Министерство образования и науки Российской Федерации

Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Труды научной школы И. Б. Симоненко

Выпуск второй

Ростов-на-Дону
Издательство Южного федерального университета
2015

УДК 517.9; 519.6
ББК 22.161.6; 22.19
Т78

Печатается по решению редакционной комиссии по математическим наукам
Института математики, механики и компьютерных наук им. И. И. Воровича
(протокол № 7 от 6 ноября 2015 г.)

Т78 Труды научной школы И. Б. Симоненко. Выпуск второй / под ред. М. Э. Абрамяна, Я. М. Ерусалимского, В. С. Пилиди, Б. Я. Штейнберга ; Южный федеральный университет. – Ростов-на-Дону : Издательство Южного федерального университета, 2015. – 304 с.
ISBN 978-5-9275-1607-0

В юбилейном сборнике, посвященном 80-летию известного российского математика И. Б. Симоненко, представлены работы по теории операторов, математической физике, асимптотическим методам, методам программирования, теории кодирования, распараллеливанию программ, алгоритмам на графах. Рекомендуется научным работникам, преподавателям вузов, аспирантам.

Статьи публикуются в авторской редакции.

ISBN 978-5-9275-1607-0

УДК 517.9; 519.6
ББК 22.161.6; 22.19

АНАЛИЗ ПОДДЕРЖКИ ОБОБЩЁННОГО ПРОГРАММИРОВАНИЯ В НОВОМ ПРОЕКТЕ КОНЦЕПТОВ ДЛЯ C++

Пеленицын А. М.
Южный федеральный университет, Ростов-на-Дону

We test current concept proposal for C++ programming language against a set of parameters suggested in seminal paper by Garcia *et al.* (2003) and extended by Oliveira *et al.* (2008). We point out how current rather moderate state of proposal follows from previous one, which proved to be overcomplicated and fall to be merged into C++11.

Введение

Неограниченный параметрический полиморфизм, как он представлен в механизме шаблонов C++98, имеет ряд известных недостатков, главным из которых является генерация длинных сообщений об ошибках. Для устранения этих недостатков на протяжении 2000-х и 2010-х рассматриваются способы введения ограниченного параметрического полиморфизма в язык C++. В статье исследуется наиболее актуальный из таких способов — описание и использование ограничений на параметры шаблонов C++, так называемые *концепты C++1z* (также: Concepts Lite). Данный проект представляет собой расширение языка, планируемое к включению в стандарт языка 2017 года.

В данной работе проект концептов C++1z анализируется с точки зрения критериев поддержки обобщённого программирования. Результаты проведённого анализа отражены в расширении столбцом C++1z сравнительной таблицы средств обобщённого программирования в различных языках программирования (см. таблицу 1). В статье также проясняется связь текущего проекта концептов с предшествующей версией для стандарта 2011 года.

Впервые таблица, подобная приведённой на с. 256, была составлена, насколько известно, в работе [1] и с тех пор расширялась. Важным этапом в модификации этой таблицы стала работа [2], которая добавляла в неё языки Scala и C++0x, а также нижние четыре строки. Отметим также, что в таблице 1 по сравнению с упомянутыми статьями опущены несколько колонок, которые соответствуют языкам, не затрагиваемым в нашей работе. Однако все эти языки являются более слабыми (имеют меньше значков ●), чем два лидера: Haskell и Scala.

Таблица 1. Поддержка обобщённого программирования в различных языках

	C++	C#	Haskell	Scala	C++0x	C++1z
Мультипараметрические концепты	–[9]	●	●	●	●	●
Ассоциированные типы	●	◐	●	●	●	●
Ограничения на ассоц. типы	–	◐	●	●	●	●
Ретроактивное моделирование	–	◐	●	●	●	○
Псевдонимы типов	●	○	●	●	●	●
Раздельная трансляция	○	●	●	●	○	○
Вывод неявных аргументов	●	◐	●	●	●	●
Модулярный контроль типов	○	●	●	●	◐	○
Области видимости моделей концептов	○	○	○	●	○	○
Перегрузка на основе концептов	●	○	○	◐	●	●
Ограничения в виде равенства типов	–	○	●	●	●	●

Обозначение C++1z используется в ряде работ для обозначения стандарта языка C++, который запланирован к выходу в 2017 году. Стремление комитета по стандартизации языка к наилучшему результату уже приводила к сдвигу сроков: так, стандарт, который должен был выйти в конце нулевых годов, долгое время обозначался C++0x (появились даже соответствующие ключи компиляторов), но в результате официально утверждённый документ появился лишь в 2011 году. Версия стандарта 2014 года, которая до утверждения называлась C++1y, появилась вовремя, но, как и планировалось, содержала лишь небольшие изменения и дополнения. Выдержит ли сроки редакция C++1z, пока неизвестно.

В анализе проекта концептов C++1z использовался ряд документов комитета по стандартизации C++: некоторые примеры взяты из неформального введения в тему [3]; детали синтаксиса уточнены по более свежему формальному документу-описанию изменений в текст стандарта языка [4]; изменения в стандартной библиотеке пока ещё не пересмотрены в свете последней редакции концептов, но некоторое представление о них можно получить по отчёту [5]. Мнение автора языка C++, Бьярне Страуструпа приводится в основном по статье [6].

Статья организована следующим образом. В первом разделе приводится пример, мотивирующий к рассмотрению механизмов ограниченного параметрического полиморфизма в C++. Во втором разделе отме-

[9]Знак ‘—’ в таблице означает, что возможность непосредственно не поддерживается на уровне языка, но может быть смоделирована комбинацией некоторых средств языка.

Листинг 1 Функция сортировки контейнера: неограниченный полиморфизм

```
template<typename Cont>
void sort(Cont & c);
```

чены требования, которые предъявлялись к первому проекту концептов в языке — так называемым концептам C++0x, — а также причины того, что часть этих требований выполнить не удалось. В третьем разделе объясняется отсутствие поддержки двух важных критериев обобщённого программирования из таблицы 1, модулярного контроля типов и полноценного ретроактивного моделирования, в новом проекте концептов как следствие проблем, выделенных в разделе 2. Четвёртый раздел посвящён тому, как в новом проекте концептов реализуется ещё один важный критерий таблицы 1 — перегрузка на основе концепт-ограничений. В пятом разделе обсуждаются оставшиеся критерии обобщённого программирования из таблицы 1.

1 Пример и мотивация

Одна из главных проблем шаблонов C++98 состоит в длинных текстах сообщений об ошибках компиляции. Происходят они от того, что несоответствие типа неявным требованиям выясняется посреди процесса подстановки этого типа в определение шаблона. В результате текст ошибки ссылается на детали реализации шаблона, которым не смог удовлетворить тип-аргумент. Однако клиент шаблона, как правило, ничего не знает о его реализации.

Типичный пример такого несоответствия можно составить с помощью распространённой ошибки, которая состоит в попытке отсортировать список общим алгоритмом. В стандартной библиотеке C++ общий алгоритм сортировки подразумевает произвольный доступ к элементам сортируемого контейнера, в то время как тип списка такой доступ не предоставляет. Приведём пример заголовка шаблона функции сортировки, который несколько проще стандартного алгоритма и принимает не пару итераторов, а контейнер целиком — в трёх вариантах (листинги 1 и 2) В своей реализации такой шаблон может, к примеру, вызывать стандартный алгоритм `std::sort`.

Как видно, в листинге 1 ничто не указывает на предположения о типе-параметре кроме его имени — типичная ситуация для неограниченного

Листинг 2 Функция сортировки контейнера: ограниченный полиморфизм, полная и сокращённая версии

```
template<typename Cont>
    requires Sortable<Cont>()
void sort(Cont & c);

template<Sortable Cont>
void sort(Cont & c);
```

полиморфизма. В листинге 2 приведены две эквивалентных версии в синтаксисе концептов C++17, причём вторая, сокращённая, совпадает также и с вариантом, предлагавшимся для концептов C++0x. В ней видно использование явного ограничения `Sortable` — это и есть именованный набор требований к шаблонному параметру `Cont`, который называется *ко́нцептом*. В данном случае набор требований должен обеспечивать произвольный доступ к элементам контейнера и наличие операции «меньше» для этих элементов.

Наиболее важным свойством концептов является то, что проверка соответствия аргумента заданным требованиям проводится до того, как начинается подстановка этого аргумента в тело шаблона. Таким образом, при возникновении несоответствий компилятор сообщит о них незамедлительно и в терминах ограничений, представленных в *интерфейсе* обобщённой компоненты, а не её реализации. Имеющиеся прототипы реализаций концептов C++0x и C++1z дают существенно более доступные сообщения об ошибках в подобных случаях, чем это было с неограниченными шаблонами C++98.

В рамках короткого примера здесь не будем дано определение достаточно обширного концепта `Sortable`. Важным здесь является то, что концепты это новая сущность языка программирования, не укладывающаяся в систему типов C++98. Это отличает механизм ограниченного параметрического полиморфизма, предлагаемый для C++, от ограниченного параметрического полиморфизма, основанного на подтипировании и характерного для чистых объектно-ориентированных языков, таких как Java и C# — их подход является более слабым с точки зрения таблицы 1 (с. 256) и неоднократно критиковался с предложением улучшений [7–9].

2 Концепты C++0x и их проблемы

Обсуждение концептов C++1z следует начать с некоторого анализа целей и причин неудачи концептов C++0x, которые также отражены в таблице 1 (с. 256). Кратко перечислим основные элементы проекта концептов C++0x, которые одновременно являются достаточно общими требованиями к механизму ограниченного параметрического полиморфизма. Они понадобятся и для обсуждения концептов C++1z.

1. Синтаксис для определения именованного набора ограничений — концепта.
2. Синтаксис для наложения ограничения (концепта) при объявлении шаблона класса или функции.
3. Способ проверки соответствия использования шаблонных параметров внутри шаблона класса или функции наложенным ограничениям.
4. Способ указания соответствия конкретных типов тем или иным концептам — так называемые *модели концептов* (concept map).

Оказалось, что основные сложности доставляли пункты 3 и 4. Первый из них требовал существенной модификации алгоритма поиска имён при обработке шаблонов компилятором. Такая модификация была выполнена в рамках проекта ConceptGCC [10], работала она весьма медленно и нестабильно. Мнение разработчиков этой версии состоит в том, что при необходимой доработке, проблем с производительностью компилятора быть не должно. Однако сам масштаб изменений, которые требовалось внести в распространённые компиляторы в отношении обработки шаблонов, негативно воспринимался членами комитета по стандартизации.

Второй проблемный пункт (4-й в списке выше) с моделями концептов неоднозначно воспринимался даже основным идеологом концептов C++0x, автором языка C++ Бьярне Страуструпом. Дело в том, что свидетельства соответствия конкретных типов некоторым концептов во многих случаях тривиальны и могут быть автоматически установлены компилятором. Однако такой автоматизм может плохо сочетаться с другими возможностями обновлённых шаблонов, как например, перегрузкой на основе концептов. Более того, в комитете по стандартизации высказывались мнения, что получение моделей концептов без участия, а значит, без явного намерения программиста просто неверно идеологически.

В целом, по состоянию на 2009 год проект концептов C++0x оценивался Б. Страуструпом как переусложнённый, рассчитанный на программистов-экспертов, в то время как изначальная идея состояла в создании механизма, облегчающего жизнь «среднему» программисту. Автор языка C++ предложил пять радикальных шагов по упрощению концептов, однако комитет по стандартизации не видел возможности обсуждать дополнительные предложения до принятия нового стандарта: сроки для внесения изменений подходили к концу. В итоге концепты были оставлены для будущих редакций стандарта языка.

3 Остутствие модулярного контроля типов и ретроактивного моделирования в концептах C++1z

Проект концептов для стандарта 2017 года был создан с нуля, но в соответствии с опытом, полученным при разработке концептов C++0x. Основные задачи, стоявшие перед авторами нового проекта, можно сформулировать следующим образом.

1. Простота и понятность для «среднего» программиста.
2. Отказ (возможно, временный) от средств, породивших наибольшие споры при обсуждении концептов C++0x.
3. Учёт объективной сложности адаптации промышленных компиляторов.
4. Обратная совместимость со стандартом языка 1998 года, в том числе, с неограниченными шаблонами (как и у концептов C++0x).

В этом разделе указываются две черты концептов C++1z, который вытекают из этих целей и отличают их от концептов C++0x: они связаны в первую очередь с двумя проблемными пунктами списка, приведённого в прошлом разделе, и составляют разницу между двумя проектами, видную на таблице 1 (с. 256) — следует лишь найти строки, в которых различаются столбцы этих проектов. В следующих двух разделах обсуждаются другие черты концептов C++1z в соответствии с таблицей 1.

Отсутствие контроля типов внутри шаблонов — возможно, самая неожиданная черта концептов C++1z, если смотреть на них без учёта изложенной выше предыстории. Контроль типов внутри шаблонов представлял потенциально особую сложность для адаптации промышленных компиляторов, как упоминалось в предыдущем разделе. Связано это, в част-

ности, с тем, что новые ограниченные шаблоны должны были успешно сосуществовать и взаимодействовать со старыми, неограниченными шаблонами C++98. Если, к примеру, из шаблона первого типа вызывался шаблон второго, то проконтролировать корректность требований внешнего шаблона было достаточно сложно: фактически, задача свелась бы, как и прежде, поиску ошибок путём подстановки шаблонных аргументов. Обработка подобных сочетаний существенно усложняла компилятор, не принося никакого выигрыша по сравнению со старым подходом стандарта 1998 года.

Таким образом, в концептах C++1z было решено полностью исключить проверку соответствия требований, наложенных в заголовке шаблона, и определения этого шаблона. То есть проверяется лишь соответствие аргументов шаблона в месте его использования, в то время как в теле шаблона параметры обрабатываются компилятором так же, как это было с шаблонами C++98. Данный подход считается разумным компромиссом в связи с тем, что автор обобщённой компоненты зачастую более квалифицированный специалист, чем её пользователь: он должен сформулировать набор ограничений к аргументам своей компоненты и написать её определение так, чтобы оно соответствовало этим требованиям, не рассчитывая на помощь компилятора в этой задаче. Далее, многочисленные использования однажды написанной компоненты будут уже проверяться именно компилятором.

Этим изменением объясняется различие двух проектов концептов с точки зрения **модулярного контроля типов** (см. таблицу 1, с. 256) — параметра сравнения, который был введён в таблицу в статье [2], но определение которого там не дано, его (в нужном здесь контексте) можно найти, например, в [11]. Модулярный контроль типов означает, что если определение обобщённой компоненты было однажды принято компилятором, то при любой подстановке аргументов, удовлетворяющих ограничениям, повторная проверка определения не требуется. В концептах C++0x это условие удовлетворялось лишь частично по причине возможного использования неограниченных шаблонов внутри ограниченных. В концептах C++1z вначале проверяется соответствие аргументов требованиям, однако во время подстановки могут произойти ошибки в том случае, если автор шаблона неточно описал эти требования.

Полностью автоматическая проверка соответствия типов наложенным ограничениям или отказ от моделей концептов — вторая важная черта, отличающая концепты C++1z. Как упоминалось в прошлом разделе, меха-

низм моделей концептов вызывал много вопросов, связанных с запретом или разрешением генерации таких моделей. В проекте для C++1z было решено максимально упростить использование концептов с помощью удаления самого понятия моделей концептов.

Таким образом, если тип не может быть однозначно и автоматически отображён на нужный набор требований, то он будет отвергнут компилятором. Имеющийся способ решения этой проблемы — модифицировать рассматриваемый тип нужным образом, зачастую не представляет трудностей, но разрушает совместимость с кодом, который уже использует старое определение данного типа. Эта проблема непосредственно связана с одним из важных свойств обобщённого программирования, известным как **ретроактивное моделирование** — см. таблицу 1 (с. 256). Ретроактивное моделирование предполагает возможность определения того, каким образом тип удовлетворяет концепту, без изменения типа. С удалением моделей концептов говорить о поддержке этого свойства если и можно, то только в очень ограниченном объёме, а именно: всегда можно добавлять перегруженные версии свободных функций, работающих с объектами данного типа. Однако наличие свободных функций является лишь одним из нескольких типов требований, которые могут накладывать концепты.

Завершая обсуждение связи концептов C++0x и концептов C++1z, отметим, что в последних по сравнению с первыми были исключены также:

– аксиомы — средство описания семантических ограничений,
– синтаксическая поддержка уточнения (refinement) или, попросту, «наследования» концептов.

Аксиомы это достаточно высокоуровневые правила, которые в общем случае невозможно проверить с помощью компилятора. Пример аксиомы приведён в листинге 3: здесь показано, что полугруппой является пара из типа `T` и бинарной операции над элементами этого типа, причём операция должна быть ассоциативной — это условие сформулировано в виде аксиомы.

Аксиомы предполагались в помощь компилятору для проведения оптимизирующих преобразований. Соответствующие исследования [12] проводились несколькими исследователями, поддерживавшими эту идею, однако число их было слишком мало, чтобы обеспечить вхождение механизма с таким нечётким назначением в новый упрощённый проект концептов.

Листинг 3 Аксиома ассоциативности — часть определения концепта полугруппы

```
concept Semigroup<typename Op, typename T> :
        CopyConstructible<T> {
    T operator()(Op, T, T);

    axiom Associativity(Op op, T x, T y, T z) {
        op(x, op(y, z)) == op(op(x, y), z);
    }
}
```

Пример уточнения также приведён на листинге 3: тип `T` в концепте полугруппы должен иметь конструктор копий (`CopyConstructible<T>`). Как замечено Дугом Грегором, одним из авторов концептов C++0x, в предложении по их упрощению [13], в отсутствии моделей концептов уточнения полностью эквивалентно ограничениям, которые можно накладывать на тип внутри определения концепта.

4 Перегрузка на основе концептов и связанная с ней проблема

Подтверждение остальных (кроме двух, разобранных в прошлом подпункте) показателей для сравнения средств обобщённого программирования таблицы 1 (с. 256) в столбце C++1z не представляет большой проблемы, хотя конкретный синтаксис текущего проекта концептов продолжает меняться. Между тем, один из этих параметров, а именно перегрузка на основе концептов, оставляет вопрос в связи с удалением из проекта моделей концептов: в техническом отчёте 2005 года ведущие специалисты в области обобщённого программирования, Дуг Грегор и Джереми Сик, высказали идею о том, что для успешной реализации перегрузки на основе концептов необходимо наличие механизма явного сопоставления концепта и типа, который должен ему удовлетворять.

Пример Грегора и Сика достаточно прост и интересен, чтобы его рассмотреть и понять, как решают возникающую в нём проблему авторы концептов C++1z. Отметим сразу, что хотя упомянутое решение можно найти в описании проекта изменений в стандартную библиотеку в связи с запланированным добавлением концептов C++1z, само описание изменений библиотеки весьма обширно, а авторы новых концептов не вынесли это важное, по нашему мнению, решение ни в одну из известных нам статей или докладов, посвящённых новому элементу языка.

Таблица 2. Категории итераторов в языке программирования C++

<table>
<tr><th colspan="3">Категория итераторов[10]</th><th colspan="2">Операции</th></tr>
<tr><td rowspan="4">Произвольного доступа (Random Access)</td><td rowspan="3">Двунаправленный (Bidirectional)</td><td rowspan="2">Прямой (Forward)</td><td>Ввода (Input)</td><td>чтение, инкремент (без многократного прохода)</td></tr>
<tr><td></td><td>инкремент (с многократным проходом)</td></tr>
<tr><td></td><td></td><td>декремент</td></tr>
<tr><td></td><td></td><td></td><td>произвольный доступ</td></tr>
</table>

Листинг 4 Перегрузка на основе концептов: конструктор типа `vector` с разной эффективностью

```
// O(lg n) allocations
template<Input_iterator Iter>
vector(Iter first, Iter last)
{
    while (first != last) push_back(*first++);
}

// 1 allocation
template<Forward_iterator Iter>
vector(Iter first, Iter last)
{
    auto n = distance(first, last);
    reserve(n);
    while (n--) push_back(*first++);
}
```

Пример, по существу, ставит проблему концептов, которые различаются только семантически. В качестве примера таких концептов можно взять широко распространённые в стандартной библиотеки C++ категории итераторов (таблица 2), которые планируется конвертировать в концепты в будущем стандарте. Из таблицы 2 видно, что итераторы ввода и прямые итераторы отличаются лишь семантикой инкремента.

В листинге 4 приведён пример перегрузки функций-членов, а именно, конструкторов типа `vector`, принимающих итераторы начала и конца диапазона, элементы которого надо загрузить в создаваемый вектор. Пе-

[10]В таблице опущены итераторы вывода, которые аналогичны итераторам ввода, но позволяют модифицировать элементы, на которые они указывают. Любой итератор из приведённых категорий, который удовлетворяет требованиям итератора вывода, называется *изменяемым* (*mutable*).

Листинг 5 Концепт итератора ввода

```
template<typename I>
concept bool InputIterator =
    WeakInputIterator<I>() && EqualityComparable<I>() &&
    Derived<IteratorCategory<I>, input_iterator_tag>();
```

регрузка осуществляется на основе категории итератора, который задаётся концептом: для итератора ввода необходимо многократно выделять ($O(\lg n)$ выделений, если расширение происходит с помощью кратного увеличения резерва памяти), в то время как прямой итератор позволяет заранее вычислить количество элементов в полученном диапазоне и обойтись одним выделением.

Проблема состоит в том, что, как отмечалось, выше любой тип, (синтаксически) входящий в категорию итератора ввода, будет принадлежать также и категории прямого итератора. Как показал анализ предложения по редактированию стандартной библиотеки в связи с планируемым добавлением концептов, семантическое различие между разными типами авторы планируют, как и в старом стандарте кодировать заранее определёнными типами-метками (tag), для которых внутри типов-итераторов задаётся стандартный псевдоним `iterator_category`. На листинге 5 показано, как примерно должен выглядеть при таком подходе концепт итератора ввода.

Из листинга 5 видно, что данный концепт уточняет (refines) концепт `WeakInputIterator`, который включает некоторые элементарные свойства итераторов, такие как наличие операций разыменования и продвижения. Ограничения на тип-параметр `I` состоят в том, что объекты этого типа должны быть сравнимы на равенство (`EqualityComparable`) и должны содержать вложенный синоним типа под названием `iterator_category` (его значение возвращает выражение `IteratorCategory<I>`), который наследуется (в том числе — и это наиболее распространённый случай — равен) от стандартного типа-метки `input_iterator_tag`.

Отметим, что в этом примере используется ограничение в виде подтипирования, в том числе, сравнения на равенство, что является одним из показателей в таблице 1 (с. 256). Строгое равенство обеспечивается аналогичным `Derived` отношением `Same`. Оба они вычисляются с помощью средств заголовка `type_traits` стандартной библиотеки, появившегося в C++11, и, таким образом, не требуют существенной модификации существующих компиляторов.

Листинг 6 Шаблон псевдонима типа

```
template<typename T>
using VecStack = std::stack<T, std::vector<T>>;
```

Листинг 7 Шаблон функции поиска: пример мультипараметрического концепта `Equality_comparable` («сравнимые на равенство»)

```
template<typename S, typename T>
    requires Sequence<S>() && Equality_comparable<T, Value_type<S>>()
Iterator_type<S> find(S&& sequence, const T& value);
```

5 Другие показатели поддержки обобщённого программирования для концептов C++1z

Области видимости моделей концептов — возможно, самое любопытное средство, которым выделяется язык Scala: как видно в таблице 1 (с. 256), он единственный осуществляет поддержку этого показателя. Следует заметить, что управление видимостью разных моделей одного концепта, о котором идёт речь, подразумевает возможность собственно определения этих моделей. Типичный пример: концепт Моноид может быть смоделирован целыми числами с помощью операции сложения и константы нуль либо с помощью операции умножения и константы один. Поскольку в концептах C++1z вообще нет средств определения моделей программистом, то говорить о поддержке управления областями видимости не приходится вовсе.

Псевдонимы типов не связаны с проектом концептов, они имелись в стандарте C++ с самого начала (ключевое слово `typedef`), а в 2011 году было добавлено давно ожидаемое средство определения параметрических псевдонимов типов. К примеру, стандартный шаблон стека реализован на базе дека, если требуется использовать в программе стеки на базе вектора, можно создать псевдоним, как показано в листинге 6. Далее этот псевдоним используется как обычный шаблон с одним параметром.

Мультипараметрические концепты не представляют сложности для текущего проекта, соответствующий пример показан на листинге 7: от типа искомого элемента (`T`) и типа элементов последовательности, в которой проводится поиск, (`Value_type<S>`) требуется, чтобы они были сравнимы на равенство.

Ассоциированные типы и **ограничения на ассоциированные типы** показаны в примере на листинге 8: от типа-аллокатора требуется,

Листинг 8 Концепт Аллокатора (менеджера памяти): пример требования ассоциированного типа и ограничения на него

```
template<typename A>
concept bool Allocator()
{
    return requires () {
        typename A::pointer;                          // (1)
        requires Pointer<typename A::pointer>; // (2)
        // ...
    };
}
```

Листинг 9 Пример использования допустимых выражений в определении концепта `Equality_comparable`

```
template<typename T>
concept bool Equality_comparable()
{
    return requires (T a, T b) {
        {a == b} -> bool;
        {a != b} -> bool;
    };
}
```

чтобы (1) у него был вложенный синонима типа `pointer` и (2) чтобы этот синоним обозначал указательный тип C++.

В листинге 8 показан пример определения концепта с требованиями на ассоциированные типы. Для полноты изложения приведём также пример требований на операции над типом-параметром (листинг 9). В этом примере показано, как `requires`-выражение позволяет вводить псевдопеременные параметра-типа `T` (в этом примере их имена `a` и `b`) и требовать, чтобы выражения сравнения на равенство и на неравенство были допустимыми для этих переменных и возвращали тип `bool`.

Раздельная трансляция шаблонов (в том числе, концептов), как и раньше, остаётся недостижимым требованием. Стандарт C++ предъявляет достаточно жёсткие требования к воспроизведению контекста определения шаблона при подстановке аргументов: этот контекст должен содержать такое количество информации, которое сложно сохранить как-то иначе, кроме как предъявив исходный код шаблона.

Заключение

В данной статье новый проект концептов для языка C++ рассмотрен с точки зрения известного набора критериев поддержки обобщённого программирования в современных языках программирования. В сравнении с такими языками как Scala или Haskell (см. таблицу 1) данный проект оказывается достаточно ограниченным. Как было показано в работе, он уступает и первому проекту концептов C++ в двух важных отношениях: в новом проекте не предполагается контроль типов внутри шаблонов, а также явное моделирование ограничений.

Несмотря на слабые стороны нового проекта концептов C++, он, по всей видимости, сильно упростит задачу генерации легко читаемых ошибок компилятора. Это являлось одной из главных задач авторов новых концептов.

Рассмотренные в данной статье средства можно опробовать на практике. В августе 2015 года прототипная реализация концептов на основе компилятора GCC была добавлена в основную ветвь разработки этого компилятора [14]. Это изменение планируется зафиксировать в версии 6 компилятора, которая запланирована к выходу весной 2016 года. Однако уже сейчас можно самостоятельно собрать данную версию компилятора из исходных кодов либо воспользоваться онлайн-ресурсами, предоставляющими веб-интерфейс к такой сборке [15]. Для включения механизма концептов при компиляции программ следует указывать ключ `-std=C++1z`.

Список литературы

[1] A Comparative Study of Language Support for Generic Programming / R. Garcia [и др.] // SIGPLAN Not. — New York, NY, USA, 2003. — Окт. — Т. 38, № 11. — С. 115–134. — ISSN 0362-1340. — DOI: 10.1145/949343.949317.

[2] *Oliveira B. C., Moors A.*, Odersky M. Type Classes As Objects and Implicits // SIGPLAN Not. — New York, NY, USA, 2010. — Окт. — Т. 45, No 10. — С. 341–360. — ISSN 0362-1340. — DOI: 10.1145/1932682.1869489.

[3] *Sutton A., Stroustrup B., Dos Reis G.* Concepts Lite: тех. отч. — 28 июня 2013. — ISO/IEC JTC1/SC22/WG21 N3701. — URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3701.pdf` (дата обр. 20.08.2015)

[4] *Sutton A.* C++ Extensions for Concepts PDTS: тех. отч. — 9 февр. 2015. — ISO/IEC JTC1/SC22/WG21 N4377. — URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4377.pdf` (дата обр. 20.08.2015).

[5] A Concept Design for the STL: тех. отч. — 13 янв. 2012. — ISO/IEC JTC1/SC22/WG21 N3351. — URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf (дата обр. 20.08.2015).

[6] *Stroustrup B.* The C++0x "Remove Concepts" Decision // Dr. Dobb's Journal. — 2009. — URL: http://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111 (дата обр. 30.07.2015).

[7] *Järvi J., Willcock J., Lumsdaine A.* Associated types and constraint propagation for mainstream object-oriented generics // OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object- oriented programming systems languages and applications. — ACM Press. New York, NY, USA : ACM Press, 2005. — С. 1–19. — ISBN 1-59593-031-0. — DOI: 10.1145/1094811.1094813.

[8] *Belyakova J., Mikhalkovich S.* Pitfalls of C# Generics and Their Solution Using Concepts // Proceedings of the Institute for System Programming. — Moscow, Russia, 2015. — Июнь. — Т. 27, No 3. — С. 29–45. — ISSN 2079-8156.

[9] Lightweight, flexible object-oriented generics / Y. Zhang [и др.] // 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). — Июнь 2015. — С. 436–445. — URL: http://www.cs.cornell.edu/andru/papers/genus (дата обр. 20.08.2015).

[10] ConceptGCC. — URL: http://www.generic-programming.org/software/ConceptGCC/ (дата обр. 30.07.2015).

[11] Concepts: Linguistic Support for Generic Programming in C++ / D. Gregor [и др.] // SIGPLAN Not. — New York, NY, USA, 2006. — Окт. — Т. 41, No 10. — С. 291–310. — ISSN 0362-1340. — DOI: 10.1145/1167515.1167499.

[12] *Tang X., Järvi J.* Axioms as generic rewrite rules in C++ with concepts // Science of Computer Programming. — 2015. — Т. 97, Part 3. — С. 320–330. — ISSN 0167-6423. — DOI: 10.1016/j.scico.2014.05.006. — Object-Oriented Programming and Systems (OOPS 2010)Modeling and Analysis of Compositional Software (papers from EUROMICRO SEAA'12).

[13] *Gregor D.* Simplifying C++0x Concepts: тех. отч. — 9 апр. 2013. — ISO/IEC JTC1/SC22/WG21 N3629. — URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3629.pdf (дата обр. 20.08.2015).

[14] *Merrill J.* Huge C++ PATCH to merge c++-concepts branch — Mail archive for the GCC project. — URL: https://gcc.gnu.org/ml/gcc-patches/2015-08/msg00377.html (дата обр. 20.08.2015).

[15] Wandbox: Social Compilation Service. — URL: http://melpon.org/wandbox (дата обр. 20.08.2015).