями конкретного пользователя.

В рассматриваемом подходе изменяемые фрагменты кода представляют собой узлы синтаксического дерева. Представляется логичным, чтобы преобразованный код также сначала конструировался из синтаксических узлов и лишь затем преобразовывался в текстовое представление. Поэтому требуемые расширения языка KAST сводятся к добавлению средств создания узлов синтаксического дерева (конструкторов) и функций, осуществляющих замену одного узла на другой, а также вставку и удаление узлов в синтаксическом дереве. Соответствующим образом доработанный KAST-анализатор в настоящее время существует в виде прототипа и, в отличие от обнаружителя дефектов, не входит в состав продуктов компании Klocwork. Однако полученные с его помощью результаты представляются авторам достаточно содержательными и достойными освещения в рамках отдельной статьи.

Проиллюстрируем возможности рассматриваемого подхода на нескольких примерах.

***Пример 1.*** Заменить тип видимости для всех публичных полей данных в классе на приватный, одновременно добавив публичные методы для получения и установки значения данного поля.

Очевидно, что в результате преобразования из примера 1 мы вероятнее всего получим синтаксически некорректный код. Нам потребуются дополнительные изменения в коде, которые, вообще говоря, сложно свести к одному правилу. В частности, будет полезно такое автоматическое преобразование:

***Пример 2.*** Заменить все непосредственные присваивания полям классов, осуществляемые не из методов этих классов, на функциональные вызовы. Имеются в виду вызовы функций установки значения полей, сгенерированные в результате преобразования из примера 1.

***Пример 3***. Преобразование кода в соответствии с гипотетическим Coding Style: добавить к именам всех параметров всех конструкторов префикс *i_*.

Преобразование соответствующего шаблону фрагмента кода можно рассматривать и в несколько ином аспекте. Если шаблон описывает некорректные языковые конструкции, то преобразование может состоять в наиболее вероятном исправлении обнаруженного дефекта. Разумеется,

предложенный вариант может не совпадать с тем, что на самом деле намерен написать пользователь, но рекомендация по возможному исправлению ошибки может оказаться полезной. Описываемый подход иллюстрирует

***Пример 4.*** Найти в коде на языке C определения всех функций, для которых не указан тип возвращаемого значения, и добавить в эти определения ‘int’ в качестве типа результата. В языке C, в отличие от современных реализаций C++, отсутствие спецификации типа у функции или переменной означает, что функция возвращает значение типа ‘int’, а переменная, соответственно, принадлежит к этому типу. Подобные умолчания затрудняют понимание кода и делают его более трудным для сопровождения, а предлагаемое в данном примере автоматическое преобразование суть наиболее очевидное решение данной проблемы.

## 2. Существующие решения

Рассмотрим некоторые существующие инструменты трансформации исходного кода. Все описанные ниже системы работают по одному и тому же принципу: после синтаксического анализа кода к построенным структурам (синтаксическому дереву или его аналогам) применяется набор правил вида:

*старая конструкция → новая конструкция*

Далее по изменённому внутреннему представлению генерируется программный код.

Главное отличие предлагаемого нами подхода от данного принципа состоит в том, что в шаблоне на языке KAST можно определить не только конструкцию, подлежащую замене, но и контекст, в котором она должна находиться, чтобы быть заменённой: шаблон на языке KAST может описывать достаточно большое поддерево синтаксического дерева, и при этом лишь небольшая часть этого поддерева будет реально заменена. Кроме того, язык KAST позволяет обращаться к семантической информации внутри шаблона, что существенно повышает точность поиска требуемых синтаксических структур.

### *2.1. DMS: Software Reengineering Toolkit*

DMS представляет из себя набор инструментов для автоматического анализа, трансформации и генерации программного кода ([1]). Система написана на языке PARLANSE, который позволяет производить символьные вычисления параллельно.

Преобразование исходного кода в DMS описывается набором правил. В общем виде правило преобразования может быть записано следующим образом:

```
LHS → RHS [ if_condition ]
```

, где *LHS* (левая сторона) и *RHS* (правая сторона) представляют собой шаблоны в виде завершенных конструкций языка программирования с метапеременными, а *if_condition* описывает условие, при котором должно осуществляться преобразование, в зависимости от значений переменных из *LHS*. Условие, в частности, может содержать вызовы функций, написанных на языке PARLANSE.

Типичное правило преобразования выглядит так:

```
default domain C.

rule auto_inc(v:lvalue):

statement->statement =

"\v = \v+1;" rewrites to "\v++;"

if no_side_effects(v).
```

В соответствии с данным правилом все конструкции вида *X=X+1* будут заменены на *X++*, где X - выражение, допустимое в качестве левой части оператора присваивания.

## 2.2. Stratego/XT

Stratego/XT ([2]) представляет собой программную среду для разработки систем трансформации кода. Она состоит из двух частей: Stratego, языка описывающего трансформацию программы, и XT - набора инструментов для трансформации.

Stratego/XT использует формат *Annotated Term* (*ATerm, аннотированный терм*) для представления синтаксических деревьев. Терм может быть числом, строкой, списком или *конструктором* вида *C(t1, ..., tn)*, где все *ti* суть тоже термы. Так, выражение *4 + f(5 * x)* представляется следующим термом формата ATerm: *Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))*. Трансформация программ описывается с помощью правил, состоящих из терма-шаблона (определяющего, что надо заменить) и замещающего терма (соответствующего изменённому коду). Каждое правило имеет вид:

```
L: p1 -> p2
```

, где *L* - имя правила, *p1* - терм-шаблон, а *p2* - замещающий терм. Так, в соответствии с правилом

```
E : Eq(x, Int("0")) -> Not(x)
```

терм *Eq(Call("foo", []), False)* будет заменён на *Not(Call("foo", []))*, при этом переменной "x" будет сопоставлен терм *Call("foo", [])*.

## 3. Синтаксис языка KAST

Язык KAST позволяет описывать структуру поддеревьев синтаксического дерева в виде шаблонов. Шаблон, написанный на KAST'e, имеет следующий вид (##):

*// Tun_1* **[** *Спецификация_1,1* **]** ... **[** *Спецификация_1,k1* **]** **/**

*Квалификатор_1* **::** *Tun_2* **[** *Спецификация_2,1* **]** ... **[**

*Спецификация_2,k2* **]**

...

*Квалификатор_n* **::** *Tun_n* **[** *Спецификация_n,1* **]** ... **[**

*Спецификация_n,kn* **]**

Здесь *курсивом* обозначены метапеременные, а **полужирным начертанием** выделены элементы языкового синтаксиса.

В (##) *Tun_i* - это имя одного из типов узлов дерева или звёздочка ('*'), что соответствует произвольному имени типа. *Квалификаторы* – это имена дочерних ссылок узла или некоторые предопределённые оси: *parent, ancestor, descendant, following-sibling*.

*Спецификации* могут быть присваиваниями (в них определяются переменные) и ограничениями (они суживают множество подходящих узлов). *Ограничения* могут содержать числовые значения и операции с ними, логические операции, строковые выражения и *подшаблоны*, которые соответствуют формату (##) за тем исключением, что начинаются сразу с квалификатора. В ограничениях могут использоваться вызовы функций, как предопределённых, так и определяемых пользователем. *Присваивания* служат для объявления и инициализации *переменных*; переменная может инициализироваться любым выражением, удовлетворяющим условиям на синтаксис ограничений.

Например, следующий шаблон соответствует всем присваиваниям, являющимся аргументами оператора sizeof (вида *sizeof (a = b)*):

*// UnaryExpr [ Op::Op [ @Code = KTC_OPCODE_SIZEOF ] ]*

*/ Expr::ParensExpr / Expr::BinaryExpr [ Op::Op [ @Code =*

*KTC_OPCODE_ASSIGN ] ]*

По сравнению с версией языка KAST, описанной в статье ([4]), в существующей его реализации добавилась поддержка *коллекций*. Коллекция представляет собой упорядоченную совокупность элементов различной природы. Каждый элемент коллекции является выражением произвольного

типа: численного, логического, семантического (ссылки на семантические элементы) или синтаксического (ссылки на узлы синтаксического дерева). Коллекция может быть как получена в результате вызова некоторой функции, так и задана явно при помощи фигурных скобок: *{1, 'abc'}*. Для работы с подобными совокупностями существует набор встроенных функций (добавления и удаления элементов, проверки наличия заданного элемента и т. п.), а также специальная конструкция for, позволяющая выполнить некоторую типовую операцию с каждым элементом. В нижеприведённом примере коллекция сначала явно определяется, а затем поэлементно распечатывается:

…

*[ $coll := { 1, 2, 'three', false() } ]*

*[ for($iter, $coll, $iter.println()) ]*

Переменная *$iter* здесь является *итератором*, то есть каждый раз указывает на очередной элемент коллекции *$coll*.

В данной статье описываются расширения языка KAST, позволяющие создавать новые узлы, отсутствовавшие в изначально построенном синтаксическом дереве, и производить модификацию синтаксического дерева путём замены, добавления и удаления узлов.

## 4. Конструкторы узлов

Для создания новых узлов синтаксического дерева вводится оператор **new**. Его синтаксис подразумевает, что после ключевого слова ”new” идёт тип создаваемого узла, после которого в круглых скобках в произвольном порядке следуют инициализаторы дочерних ссылок и атрибутов:

**new** *TypeName***(***Child1* **:** *Node1***,**

…

*ChildN* **:** *NodeN***,**

***@****Attribute1* **:** *Value1***,**

…

***@****AttributeM* **:** *ValueM***)**

Нетрудно заметить, что синтаксис создания узла дерева в расширенном языке KAST схож с синтаксисом создания объекта в языках C++ и Java.

Проиллюстрируем сказанное на примере:

```
// IdExpr [ new BinaryExpr(Left : new LiteralExpr(@Value : 1),
                           @Op : KTC_OPCODE_ADD,
                           Right : this())
       ]
```

В результате применения данного шаблона для всякого выражения-идентификатора *X* будет построено бинарное выражение *1 + X* (константа *KTC_OPCODE_ADD* в языке KAST соответствует бинарному плюсу). Отметим, что фактического преобразования кода в соответствии с данным шаблоном не произойдёт, так как в этом шаблоне отсутствует вызов функции замены узлов в дереве.

Для некоторых дочерних ссылок и атрибутов возможны значения по умолчанию, которые используются, если данная ссылка или данный атрибут не были инициализированы явно. В случае ссылок значения по умолчанию допустимы для тех из них, где синтаксис языка программирования допускает отсутствие соответствующего данной ссылке поддерева (отсутствие аргументов у вызываемой функции, отсутствие инициализатора в декларации переменной и т. п.). К примеру, оператор new для создания (в терминах синтаксического дерева) вызова функции «foo» без аргументов может выглядеть так:

```
new CallExpr(Func : new IdExpr(@Value : 'foo'))
```

В случае, когда в конструируемом поддереве некоторое множество узлов должно быть объединено в однородный список, следует использовать коллекции. Конструктор вызова функции “foo” с параметрами “1” и “2” будет выглядеть так:

```
new CallExpr( Func : new IdExpr(@Value : ‘foo’),
              Args : {new LiteralExpr(@Value : 1), new
LiteralExpr(@Value : 2)}
            )
```

## 5. Функции модификации синтаксического дерева

Рассмотрим произвольное корневое ориентированное дерево (например, синтаксическое дерево, построенное по одной единице компиляции). Выделим в нём некоторую вершину N. Существует конечное непустое множество поддеревьев исходного дерева, для которых вершина N является корнем.

Среди этого множества поддеревьев существует максимальное. Очевидно, оно включает в себя все вершины, достижимые из вершины N в исходном ориентированном дереве. Описанное максимальное поддерево мы будем называть *листовым поддеревом (с корнем N).*

Описанные ниже функции модификации синтаксического дерева оперируют именно листовыми поддеревьями, а не отдельно взятыми узлами. Иными словами, при удалении или добавлении узла удаляется (соответственно, добавляется) не только сам узел, но и все его дочерние узлы, дочерние узлы его дочерних узлов и так далее. При замене одно листовое поддерево целиком подменяется другим.

Функция, осуществляющая замену одного (существующего) листового поддерева в синтаксическом дереве на другое (как правило, вновь созданное), имеет следующий синтаксис:

**substitute(***Old_node***,** *New_node***)**

Дополним пример из предыдущей части:

```
// IdExpr [ $new_node := new BinaryExpr(Left : new
LiteralExpr(@Value : 1),
                                        @Op :
KTC_OPCODE_ADD,
                                        Right : this())
    ]
    [ substitute(this(), $new_node) ]
```

В соответствии с данным шаблоном всякое выражение-идентификатор *X* будет заменено на бинарное выражение *1 + X*.

В случае, когда некоторое листовое поддерево требуется заменить на несколько других поддеревьев, образующих в синтаксическом дереве однородный список, следует использовать коллекции:

**substitute(***Old_node***, {***New_node_1***, …, ***New_node_n***})**

Функция вставки листового поддерева в дерево позволяет добавить поддерево в уже имеющийся в дереве однородный список узлов (например, список деклараций и операторов в теле функции). В качестве параметра указывается узел (т. е. корень некоторого другого поддерева), перед которым следует произвести вставку:

**insert(***Node_after***,** *New_node***)**

Возможно добавление в список сразу нескольких поддеревьев, для чего также следует использовать коллекции:

**insert(***Node_after***, {***New_node_1***, …,** *New_node_n***})**

Подчеркнём, что в описании всех четырёх приведённых выше функций *Old_node* и *Node_after* - это корневые узлы уже существующих поддеревьев, найденных в процессе сопоставления шаблона с конкретным синтаксическим деревом, а все *New_node* могут быть как вновь созданными поддеревьями, так и уже существующими, если требуется изменить их положение в дереве.

Удаление поддеревьев из дерева осуществляется при помощи функции **delete**:

**delete(***Existing_node***)**

## 6. Модификация исходного кода

В результате работы функций модификации синтаксического дерева в него вносятся изменения, которые далее необходимо трансформировать в изменения программного кода. Важно не просто получить код для новых узлов дерева, но и правильно вставить его в исходную программу, сохранив пользовательский стиль, комментарии и макровызовы. Стоит заметить, что вновь созданные поддеревья конструируются из узлов двух видов: часть узлов берётся из исходного дерева, а часть создаётся с помощью конструкторов. При этом узлы, изначально существовавшие в исходном дереве, обязательно целиком образуют одно или несколько *листовых поддеревьев*, являющихся частью подобного нового поддерева.

Мы не будем вдаваться в подробности построения синтаксического дерева программы, они описаны в статье [3]; отметим лишь, что наряду с синтаксическим деревом в нашем инструменте сохраняется соответствующая этому дереву последовательность лексем исходного кода, которая обычно удаляется после стадии синтаксического анализа. При этом каждый узел синтаксического дерева имеет ссылку на свою начальную и конечную лексемы. Данный подход позволяет предельно точно сохранить изначальный стиль исходного кода, а также комментарии и макровызовы для каждого из узлов синтаксического дерева.

Рассмотрим, каким образом замена одного поддерева другим преобразуется в модификацию исходного кода программы, отметив, что изменение кода при вставке и удалении поддеревьев происходит аналогично.

Чтобы сохранить информацию о том, что в определенном месте синтаксического дерева некоторое листовое поддерево было заменено на новое, ребро, ведущее к этому поддереву, помечается специальной меткой. В метке указываются начальная и конечная лексемы старого поддерева. На рис.

1 показан пример, когда в выражении “*a = b + c*” происходит замена “*b + c*” на вызов функции “*bar()*”.

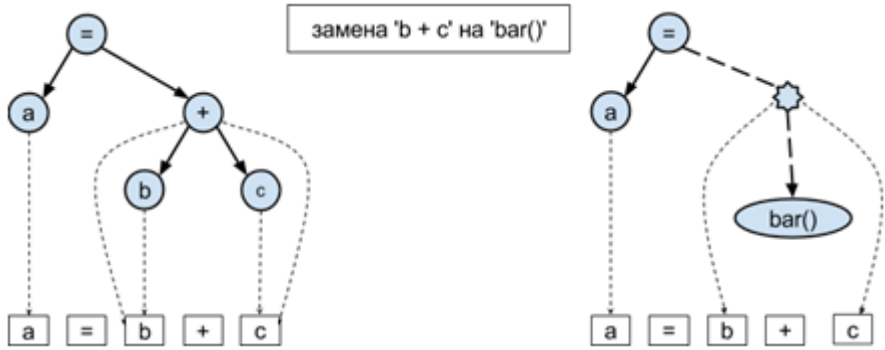


Рис. 1.

Ребро ведущее к узлу b + c помечается меткой L, а сам узел заменяется новым узлом, соответствующим вызову функции bar(). При этом последовательность лексем остается прежней, а в метку L лишь добавляется ссылки на первую и последнюю лексемы старого узла (“b + c”).

После того, как в дерево были внесены все изменения, достаточно совершить его обход и найти все метки. Очевидно, что если входящее в некоторое поддерево ребро уже помечено, то в этом поддереве не может быть других меток. Это значит, что изменения, описанные метками, не пересекаются, то есть их можно преобразовывать в изменения исходного кода в произвольном порядке (пересечений в исходном коде также не будет). Поэтому для каждой из меток:

- находится место вставки в исходный код (достаточно взять позиции из сохраненных для метки лексем)
- генерируется код для измененных частей поддерева

При генерации кода поддерева используется различный подход в зависимости от способа создания узла. Для узлов, созданных с помощью конструкторов, используется техника обхода и распечатки заранее заготовленных текстовых шаблонов для каждого конкретного типа узла. Для существующих узлов синтаксического дерева такой подход неприменим, поскольку не позволяет отобразить существующие узлы в том виде, в котором они присутствуют в исходном коде. Для них распечатывается последовательность лексем, заданная ссылками на начальную и конечную лексемы. В этой последовательности полностью сохранены изначальный стиль и комментарии, и это позволяет воспроизводить код для узлов рассматриваемого типа в неизменном виде.

Отдельно стоит упомянуть про сохранение макровызовов. Для того, чтобы информация о них присутствовала в синтаксическом дереве, существуют дополнительные ссылки от лексем, представляющих раскрытый макрос, на текстовый вид макровызова. Это дает возможность определить, какие лексемы были частью раскрытого макроса. Рассмотрим пример:

*#define ADD(x, y) x + y*

*a = ADD(b, c);*

Для данного кода будет построено синтаксическое дерево, изображённое на рис. 2.

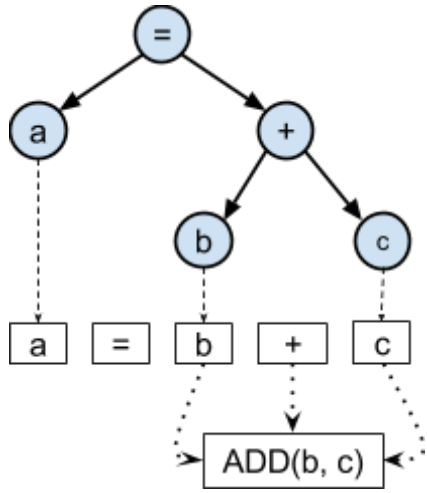


*Рис. 2.*

На рисунке также показаны лексемы, сопоставленные узлам дерева. Те из них, что находятся внутри раскрытого макровызова, имеют ссылку на его текстовое представление. Это позволяет сохранять изначальный вид макровызова при распечатке поддерева по последовательности соответствующих ему лексем.

## *7.* Ограничения *метода и пути их преодоления*

1. Данный метод применим для преобразования кода исключительно в пределах одной единицы компиляции (одного исходного файла; в случае языков C/C++ - с подключёнными заголовочными файлами). Данное ограничение является достаточно принципиальным и не может быть преодолено в рамках описываемого подхода. Тем не

менее, класс задач, решаемых с применением предлагаемого метода, остаётся весьма широким.

2. Потенциальная неэффективность. Рассмотрим **пример 3** более внимательно. При наличии у конструктора нескольких параметров обход его тела будет осуществляться по-новой для каждого параметра. Данное ограничение теоретически преодолимо, но в силу ограничения 1 и исходя из общих соображений о применимости предлагаемого подхода, едва ли стоит считать его критическим.
3. Недостаточная надёжность. По сравнению со встроенными в программный продукт Klocwork видами рефакторинга ([3]), в рассматриваемом подходе в принципе не гарантируется, что полученный в результате преобразования код будет семантически эквивалентен первоначальному или хотя бы будет компилироваться без ошибок. Тем не менее, предлагаемый метод является очевидно более точным, чем стандартные средства текстовых редакторов, и легко настраиваемым под потребности конкретных пользователей.
4. Преобразование кода на основе преобразования синтаксического дерева имеет, помимо очевидных преимуществ, и свои недостатки. Так, существуют фрагменты кода, для которых синтаксическое дерево не строится из-за условных директив препроцессора, и трансформация этих фрагментов рассматриваемым в данной статье способом невозможна.

## 8. Заключение

Описанный нами язык позволяет в сравнительно простом и понятном для пользователя виде специфицировать весьма сложные трансформации синтаксического дерева и выгодно отличается от существующих инструментов аналогичного назначения. Так, использование шаблонов языка KAST позволяет определять подлежащие замене конструкции максимально точно, с учётом синтаксического и семантического контекста, в котором эти конструкции находятся. Кроме того, предлагаемые функции модификации синтаксического дерева позволяют конструировать новые поддеревья, произвольным образом включая в их состав узлы, уже имеющиеся в исходном дереве.

Использование расширенного языка KAST для автоматического исправления дефектов представляет большой интерес не только в качестве исследовательского эксперимента. Подобная возможность была бы востребована в реальных программных продуктах, осуществляющих статический анализ кода. Планируется включить поддержку описанных в данной статье языковых расширений в состав продуктов компании Klocwork.

## Приложение

Реализуем на языке KAST примеры 2 и 4 из введения.

***Пример 2'.*** Заменить все непосредственные присваивания полям классов, осуществляемые не из методов этих классов, на функциональные вызовы.

```
// BinaryExpr [ @Op = KTC_OPCODE_ASSIGN ]
        [ Left::MemberExpr [ isVariable() ]
                [ isClassMember() ]
                [ $me_code := @Op ]
                [ $var_name := getName() ]
                [ $object := Expr::* ]
        ]
        [ $assignment := Right::* ]
        [ $func_name := concat('set_', $var_name) ]
        [ $new_func := new MemberExpr(Expr : $object,
                                Name : new Name (
@Id : $func_name ),
                                @Op : $me_code) ]
        [ substitute(this(), new CallExpr(Func : $new_func,
Args : { $assignment })) ]
```

***Пример 4'.*** Найти в коде на языке C определения всех функций, для которых не указан тип возвращаемого значения, и добавить в эти определения 'int' в качестве типа результата.

```
// FuncDef [ isCLanguage() ]
        [ not DeclSpecs[*]::AnyTypeOf ]
        [ not DeclSpecs[*]::ReservedTypeSpec ]
        [ not DeclSpecs[*]::ClassType ]
```

*[ not DeclSpecs[*]::EnumType ]*

*[ not DeclSpecs[*]::TypeName ]*

*[ not DeclSpecs[*]::AutoType ]*

*[ $declarator := Declarator::* ]*

*[ $body := FuncBody::* ]*

*[ $declspecs := add-element(getDeclarationSpecifiers(),*

*new BuiltinType(@Spec :*

*KTC_BUILTINTYPE_INT)) ]*

*[ substitute(this(), new FuncDef(DeclSpecs : $declspecs,*

*Declarator : $declarator,*

*FuncBody : $body)) ]*

### *Список литературы*


[1]. http://www.semdesigns.com/Products/DMS/DMSToolkit.html

[2]. http://strategoxt.org/Tools/WebHome

[3]. Н. Л. Луговской. Подход для проведения рефакторинга «Выделение функции» в инструменте Klocwork Insight. Сборник трудов Института системного программирования РАН. Под ред. акад. РАН Иванникова В. П. Т. 23. М., ИСП РАН, 2012. 476 с., с. 107-132.

[4]. С. В. Сыромятников. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST. Сборник трудов Института системного программирования РАН. Под ред. акад. РАН Иванникова В. П. Т. 21. М., ИСП РАН, 2011. 296 с., с. 51-68.

# Source code transformation and automatic correction of defects with KAST language

*N. L. Lugovskoy, S. V. Syromyatnikov*
*lugovskoy@ispras.ru, syrom@ispras.ru*
*ISP RAS, Moscow, Russia*

**Abstract**. This article is devoted to KAST pattern language extensions introduced for purposes of source code transformation. Currently KAST is used for matching syntactic patterns in syntactic trees built of C/C++, Java or C# sources. After introducing several new functions that can be called from a pattern, KAST patterns became applicable not only for detecting desired fragments of syntactic trees, but for specifying modifications of those fragments as well. Several existing approaches to code transformation are also considered. The main advantage of the described KAST based technique over those approaches is that our patterns can describe constructs we need to change together with the context these constructs are used in. Another important benefit is that KAST patterns can deal with semantic information about tree nodes. This significantly increases precision of pattern matching. The article also describes a method for converting modifications of syntactic trees into modifications of source code. The described technique allows to preserve comments, macro calls and indentation that were used in the initial code fragment. Though our approach is applicable only to a single compilation unit and doesn't guarantee syntactic correctness of newly generated code, it can greatly ease the task of automatic code transformation.

**Keywords**: refactoring; code transformation; static analysis; program architecture; pattern language; KAST

## References

[1]. http://www.semdesigns.com/Products/DMS/DMSToolkit.html
[2]. http://strategoxt.org/Tools/WebHome
[3]. N. L. Lugovskoj. Podkhod dlya provedeniya refaktoringa «Vydelenie funktsii» v instrumente Klocwork Insight [“Extract Function” Refactoring in Klocwork Insight Toolkit]. Trudy ISP RAN [The Proceedings of ISP RAS]. 2012, vol. 23, pp. 107-132 (in Russian).
[4]. S. V. Syromyatnikov. Deklarativnyj interfejs poiska defektov po sintaksicheskim derev'yam: yazyk KAST [Declarative Interface of Detecting Defects on Syntax Trees: KAST Language]. Trudy ISP RAN [The Proceedings of ISP RAS]. 2011, vol. 20, pp. 51-68 (in Russian).