УДК 681.3.068

АВТОМАТИЗАЦИЯ ПРОВЕРОК СЕМАНТИЧЕСКОЙ КОРРЕКТНОСТИ РАСПАРАЛЛЕЛИВАЮЩИХ ПРЕОБРАЗОВАНИЙ

З.Я. Нис
Ростовский государственный университет
Россия, 344090, Ростов-на-Дону, ул. Мильчакова, 8а
E-mail: irishrover@mail.ru

Ключевые слова: распараллеливающие компиляторы, статическая семантика, преобразования программ, корректность преобразований, автоматизация проверки корректности
Key words: parallelizing compilers, static semantics, program transformations, transformation correctness, automatizing correctness validation

В статье описывается способ автоматизации проверок статической семантической корректности распараллеливающих и оптимизирующих преобразований программ. Способ может быть полезен разработчикам компиляторов. Распараллеливающие и оптимизирующие преобразования программ обычно представляют собой высокоуровневые преобразования, т.е. осуществляются фактически над исходными текстами программ. При некоторых условиях преобразованные программы могут стать некорректными с точки зрения статической семантики языка. Автоматическое определение таких условий и «узких мест» в преобразовании, в которых потенциально может нарушиться статическая семантика, и является предметом обсуждения работы.
AUTOMATION OF THE SEMANTICS VALIDATION CHECKING OF PARALLELIZING TRANSFORMATIONS / Z.Ya. Nis (Rostov State University, Milchakova, 8a, Rostov-on-Don 344090, Russia, E-mail: irishorver@mail.ru). The paper describes a method for automation of the semantics validation checking of parallelizing program transformations. The method can be used by compiler developers. Parallelizing and optimizing program transformations are often represented as modifications of the source texts of corresponding programs. Under some conditions, these transformed programs may become incorrect in the sense of static semantics. Automatizing validation of these conditions and “potentially dangerous” places in the transformations where static semantics rules may be broken is the subject of this paper.

1. Введение

Большинство промышленных компиляторов (MS Visual C++, Intel C++, gcc [5, 11] и др.) содержат подсистемы оптимизации кода. Целью всякой оптимизации, проводимой компилятором, является улучшение тех или иных параметров генерируемого кода с обязательным сохранением логической семантики исходного кода, т.е. исходная и преобразованная программы должны решать одну и ту же задачу, хотя, возможно, и разными способами.

Современные компиляторы способны существенно ускорять выполнение генерируемых программ, лишь незначительно увеличивая их объем (оптимиза-

ция по скорости), либо же, наоборот, уменьшать объем порождаемого машинного кода практически без ущерба для производительности (оптимизация по размеру кода). Также возможна оптимизация по числу обращений к памяти, числу выполняемых операций и даже по энергопотреблению (что может быть актуально для различного рода мобильных устройств).

Другим видом оптимизации является возможность компилятора «распараллеливать» или «конвейеризовать» исходную программу, приводя ее к виду, удобному для выполнения на многопроцессорных системах – так называемое «повышение степени параллелизма», применяемое в распараллеливающих компиляторах и других системах преобразований программ.

Оптимизация может производиться, как на уровне исходных кодов программ, так и на уровне машинных кодов целевой платформы.

В качестве примеров типичных оптимизационных преобразований в компиляторах можно привести (см. также [3, 4]):

- размещение переменных в регистрах между использованиями,
- замену вызова подпрограммы подстановкой ее тела (inline expansion),
- удаление неиспользуемого («мертвого») кода, «лишних» выражений и переменных, невызываемых подпрограмм,
- вынесение инвариантов цикла из тела цикла,
- локальное и глобальное удаление общих подвыражений,
- замену умножений и делений сдвигами, сложениями или вычитаниями,
- перегруппировку операторов или машинных команд для оптимизации обращений к памяти,
- подстановку вперед и «протягивание» констант,
- оптимизация потока управления,
- замену рекурсии циклом и т.д.

Оптимизаторы, способные повышать степень параллелизма, могут проводить следующие преобразования (см. также [2-4]):

- расщепление циклов,
- разрыв итераций циклов,
- гнездование циклов,
- развертка циклов,
- раскрутка циклов,
- растягивание скаляров,
- вставки MPI- или OpenMP-кода и т.д.

В данной работе мы будем иметь дело с машинно-независимыми (высокоуровневыми) преобразованиями, которые по своей сути ближе к преобразованиям исходных текстов.

При проведении высокоуровневого преобразования необходимо, чтобы и исходная и результирующая программы удовлетворяли правилам статической семантики языка. (Далее будем называть свойство сохранения статической семантической корректности преобразованных данным преобразованием программ статической семантической корректностью этого преобразования.)

2. Автоматизация проверок корректности

Необходимость автоматизации проверок преобразований на статическую семантическую корректность возникла при работе над Открытой распараллели-

вающей системой (OPC, [1]) – проектом, разрабатываемом на кафедре Алгебры и дискретной математики РГУ под руководством д.т.н. Штейнберга Б.Я.

OPC предназначена для распараллеливания программ с процедурных языков программирования на параллельные компьютеры, ориентированные на математические вычисления. OPC включает в себя библиотеку из более чем двадцати высокоуровневых преобразований программ, поэтому автоматизация проверок корректности преобразований OPC очень актуальна.

Анализ показал, что даже в простых преобразованиях возможны неочевидные проблемы с семантикой. Рассмотрим преобразование «раскрутка цикла», заключающееся в замене цикла со счетчиком N-кратным повторением тела цикла, где N – число итераций в исходном цикле. Если тело преобразуемого цикла содержит операторы досрочного перехода к следующей итерации (оператор continue в языке C), то после применения преобразования эти операторы окажутся вне цикла, что является нарушением семантических ограничений в C ([6, §6.8.6.2]). В случае, когда преобразуемый цикл вложен в другой цикл, семантика нарушена не будет, но логика программы изменится. Если в теле цикла была метка, на которую происходил переход извне цикла, то статическая семантика программы после применения рассматриваемого преобразования также нарушится. Таким образом, разработчик данного преобразования должен учитывать особые случаи, возникающие, когда в теле преобразуемого цикла встречаются «опасные» для преобразования конструкции. Для преобразований, являющихся композициями других преобразований, «особых случаев» будет еще больше, а их обнаружение будет представлять проблему.

Проверку статической семантической корректности исходной программы осуществляет сам компилятор, выдавая в случае нарушений диагностические сообщения пользователю, который эти ошибки и должен исправить. Если же статическая семантика нарушена в автоматически преобразованной программе, то исправлять такие нарушения не представляется возможным – пользователь компилятора об этих ошибках знать не может и не должен, а компилятор вряд ли может исправить их самостоятельно. Именно поэтому может получиться, что преобразование, корректно работающее на 99% входных программ, будет генерировать ошибку времени выполнения или порождать некорректный с точки зрения статической семантики код в 1% случаев. Разумеется, такое преобразование нельзя использовать в компиляторе, так как заранее неизвестно, что именно будет подано на вход пользователем такого компилятора, а, значит, нельзя и избежать возможных ошибок.

Итак, становится ясным, что существует потребность в методах и средствах автоматизации проверок преобразований на предмет их статической семантической корректности. В настоящей статье делается попытка описать такие методы автоматизации. Для достижения этой цели предлагается решить следующие задачи:

- выбрать удобный для осуществления преобразований способ представления программ;
- разработать специальный язык описания преобразований программ в таком представлении, который был бы достаточно простым для пользователя и в то же время емким с выразительной точки зрения, позволяя описывать большинство из используемых на практике высокоуровневых преобразований;
- разработать теоретические методы и программные средства, позволяющие априори, до применения преобразования, записанного на этом специальном

языке, проверять статическую семантическую корректность, выдавая диагностические сообщения разработчику в случае обнаружения «проблемных мест»;

- разработать инструментарий для автоматического перевода (трансляции) преобразования, записанного на указанном специальном языке, в классы языка C++.

Для решения поставленных задач автором применялось несколько методов исследования. На начальном этапе делались попытки представлять программы и преобразования в виде λ-функций или в операциональном виде. В конечном итоге был выбран путь использования абстрактных синтаксических деревьев (AST, [7]) для представления программ; преобразования в таком случае естественным образом представляются в виде операций над AST.

Для задания статических семантических правил также было опробовано несколько подходов. От задания статических семантических ограничений как дуг специального вида на дереве синтаксического разбора ([9]) до подхода, связанного с заданием достаточно абстрактных ограничений в виде логических выводов ([12]).

Сами формулировки задач также претерпевали изменения – от контроля преобразований во время их выполнения до текущего, априорного подхода, который, очевидно, более предпочтителен для разработчиков преобразований.

3. Представление программ в виде AST

3.1. Абстрактные синтаксические деревья

Как известно, компиляторы проводят преобразования над программами, представленными не в виде своих исходных текстов, а в некотором внутреннем представлении, например в виде абстрактных синтаксических деревьев – AST (от англ. Abstract Syntax Tree) [7]. После построения AST компилятор может проводить над ним ряд преобразований, в т.ч. оптимизирующих, а затем по уже преобразованному AST генерировать код для целевой архитектуры или «передавать» полученное AST следующей фазе компиляции для дальнейших манипуляций и оптимизаций – теперь уже на более низком, машинно-зависимом уровне.

AST строится компилятором на стадии синтаксического анализа входной программы одновременно с разбором. Абстрактное синтаксическое дерево представляет собой дерево синтаксического разбора программы, поэтому AST довольно близко к исходному тексту программы. С другой стороны, AST имеет «сжатый» вид, удобный для представления языковых конструкций. «Сжатость» означает то, что из AST исключаются «знаки препинания» языка – точки, запятые, разделители и иные объекты, не несущие семантической информации. Листья AST соответствуют терминалам, а остальные узлы – нетерминалам грамматики языка. Корень AST соответствует стартовому символу грамматики. Дуги AST – это правила вывода грамматики. Если символ в грамматике A следует в некотором правиле лексически раньше символа B, то в AST он будет расположен левее, чем B. Таким образом, порядок расположения потомков нетерминала является фиксированным, определяясь грамматикой.

В синтаксическом дереве операции и ключевые слова не представлены в качестве листьев, а связываются с внутренним узлом, который выступает в дереве разбора родительским для этих листьев.

На практике листья и узлы AST помечаются атрибутами, которые характеризуют их тип. Например, для листа «идентификатор» атрибутами могут являться имя идентификатора и ссылка на вершину дерева AST, в которой идентификатор описан.

Удобство использования AST заключается в том, что это дерево отражает общую синтаксическую структуру разобранной программы. Поэтому, например, условный оператор представляется как родительская вершина с тремя непосредственными потомками – условие, ветка then и ветка else, а потомки блочного оператора (в языке C – это оператор «фигурные скобки») – операторы, непосредственно вложенные в него. Также, благодаря помеченным атрибутами вершинам, AST отражает еще и семантическую структуру программы.

По атрибутированному AST и грамматике языка можно «восстановить» исходную программу с точностью до пробельных символов. Поэтому можно говорить о программе (точнее о классе эквивалентных программ), соответствующей данному AST.

3.2. AST программ на языке C

Для представления преобразуемых программ будем использовать формализм атрибутированных AST для языка C в стандарте от 1999 года [6]. Этот выбор обусловлен тем, что язык является промышленным стандартом для многих реальных приложений, в частности популярные компиляторы Intel C++ и gcc ([11]) написаны именно на «чистом» C. OPC, для которой система автоматизации проверок преобразований изначально разрабатывалась, также содержит парсер языка C. Язык C, согласно стандарту, имеет достаточно обширную математическую библиотеку, что позволяет использовать его для эффективной реализации многих вычислительных алгоритмов. И, наконец, язык C хорошо переводится в машинные коды для многих практически важных целевых архитектур.

При построении AST для программ на C мы будем считать, что программы не содержат директив условной компиляции и макросов. Это не очень ограничивающее требование, если вспомнить, что на вход реальному компилятору подается программа, в которой директивы и макросы уже предварительно «развернуты» препроцессором (кроме директив #pragma и #line, которые согласно стандарту языка можно игнорировать; см. [6, §6.10.6]).

Также будем считать, что программы не содержат множественных присваиваний (т.е. присваиваний вида: $x = y = z$) и не используют оператора «запятая», а операторы if, while, switch и т.д. в своих условиях не содержат операторов присваивания или иных операторов с побочным эффектом. Эти ограничения преследуют цель упрощения языка преобразований.

В любой программе на языке C (а также в Паскале, Алголе, Фортране и во многих других процедурных языках) можно выделить три основные составляющие:

- операторы (пустой, цикла, условный, присваивания и т.д.);
- выражения (обращения к переменной, элементу массива, константе, вызов функции, унарные, бинарные, тернарные выражения и т.д.);
- объявления (переменных, констант, пользовательских типов и т.д.).

Этот факт мы отразим в структуре используемого нами AST, а сами составляющие будем называть видами узлов AST.

Будем считать, что одно синтаксическое дерево относится только к одной единице трансляции, т.е. к одному С-файлу. Это сделано только для удобства, чтобы избежать несущественных для рассматриваемой задачи проблем со спецификаторами extern, static и другими известными в С проблемами многофайловой компоновки.

Итак, корнем AST является «программа». Все дочерние поддеревья строятся согласно правилам грамматики языка С. От корня исходят дуги к глобальным объектам, таким как объявления глобальных переменных, пользовательских типов данных, функций ([6, §6.2.1]). У функций в свою очередь могут быть дочерние узлы, объявляющие локальные переменные. Одна из дочерних дуг функции обязательно ведет на блочный оператор, образующий тело функции. От блочного оператора могут исходить дуги на операторы, вложенные в данный блочный. Операторы могут иметь дуги-потомки, которые ведут к выражениям-операндам. Листьями дерева программы являются константы, обращения к скалярным переменным.

4. Преобразования как трансформации AST

Формализм AST удобно использовать для представления преобразований программ. В этом случае процесс преобразования программы имеет вид:

- нахождение поддерева AST программы, удовлетворяющего некоторому условию (условие применимости преобразования);
- замена его на другое поддерево, либо изменение его атрибутов.

Более того, так как программы можно представлять в виде деревьев, то преобразования можно представлять как операции над деревьями (вставить, удалить, изменить поддерево или его атрибуты и т.д.). Похожий подход используется в системах, построенных по принципу переписывания термов (term rewriting) [10].

Таким образом, мы приходим к идее языка преобразований, оперирующего поддеревьями AST. Список операций такого языка не очень велик, в частности потому, что не все операции, имеющие смысл для поддеревьев, имеют смысл для программ. Например, вставка четвертого потомка к узлу IF (к имеющимся «условие», “then” и “else”) явно лишена смысла и приведет к заведомо синтаксически некорректной программе.

Основные операции для AST для операторов: вставка, удаление (для управляющих операторов с заголовком еще и удаление или модификация заголовка), перенос, замена пустым оператором. Для выражений – это вставка, удаление, перенос, замена операции (например, замена сложения на вычитание). Для объявлений – вставка, удаление, изменение модификаторов (константный, статический, внешний, автоматический, регистровый и т.д.).

Для всех видов сущностей язык описания преобразований должен позволять изменение их атрибутов. Также язык должен позволять последовательное применение преобразований, применение преобразований в зависимости от выполнения условий и циклическое применение преобразований.

5. Формализация статической семантики

Необходимым условием автоматизации обнаружения ошибок статической семантики является формализация задания самой статической семантики. В настоящее время в литературе не предложено единого подхода – конкретный способ выбирается в зависимости от преследуемых целей.

5.1. Методы описания статической семантики

В работе [9], посвященной автоматической генерации семантических тестов для компиляторов, показывается, что статические семантические связи можно формально задать как отношения (связи) между объектами синтаксических деревьев и их атрибутами. Например, правило статической семантики «каждая переменная перед использованием должна быть объявлена» требует наличия связи «многие-к-одному» между использованиями переменной (вид – «выражение») и точкой ее объявления (вид – «объявление»). Если такую связь удается установить, то использование переменной является корректным с точки зрения статической семантики. Если нет, то должна быть сгенерирована ошибка, ассоциированная с нарушенным правилом.

Другой пример. В правиле «оператор continue может встречаться только внутри тела цикла» оператор continue должен иметь в качестве одного из своих родительских узлов в AST оператор цикла. Если такой цикл найден, то использование оператора continue правомерно, иначе – нет.

Правила могут быть не только позитивными («связь должна существовать»), но и негативными («связь не должна существовать»). Например, рассмотрим правило «в одной области видимости (scope) не может быть двух и более идентификаторов с одинаковым именем». Если существует связь, объединяющая несколько идентификаторов с одним именем в одной области видимости, то это говорит о нарушении статической семантики.

В работе описаны и другие виды задания связей между синтаксически объектами с целью полного определения статической семантики. Мощности предложенного подхода, по словам автора, достаточно для выражения семантики таких языков как C, Java и C#, даже с учетом объектной ориентированности двух последних языков.

Другой способ описания статической семантики (применительно к языку Паскаль) указан в работе [13]. Способ использует алгебраический подход, описывая ограничения языка с помощью набора особых семантических функций и уравнений специального вида, используемых системой переписывания термов.

Способ описания статической семантики языка, основанный на использовании логических спецификаций, предложен в работе [12]. Согласно этому способу все правила статической семантики языка сводятся к небольшому набору аксиом и двум видам логических правил – определениям и ограничениям. Каждая конкретная программа с точки зрения метода представляет собой совокупность исходных фактов, касающихся ее статической семантики. Правила-определения являются связующим звеном между зависящими от синтаксиса языка «фактами программы» и небольшим количеством абстрактных правил-ограничений, которые могут не зависеть от конкретного языка.

Из исходных фактов программы с помощью правил-определений выводятся новые факты. Если вновь полученные факты не противоречат выполнению правил-ограничений, то программа признается корректной с точки зрения статической семантики. Если какое-то из правил не выполняется, то это означает ошиб-

ку в статической семантике программы. Причем тип ошибки определяется самим невыполнившимся правилом-ограничением.

Предлагаемый автором работы [12] подход позволяет заметно ограничить количество проверяемых правил-ограничений за счет увеличения сложности их разработки. Дополнительно при использовании этого подхода необходимо ввести и определить некоторые вспомогательные предикаты, например предикат «*объявлен*(*имя _ идентификтаора*,*блок*)», принимающий истинное значение, если, идентификатор с данным именем определен в данном блоке.

Одно из удобств использования именно этого подхода состоит в том, что для разных языков часть правил совпадает (типичный пример – правило «использования только объявленных идентификаторов», которое встречается практически во всех алголоподобных языках), а, следовательно, их можно использовать повторно – изменятся лишь правила-определения.

В настоящей статье мы будем использовать некоторую модификацию последнего предложенного подхода, связанного с применением логических правил.

5.2. Использование логических схем для описания статической семантики

Для описания статической семантики предлагается использовать логические правила и предикаты.

С каждым грамматическим символом в AST разобранной программы – терминалом и нетерминалом – свяжем некоторое число атрибутов, описывающих характеристические свойства символа. Например, для идентификатора атрибутами могут быть строка, представляющая его имя, и ссылка на поддерево AST, представляющее тип идентификатора. В то же время для оператора “if” атрибутами не могут являться ссылки на поддеревья, соответствующие условию оператора, веткам “then” и “else” – они являются дочерними поддеревьями. Т.о., можно считать, что атрибуты – это свойства грамматических символов, которые не следуют из структуры дерева AST.

Логические правила имеют общий вид: $A:\varphi \to \phi$, где φ и ϕ – предикаты, зависящие от терминалов и нетерминалов грамматики или их атрибутов. Такая запись правил подразумевает следующее: если предикат φ истинен для некоторого набора аргументов, то предикат ϕ истинен для своего набора аргументов.

Каждое правило имеет область применимости – множество объектов, к которым оно может применяться. Область применимости правила – это подмножество декартового произведения допустимых значений аргументов предиката φ, на котором он истинен. Типами аргументов, определяющими эти допустимые значения, являются классы грамматических символов. Т.е. символы “if” и “while” принадлежат разным классам.

Предикаты составляются из термов, соединенных между собой знаками логических операций – конъюнкции, дизъюнкции и отрицания, возможно заключенных в скобки для изменения приоритета вычислений, а также кванторами общности и существования. Термы состоят из констант, имен атрибутов и функций, соединенных знаками сравнения (равенства, неравенства и т.д.).

Пример предиката: унарный предикат, истинный тогда и только тогда, когда его аргумент-идентификатор объявлен где-либо в программе.

Функции – это вспомогательные отображения, цель использования которых – исключение повторного написания одинакового кода. По сути, функции в

данном контексте – это аналоги предикатов, которые вычисляются не в булевское значение, – «истина» или «ложь» – а в значение произвольного типа. В качестве примера функции можно привести функцию, «возвращающую» для данного оператора блочный оператор, в который он вложен.

5.3. Задание семантики

Разработка правил для представления статической семантики производится в два этапа: на первом этапе задаются правила-определения, а на втором – правила-ограничения. Строгой фиксированной границы между двумя типа правил нет. Эта граница определяется самим разработчиком, но с учетом того, что правила-ограничения должны максимально абстрагироваться от конкретного языка.

Правила-определения должны оперировать атрибутами грамматических символов, порождать факты, «делая истинными» предикаты, описывающие менее связанные с синтаксисом понятия языка.

Например, правило-определение может вводить в рассмотрение предикат «*идентификатор _ объявлен _ в _ блоке*$(\ldots)$» при обнаружении в дереве разбора соответствующей синтаксической конструкции.

Правила-ограничения являются основным средством описания статической семантики, а правила-определения – это только способ сделать семантические ограничения в меньшей степени зависящими от конкретного языка. Правила-ограничения должны оперировать «высокоуровневыми» предикатами, полученными после применения правил-определений. Примером правила-ограничения может служить многократно уже упоминавшееся правило «используемый идентификатор должен быть объявлен», характерное для большинства процедурных языков, в частности для Паскаля и С. На языке предикатов это можно записать так:

$$\forall id \in ID \quad \textit{использован}(id) \rightarrow \textit{объявлен}(id).$$

Если для некоторого идентификатора в процессе порождения предикатов правилами-определениями предикат «объявлен» не будет определен как истинный, то данное правило-ограничение будет нарушено, что будет означать ошибку в статической семантике программы. Причем диагностика такой ошибки не вызывает затруднений – необходимо указать правило и аргументы «ложного» предиката.

6. Обнаружение нарушаемых правил

Если статическая семантика задана в виде логических правил, то для каждой элементарной операции над каждой сущностью каждого вида дерева программы (определение, выражение, оператор) можно определить, какие правила могут нарушаться после применения данной операции.

Например, операция удаления описания идентификатора может привести к тому, что его использование нарушит правило о необходимости предварительного описания идентификаторов.

Путем анализа статических семантик в нескольких процедурных языках были выбраны типовые правила, которые могут встречаться при задании семантических ограничений в виде логических условий, и их реализации в реальных языках программирования. Считаем, что $A \in N_A$, $B \in N_B$, где N_A и N_B – мно-

жества грамматических символов грамматики, в каждом конкретном правиле – свои.

a) $\forall A \exists B \mid f(A,B) = true$ (для любого оператора continue должен существовать цикл, которому он принадлежит);
b) $(\forall A \forall B \mid f_1(A,B) = true) \Rightarrow f_2(A,B) = true$ (имена идентификаторов, объявленных в одной области видимости, должны различаться);
c) $\forall A \Rightarrow f(A) = true$ (выражение в условии оператора if должно быть приводимо к логическому типу);
d) $\exists A \mid f(A) = true$ (в операторе switch должна быть как минимум одна ветка не default);
e) $\exists A \mid \forall B \Rightarrow f(A,B) = true$ (оператор return должен возвращать выражения типа, совпадающего с типом функции).

Для каждого правила определим, какие элементарные преобразования могут привести к его нарушению. Для сокращения числа затрагиваемых одним элементарным преобразованием правил необходимо учитывать, что правила обычно относятся только к нескольким заранее известным видам символов, поэтому проверять «чужие» символы нет необходимости. Считая, что программа изначально корректна с точки зрения статической семантики, получаем:

a) условие будет нарушено, если требуемый символ B не существует. Т.к. изначально программа была корректной, то это означает, что символ B был удален или были изменены его атрибуты, которые используются для вычисления предиката f. Следовательно, для данного типа правил потенциально опасными с точки зрения статической семантики будут элементарные операции удаления и изменения атрибутов, которые применяются к символам типа N_B;
b) условие будет нарушено, если найдется хотя бы один символ A или символ B такой, что второй предикат вычисляется в ложь, а первый остается истинным. С учетом изначальной корректности программы, это означает, что была выполнена элементарная операция вставки или копирования нового символа, либо были изменены атрибуты уже существующего символа;
c) см. предыдущий пункт;
d) условие нарушится, если предикат будет вычисляться в ложь для всех нетерминалов данного вида. Такая ситуация говорит о том, что к искомому символу A была применена элементарная операция удаления или изменения атрибутов, после чего символ перестал удовлетворять условию предиката.
e) возможны два случая: символ A был удален или были изменены его атрибуты, либо появился символ B (копированием, вставкой или изменением атрибутов) такой, что для него и A предикат не вычисляется в истину.

Для каждого предиката, входящего в правило, можно выписать те атрибуты, которые он использует для своего вычисления. В этом случае можно с большей точностью определять, изменения каких именно атрибутов приводят к изменению значения предиката.

Таблица 1 в очень сжатой форме отражает зависимости между элементарными операциями и теми правилами статической семантики, которые могут нарушиться в результате их применения.

Таблица 1. Правила статической семантики языка, которые потенциально могут быть нарушены при применении основных элементарных операций.

Элементарная операция	Нарушаемые правила
Вставка	может нарушиться единственность
Удаление	может нарушиться существование
Копирование	может нарушиться единственность
Изменение атрибутов	• может нарушиться единственность • может нарушиться существование

На основе приведенных рассуждений построена рассматриваемая в статье система ASTOL для проверки семантической корректности преобразований.

7. Система ASTOL

Система ASTOL предназначена для решения сразу нескольких задач:

- унификация разработки преобразований программ за счет использования единого языка описания преобразований;
- унификация описания статической семантики процедурных языков с помощью логических правил и предикатов;
- автоматизация проверок статической семантической корректности преобразований.

Для автоматизации проверок преобразований на семантическую корректность предлагается следующая схема:

1) Все программы представляются в виде соответствующих абстрактных синтаксических деревьев AST с атрибутами. Построение AST по исходной (подлежащей преобразованию) программе осуществляет парсер с языка C, входящий в состав системы ASTOL. AST представляется в виде связанных взаимными ссылками объектов специальных классов, представляющих собой различные синтаксические конструкции языка C.
2) Преобразования записываются в виде программ на специализированном языке описания преобразований ASTOL (AST Operating Language), содержащем конструкции для манипулирования деревьями AST (элементарные преобразования) и атрибутами вершин.
3) Ограничения на статическую семантику языка записываются на специализированном интерпретируемом языке, основанном на формализме логических правил.
4) Из полученной программы-преобразования компилятор преобразований ASTOL порождает класс языка C++, который собственно и реализует преобразование AST. Интерфейс класса является унифицированным для всех преобразований. Методы класса используют библиотеку поддержки, поставляемую совместно с компилятором ASTOL. В процессе работы компилятор ASTOL выдает диагностические сообщения, указывающие разработчику преобразования, какие потенциально опасные места с точки зрения статической семантики содержит его преобразование, а также предлагает способы их исключения и по возможности генерирует тестовые программы, на которых могут нарушиться правила статической семантики при проведении данного преобразования.
5) После устранения «опасных» мест из программы-преобразования, разработчик может использовать сгенерированный компилятором ASTOL класс при

разработке своего компилятора (или иной системы преобразований программ) для реализации безопасных с точки зрения статической семантики преобразований.

8. Программная реализация системы ASTOL

В настоящее время система ASTOL находится в стадии реализации. Основным средством разработки является среда MS Visual Studio 7.1, язык C++. Для реализации парсеров используется генератор компиляторов ANTLR [15].

С программной точки зрения система ASTOL представляет собой ядро, которое состоит из библиотеки классов, предназначенных для построения и манипулирования AST, а также из библиотеки, включающей методы проверки корректности статической семантики преобразований, описываемые в настоящей работе. Также система включает в себя функции работы с логическими предикатами. Важной частью системы ASTOL является система порождения классов языка C++, представляющих собой пользовательское преобразование, а также библиотека поддержки времени выполнения для этих классов-преобразований.

9. Заключение

В статье описывается метод автоматизации проверок статической семантической корректности распараллеливающих преобразований. Метод основан на представлении правил статической семантики языка в виде логических спецификаций и последующей проверке выводимых фактов, а также на представлении преобразований в унифицированном виде за счет использования языка описания преобразований. Автоматизация проверок статической семантической корректности преобразований может быть полезна разработчикам распараллеливающих и оптимизирующих компиляторов.

Список литературы

1. Открытая распараллеливающая система. http://ops.rsu.ru.
2. Касперски К. Техника оптимизации программ. Эффективное использование памяти. С-Пб.: БХВ-Петербург, 2003. 464 с.
3. Касьянов В.Н. Оптимизирующие преобразования программ. М.: Наука, 1988. 336 с.
4. Штейнберг Б.Я. Математические методы распараллеливания рекуррентных циклов для суперкомпьютеров с параллельной памятью. Ростов-на-Дону: Изд-во Рост. Ун-та, 2004. 192 с.
5. Практикум «Оптимизирующие компиляторы». Нижегородский государственный университет. Препринт, 2004.
6. ISO/IEC 9899:1999 (E), Programming languages – C.
7. Ахо А. Ульман Дж. Теория Синтаксического анализа, перевода и компиляции. М.: Мир, 1978. Т. 2.
8. Штейнберг Б.Я., Напрасникова М.В., Нис З.Я. Тестирование преобразований Открытой распараллеливающей системы // Искусственный интеллект. Научно-теоретический журнал. Украина, Донецк: Институт проблем искусственного интеллекта НАНУ, 2004.
9. Архипова М.В. Конструктивное описание правил статической семантики языков программирования // Вторая Всероссийская научная конференция «Методы и средства обработки информации». М.: МГУ, 2005. С. 323-329.
10. Baader, T. Nipkow. Term Rewriting and All That. Cambridge University Press, 1998. 301 p.

11. GCC Internals. http://www.gcc.gnu.org.
12. Ильичева О.А. Интерпретатор логических спецификация с диагностикой ошибок. Минск: АН Беларуси, Вычислительный центр, 1991. 15 с.
13. van Deursen A. An Algebraic Specification For The Static Semantics Of Pascal. CWI, 1997.
14. F. L. Morris. Correctness of Translation of Programming Languages, an algebraic approach. AIM 174, 1972.
15. Генератор компиляторов ANTLR. http://antlr.org.