ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № 3(3), 2010, с. 13–36

УДК 004.021

Ю. А. Климов

Специализатор CILPE: частичные вычисления для объектно-ориентированных языков

Аннотация. В работе рассмотрена специализация программ на основе метода частичных вычислений применительно к программам на объектно-ориентированных языках. Дан обзор возможностей известных специализаторов для языков этого класса, приведено сравнение специализаторов. Описаны возможности специализатора CILPE и приведен пример его использования.

Ключевые слова и фразы: специализация программ, частичные вычисления, объектно-ориентированные языки программирования, специализатор CILPE.

Введение

С момента своего появления компьютеры применялись для решения задач все большей сложности. При этом увеличивалась и вычислительная мощность компьютеров. В 1965 году Гордон Мур обнаружил закономерность, впоследствии названую «Законом Мура»: количество транзисторов в микросхемах удваивалось каждые полтора–два года. Вместе с удвоением количества транзисторов в микросхеме, удваивалась и производительность компьютеров.

В то время как вычислительная мощность компьютеров росла экспоненциально, сложность задач, решаемых компьютерами, возрастала еще более быстрыми темпами, вследствие чего стало практически невозможно делать глобальную оптимизацию программ вручную. Таким образом, несмотря на рост производительности компьютеров, автоматическая оптимизация программ не только не потеряла свою актуальность, но и даже стало еще более важной.

Языки программирования также непрерывно развивались. В настоящее время для написания программ любого уровня — от простых программ до сложных систем — наиболее широко используются объектно-ориентированные языки типа C++, C# или Java.

Работа поддержана проектами РФФИ № 08–07–00280–а и № 09–01–00834–а.

Многие современные компиляторы обладают встроенными оптимизаторами. Они преобразуют программу в машинный код, оптимизируя один или несколько показателей, таких как: размер скомпилированной программы, среднее количество исполняемых инструкций, среднее время выполнения программы, объем используемой оперативной памяти и так далее. В результате таких оптимизаций получаем программу, эквивалентную исходной программе, но в некоторых случаях более оптимальную по этим показателям.

Для более эффективной оптимизации необходимо использовать дополнительную информацию о программе или об исполняемой системе. Например, для эффективной компиляции используется информация об инструкциях, поддерживаемых данным процессором. В результате такой компиляции получается программа, которая может быть исполнена только на конкретном процессоре, но более эффективно, чем программа, которая может исполняться на целом семействе процессоров.

Подход к оптимизации программ, основанный на учете дополнительной информации об условиях, в которых будет эксплуатироваться программа, называется специализацией программ. В частности, ограничения на условия эксплуатации могут заключаться в том, что часть исходных данных известна заранее и не меняется от одного запуска программы к другому. Или, в общем случае, ограничения на исходные данные могут быть заданы в виде каких-то условий. В таких случаях принято говорить, что выполняется специализация программы по отношению к ограничениям на исходные данные.

Методы специализации изначально развивались для функциональных языков, и в этом направлении был достигнут существенный прогресс, причем наиболее широко распространенным подходом является метод частичных вычислений (Partial Evaluation, PE) [22]. Для объектно-ориентированных языков методы специализации в настоящее время развиты в гораздо меньшей степени.

Основной особенностью объектно-ориентированных языков, отличающей их от функциональных языков, является наличие сложно устроенного состояния машины (стека, переменных, кучи). То есть в каждый момент (непосредственно или косвенно) для чтения или изменения доступны очень многие данные, тогда как в функциональных языках функция может читать значение только своих аргументов и создавать только новые данные, но не может изменить уже созданные данные.

Поэтому многие методы частичных вычислений, применяющиеся для функциональных языков, не могут быть напрямую применены для объектно-ориентированных языков.

В данной статье описывается метод частичных вычислений (разделы 1–4) и особенности объектно-ориентированных языков, которые должны учитываться при разработке метода (раздел 5). Приводится обзор существующих специализаторов для программ на объектно-ориентированных языках и описывается разработанный автором специализатор CILPE [1–11, 20, 24] (раздел 6). В завершении приводится пример его применения к программе на объектно-ориентированном языке C# платформы Microsoft.NET [28, 30] (раздел 7).

1. Специализация программ методом частичных вычислений

Оптимизация программ на основе использования априорной информации о значении части переменных называется специализацией.

Рассмотрим программу $f(x, y)$ от двух аргументов x и y и значение одного из ее аргументов $x = a$. Результатом специализации программы $f(x, y)$ по известному аргументу $x = a$ называется новая программа одного аргумента $g(y)$, обладающая следующим свойством: $f(a, y) = g(y)$ для любого y.

Частичные вычисления (Partial Evaluation, PE) [22] — это такой метод специализации, который заключается в получении более эффективного кода на основе использования априорной информации о части аргументов и однократного выполнения той части кода, которая зависит только от известной части аргументов (и не зависит от неизвестной части).

В процессе частичных вычислений операции над известными данными исполняются, а над неизвестными — переносятся в остаточную программу. Остаточная программа зависит только от неизвестной (на стадии специализации) части аргументов и будет исполняться, только когда значения этих аргументов станут известны. Цель частичных вычислений — генерация остаточной программы.

Операции и элементы данных, которые выполняются и используются на этапе генерации остаточной программы, называются статическими, и будут в дальнейшем обозначаться S, а остальные, оставленные в остаточной программе, — динамическими — D.

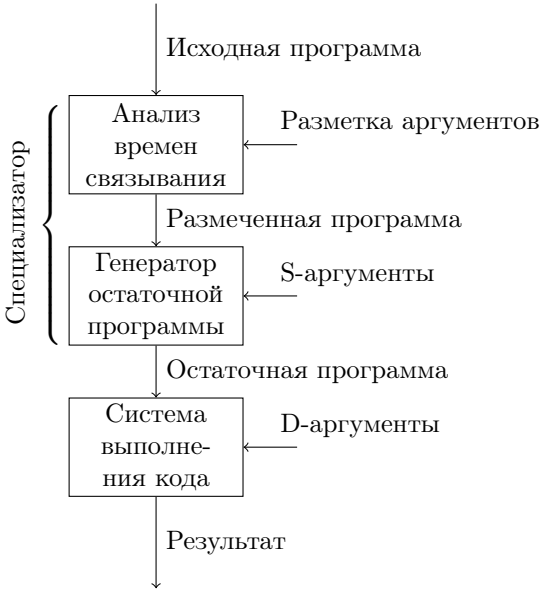

Рис. 1. Общепринятая структура специализатора, основанного на методе частичных вычислений.

Метод частичных вычислений основан на разделении операций и других программных конструкций на статические (S) и динамические (D). При этом понятие «статические операции и конструкции» не следует путать с понятием «статические методы и классы» (static), которое используется в языках программирования, например, C# и Java.

Часть метода специализации, отвечающая за разделение операций и данных, называется анализом времен связывания (Binding Time Analysis, BTA, BT-анализ) [4,6,8] (рис. 1). Вторая часть метода специализации, отвечающая за вычисление статической части программы и выделение динамической части в отдельную программу, называется генератором остаточной программы (Residual Program Generator, RPG) [2, 9]. В этой части, собственно, и происходят «частичные вычисления».

Для примера рассмотрим функцию возведения в степень, написанную на функциональном языке Haskell (рис. 2). И поставим задачу

проспециализировать метод toPower по второму аргументу, равному 5.

```
toPower :: Double -> Int -> Double
toPower x 0 = 1.0
toPower x n = x*(toPower x (n-1))
```

Рис. 2. Пример: исходная программа на функциональном языке.

Тогда мы считаем, что первый аргумент — динамический, а второй — статический, и строим разметку всей программы (рис. 3).

```
toPower :: Double^D -> Int^S -> Double^D
toPower x^D 0^S = 1.0^D
toPower x^D n^S = x^D *^D (toPower^S x^D (n^S -^S 1^S))
```

Рис. 3. Пример: размеченная программа на функциональном языке.

Проведя генерацию остаточной программы, выполнив все S-операции и оставив все D-операции, получаем остаточную программу (рис. 4).

```
toPower5 :: Double -> Double
toPower5 x = x*x*x*x*x*1.0
```

Рис. 4. Пример: результат специализации программы на функциональном языке.

В результате остаточная программа не содержит проверок показателя степени на равенство нулю и операции с этим показателем.

Аналогично можно провести специализацию и императивной программы возведения в степень (рис. 5). Разметка программы строится аналогично (рис. 6).

В результате специализации останутся только операции с переменной x (рис. 7).

```
double toPower (double x, int n) {
  if (n == 0)
    return 1.0;
  else
    return x*toPower(x, n-1);
}
```

Рис. 5. Пример: исходная программа на императивном языке.

```
double^D toPower (double x^D, int n^S)
  if^S (n^S ==^S 0^S)
    return^D 1.0^D;
  else^S
    return^D x^D *^D toPower^S(x^D,n^S -^S 1^S);
}
```

Рис. 6. Пример: размеченная программа на императивном языке.

```
double toPower5 (double x) {
  return x*x*x*x*x*1.0;
}
```

Рис. 7. Пример: результат специализации программы на императивном языке.

Следует отметить, что BT-анализ и генератор остаточной программы могут работать как одновременно, так и по очереди: сначала анализ времен связывания, затем генератор остаточной программы. В первом случае мы получим так называемый online PE, во втором — offline PE. В дальнейшем мы будем говорить только об offline PE.

В offline частичном вычислителе сначала конструкции программы размечаются на статические (S) и динамические (D) с помощью анализа времен связывания. В процессе анализа могут быть выполнены некоторые преобразования программы для улучшения качества разметки (например, добавлены операции поднятия Lifting).

После этого генератор остаточной программы, используя частично заданные аргументы и размеченную программу, выполняет все

статические операции, а все динамические операции переносит в остаточную программу. В результате получаем остаточную программу, зависящую только от аргументов, неизвестных во время специализации.

Далее подробно рассмотрим оба этапа offline частичных вычислений: анализ времен связывания и генерация остаточной программы.

2. Определение корректности специализатора (основное свойство специализатора)

Рассмотрим программу f от переменных $x, \ldots, y, u, \ldots, v$ и множество значений переменных D. Пусть значения аргументов $x, \ldots, y$ известны и равны $x_0, \ldots, y_0 \in D$. Пусть в результате специализации программы f при известных значениях $x_0, \ldots, y_0$ переменных $x, \ldots, y$, получилась программа g, зависящая от переменных $u, \ldots, v$, неизвестных во время генерации остаточной программы. Тогда для любых значений $u_1, \ldots, v_1 \in D$ переменных $u, \ldots, v$ результат вычисления программы f от значений $x_0, \ldots, y_0, u_1, \ldots, v_1$ и программы g от значений $u_1, \ldots, v_1$ должны совпадать (рис. 8).

Если $spec(f, x_0, \ldots, y_0) = g$, где $x_0, \ldots, y_0 \in D$
то $\forall u_1, \ldots, v_1 \in D : f(x_0, \ldots, y_0, u_1, \ldots, v_1) = g(u_1, \ldots, v_1)$

Рис. 8. Определение корректности специализации.

3. Анализ времен связывания

Цель анализа времен связывания [4, 6, 8] — разделить программу на статическую и динамическую части. Такое разделение может быть произведено несколькими способами. Однако необходимо, чтобы статические операции в таком разделении всегда получали только статические данные, а операции, которые в каких-то случаях могут получить динамические данные, должны быть размечены как динамические.

Такое разделение, как правило, может быть выполнено различными способами. Можно даже все данные отнести к динамическим.

И для любого способа остаточная программа будет получаться правильной (если разделение построено согласно описанным ниже правилам). Но чем больше операций будет отнесено к статическим, тем более эффективной будет остаточная программа.

Для разделения на статическую и динамическую части анализ времен связывания использует BT-разметку: разметку инструкций, элементов стека и переменных. Построенная разметка должна быть внутренне согласованной, удовлетворять правилам, которые будут описаны ниже: например, всякая статическая операция должна иметь статические аргументы, а динамическая — динамические. Статическое данное может стать аргументом динамической операции, если оно будет предварительно преобразовано специальной операцией поднятия Lifting. Обратный переход — из динамики в статику — невозможен.

Для достижения согласованности, помимо добавления операций поднятия, исходная программа может быть подвергнута и другим преобразованиям.

Возможно несколько вариантов реализации BT-анализа:

(1) Моновариантный по переменным — каждую переменную разрешается разметить одним способом независимо от точки ее использования в программе.
(2) Поливариантный по переменным — разрешается в разный частях программы размечать одну и ту же переменную по-разному. Это позволяет более эффективно производить специализацию кода, в котором одна и та же переменная используется в разных местах программы с разными целями.
(3) Моновариантный по операциям — запрещается преобразовывать и размножать код внутри метода и, следовательно, приходится каждую операцию размечать только одним способом.
(4) Поливариантный по операциям — можно преобразовывать и размножать код внутри метода, что позволяет одному участку кода исходной программы сопоставить несколько участков выходной размеченной программы с разными BT-разметками. Это позволяет более качественно разметить программу, но существует опасность многократного роста размера программы.

(5) Моновариантный по методам — каждый метод размечается только одним способом. В этом случае все использования метода будут одинаково размечены, хотя метод мог использоваться в совершенно разных условиях (например, в одном случае все известно, в другом — ничего не известно)
(6) Поливариантный по методам — каждый метод может быть размечен в зависимости от его использования — разметки аргументов и результатов в точке вызова. Также во время специализации некоторые методы целесообразно развернуть — подставить в место вызова тело метода (inline). В таких случаях программист должен добавить соответствующую аннотацию. Если метод не разворачивается, то это накладывает на его разметку дополнительные требования. А развертка метода может привести к увеличению программы и бесконечной работе специализатора.
(7) Моновариантный по классам — каждый класс размечается только одним способом. В таком варианте все объекты данного класса будут одинаково проспециализированы, независимо от особенностей использования.
(8) Поливариантный по классам — каждый класс размечается одним или несколькими способами. Это позволяет по-разному разметить различные использования объектов данного класса.

В итоге, анализ времен связывания по программе и начальной разметке, которая содержит разметку входных данных и, быть может, разметку других элементов, заданных пользователем, должен построить BT-разметку всех операций и используемых данных, удовлетворяющую правилам согласованной разметки в программе.

4. Генерация остаточной программы

После построения корректно размеченной программы, генератор остаточной программы, с помощью частичных вычислений, строит окончательный результат специализации: остаточную программу [2, 9].

На вход генератору остаточной программы поступают размеченная программа и значения S-переменных. По этим данным он строит остаточную программу, зависящую только от части переменных — только от D-переменных, динамических исходных данных, неизвестных во время специализации программы.

Если разметка программы была построена согласованно, то генератор остаточной программы построит программу, которая будет эквивалентна исходной программе при указанных значениях S-переменных и всех значениях D-переменных. Это основное свойство генератора остаточной программы.

Генератор остаточной программы производит обобщенное выполнение программы: S-инструкции выполняются, а D-инструкции переходят в остаточную программу. Собственно на этом этапе выполняется часть операций, откуда и происходит термин «частичные вычисления».

В процессе порождения остаточной программы генератор сравнивает состояния для возможного построения циклов или рекурсии в остаточной программе. Но при некоторых начальных данных, т.е. для некоторых программ и их разметок, генератор остаточной программы может работать бесконечно долго, пытаясь построить остаточную программу.

Если работа генератора остаточной программы завершается, то получается остаточная программа эквивалентная исходной. Если в задании на частичные вычисления часть аргументов метода была статической, то в результате порождается новый метод, зависящий только от оставшихся динамических аргументов. И вызов метода необходимо вставить в остаточную программу.

Если же в задании на частичные вычисления все аргументы были динамическими, а частичные вычисления производились только по внутренним статическим данным, то в результате получаем метод, полностью эквивалентный исходному методу. Поэтому в остаточную программу вставляется эквивалентный метод (хотя, может быть, и оптимизированный).

5. Особенности объектно-ориентированных языков

Функциональные языки обладают множеством свойств, облегчающих анализ программ. К числу таких свойств можно отнести отсутствие побочных эффектов, отсутствие глобального состояния, передача аргументов и результатов по значению, и т.п.

Объектно-ориентированные языки не обладают этими удобными свойствами, поэтому при анализе и преобразовании программ приходится учитывать [1, 5]:

(1) Наличие глобального изменяемого состояния (в функциональных языках такое состояние отсутствует).
(2) Наличие ссылок на элементы (объекты) глобального состояния (которые в некоторых языках могут идентифицироваться по именам, известным во время компиляции программы).
(3) Объекты могут состоять из полей (обращение к которым происходит по имени, известному во время компиляции), либо являться массивами (обращение к элементам которых происходит по номерам, неизвестным во время компиляции программы).
(4) Передача аргументов и результатов по ссылке (в функциональных языках аргументы и результаты передаются по значению).
(5) Представление программы в виде последовательности инструкций (а не в виде графа передачи данных).
(6) Наличие конструкций передачи управления (цикл, goto).
(7) Наличие конструкции наследования классов.
(8) Наличие виртуальных методов.

6. Обзор существующих специализаторов для объектно-ориентированных языков

Существует много работ посвященных частичным вычислениям для функциональных языков [22]. Одной из фундаментальных работ, посвященных методам частичных вычислений, является работа N.D. Jones, C.K. Gomard, P. Sestoft «Partial Evaluation and Automatic Program Generation» 1993г. [22]. В ней в основном рассматриваются методы частичных вычислений для функциональных языков. Методам частичных вычислений для императивных языков посвящена всего одна глава — «Частичные вычисления для языка C».

Однако, имеется гораздо меньше работ по специализации и частичным вычислениям для языков с императивными особенностями, к рассмотрению которых мы и переходим.

6.1. Ламбда-исчисления с императивными конструкциями

Одним из направлений исследований частичных вычислителей для языков с императивными конструкциями является разработка частичного вычислителя для ламбда-исчисления с побочными эффектами, выполненная Кэнъити Асай (Kenichi Asai) из университета города Токио [16]. В представленном ламбда-исчислении присутствуют операции для изменения значения выражения. В работе предложен алгоритм разделения данных на изменяемые и неизменяемые и

построен частичный вычислитель, использующий такое разделение. В нем обращения к потенциально изменяемым данным переходят в остаточную программу.

Вторым направлением работ К. Асай — разработка частичного вычислителя для ламбда-исчисления, способного одновременно генерировать код для вычисления выражения и вычислять это выражение [17]. Такая возможность используется для преобразования данных из статических в динамические: в момент преобразования выражение строится не сразу, а постепенно, и его части могут использоваться для преобразования других данных из статических в динамические.

Третьим направлением работ группы под руководством К. Асай является создание частичного вычислителя для ламбда-исчисления с продолжениями (continuations) [18].

Разработанные К. Асай методы расширяют возможности классического метода частичных вычислений, делая его применимым к ламбда-исчислению с побочными эффектами. Этого все же недостаточно для применения этого метода к объектно-ориентированным языкам.

6.2. Специализаторы для языка C

Первыми работами по созданию специализаторов для императивных языков являются работы по созданию специализатора для языка C. Однако язык C сложен для автоматического анализа: его семантика не всегда достаточно прозрачна. Также трудности связаны со структурой данных, с которыми он работает: фактически вся память представляет собой один большой массив, из-за чего затруднительно реализовать достаточное для практических задач разделение данных на статические и динамические.

Также и сами программы на языке C обычно пишутся в сложном для анализа стиле: сам язык провоцирует программиста писать программы в таком стиле, который затрудняет автоматический анализ программ.

6.2.1. *C-Mix*

Еще в начале 90х годов Ларс Оле Андерсен (Lars Ole Andersen) в университете Копенгагена в Дании разработал [13–15] специализатор C-MIX для языка C.

Специализация программы на языке C проводится в несколько этапов:

(1) Первый этап — анализ ссылок. Для каждой переменной-ссылки строится множество переменных и областей памяти, на которые она может ссылаться. Если несколько ссылок-переменных в результате анализа могут ссылаться на одно и то же, то они должны быть размечены одинаково.
(2) Второй этап — построение разметки времен связывания. Ссылки на одно и то же должны быть размечены одинаково.
(3) Третий этап — генерация остаточной программы.

6.2.2. *Tempo*

В результате развития идей, заложенных в C-Mix, группой Шарля Консель (Charles Consel), Джулии Лаваль (Julia L. Lawall) (Университет города Копенгагена, Франция) и Анн-Франсуаза Лё Мёр (Anne-Francoise Le Meur) в лаборатории исследований информатики и вычислительной техники в Бордо во Франции был создан специализатор Tempo [21]. Это специализатор для языка C, во многом аналогичный C-Mix.

6.3. Специализация для Java байт-кода

6.3.1. *Работы в университете города Токио*

В университете города Токио под руководством Акинори Ёнэдзава (Akinori Yonezawa) и Хидэхико Масухара (Hidehiko Masuhara) разрабатывается специализатор для Java байт-кода, который можно было бы встроить в JIT-компилятор байт-кода.

Описанный в работе [25] специализатор обрабатывает только примитивные данные и статические методы. Допускается различная разметка переменных в различных точках программы. В работе показано, как для метода построить систему ограничений на разметку локальных переменных и инструкций.

В поздней работе [12] группой Рейналд Аффелдт (Reynald Affeldt), Хидэхико Масухара (Hidehiko Masuhara), Эйдзиро Сумми (Eijiro Summi), Акинори Ёнэдзава (Akinori Yonezawa) предложено расширение специализатора, позволяющее обрабатывать и объекты. Разметка переменных описывается деревьями: S или D в вершине дерева определяет статический или динамический объект будет находиться

в этой переменной, а поддеревья определяют разметку полей объекта.

Анализ метода основан на разделении объектов на локальные объекты (которые создаются внутри этого метода) и нелокальные (которые передаются методу извне).

Локальные объекты делятся на те, которые могут быть выданы в качестве результата работы метода, и те, которые используются только внутри метода. Нелокальные объекты разделяются на те, в поля которых может осуществляться присваивание, и те, в поля которых присваивание не происходит. Инструкции создания локальных объектов, которые могут быть выданы, и присваивания в поля нелокальных объектов обязаны размечаться динамически и переходить в остаточную программу.

После разделения строится разметка всех инструкций в программе.

В обеих работах [12, 25] описаны специализаторы времени исполнения (run-time specialization). По построенной разметке строится метапрограмма: программа, генерирующая результат специализации по известным значениям статических аргументов. Во время выполнения остаточной программы, сначала по статической части аргументов строится специализированная версия метода, а затем эта специализированная версия метода выполняется.

6.3.2. *Работы П. Бертелсена*

В королевском ветеринарно-земледельческом университете (Royal Veterinary and Agriculture University) в городе Копенгагене в Дании Питер Бертелсен (Peter Bertelsen) занимался семантикой и анализом времен связывания Java программ.

В своей работе [19] П. Бертелсен рассматривает подмножество Java байт-кода. Это подмножество содержит инструкции для работы с локальными переменными, операции на стеке с примитивными данными и массивами и инструкции перехода Goto и If. Инструкций вызовов методов и возможность создания объектов нет.

В работе описаны допустимые разметки и ограничения на корректную разметку программы, и предложен метод нахождения решения для системы ограничений.

Описан генератор остаточной программы, который по размеченной программе и значениям статических аргументов строит остаточную программу.

6.3.3. *Работы У.П. Шульца*

В университете города Реннес (Rennes) во Франции Ульрик Пагх Шульц (Ulrik Pagh Schultz) в рамках своей диссертации разработал частичный вычислитель для подмножества Java байт-кода [26, 27].

Описанное подмножество относительно широко: можно описывать классы, конструкторы и методы. Разрешены операции над примитивными данным и виртуальные вызовы.

Но отсутствуют операции работы с массивами, возможность изменять значения полей объектов, тело метода представляется не в виде последовательности инструкций, а в виде одного выражения.

Как и в предыдущих работах, программе сопоставляется система ограничений на разметку. Но в отличие от них, разметка строится для описаний классов, а не для переменных в программе. Разметка программы строится путем решения системы ограничений.

На основе размеченной программы генератором остаточной программы по известным значениям статических переменных строится остаточная программа.

В следующих главах У.П. Шульц описывает усовершенствования, которые необходимы для работы с реальными программами:

(1) class polyvariance — класс размечается несколькими способами и каждому new соответствует одна из разметок;
(2) method polyvariance — для каждого вызова метода метод размечается отдельно;
(3) alias polyvariance — анализ, помечающий где могут быть одинаковые данные, а где — разные. Желательно, чтобы и alias analysis был поливариантным (размножал классы и методы).

Однако математического аппарата для реализации такого рода усовершенствований не предлагается.

6.4. Специализатор CILPE

На основе метода частичных вычислений автором создан специализатор CILPE [1–11, 20, 24] для внутреннего языка Common Intermediate Language (CIL) платформы Microsoft .NET [29, 30].

Язык CIL является объектно-ориентированным стековым языком. Он не предназначен для ручного написания программ. Но все

языки платформы Microsoft .NET, такие как C#, J#, SML.NET, компилируются в язык CIL. Наличие единого внутреннего языка позволяет реализовать специализатор только для этого языка, а затем использовать его для специализации программ, написанных на различных языках программирования.

Основным языком платформы Microsoft .NET является высокоуровневый объектно-ориентированный язык C# [28]. Язык CIL хотя и не обладает всеми выразительными средствами языка C#, однако поддерживает все понятие языка C#. Это позволяет обрабатывать результат компиляции программ с языка C# на язык CIL без потери необходимой для специализатора информации о программе.

Специализатор CILPE поддерживает ограниченное, но широко используемое подмножество операций языка CIL. Не поддерживаются только исключения, передача аргументов по ссылке и структуры.

В работах [8, 9, 11] формально описан специализатор CILPE: описан внутренний язык SOOL [11], анализ времен связывания [8] и генератор остаточной программы [9]. В работе [10] доказана корректность специализатора.

Используемые в специализаторе CILPE техники и идеи могут быть легко адаптированы для различных объектно-ориентированных языков, например, для языка Java байт-код (Java Byte Code, JBC) платформы Java.

В рассмотренных выше работах предложены различные методы частичных вычислений для подмножеств объектно-ориентированных языков C# или Java. При этом, в каждом из этих подмножеств наложены такие сильные ограничения, которые обычно не выполняются для реальных программ на языках C# или Java. В отличие от рассмотренных работ, специализатор CILPE не накладывает таких ограничений (таблицы 1 и 2) [1, 5].

Для практических применений к программам на объектно-ориентированных языках специализатор CILPE обеспечивает:

(1) Обработку инструкций передачи управления Goto и If.
(2) Поддерживает работу с объектами и массивами, допускает возможность изменения полей объектов и элементов массива в программе.
(3) Обладает единым анализом времен связывания (не проводит отдельный alias analysis).

	Инструкции передачи управления Goto и If	Работа с	
		массивами	объектами
Х. Масухара	да	нет	да
П. Бертелсен	да	да	нет
У.П. Шульц	нет	нет	да
CILPE	да	да	да

Таблица 1. Классификация специализаторов на базе метода частичных вычислений по полноте реализации объектно-ориентированного языка.

	Поливариантность по			
	переменным	инструкциям	методам	классам/ массивам
Х. Масухара	да	нет	нет	да
П. Бертелсен	да	нет	нет	да
У.П. Шульц	нет	нет	нет	нет
CILPE	да	да	да	да

Таблица 2. Классификация специализаторов на базе метода частичных вычислений по поливариантности.

(4) Обладает поливариантностью по переменным (строит одну или несколько различных разметок одной и той же переменной в различных точках программы).
(5) Обладает поливариантностью по инструкциям (строит одну или несколько различных разметок частей метода в зависимости от разметки переменных и стека).
(6) Обладает поливариантностью по методам (для каждого метода строит одну или несколько различных разметок в зависимости от разметки аргументов и результатов метода).
(7) Обладает поливариантностью по классам и массивам (строит одну или несколько различных разметок однотипных переменных).
(8) Обладает расширенной разметкой объектов и массивов: одни поля статических объектов могут иметь статическую разметку, в то время как другие поля — динамическую.

7. Пример применения специализатора CILPE

Следующий пример взят из книги Refactoring to Patterns [23] глава «Replace Implicit Language with Interpreter».

Допустим, что имеется набор объектов таких, что каждый объект обладает каким-нибудь набором свойств. И нужно отбирать объекты по этим свойствам с помощью каких-нибудь критериев.

Например, свойством объекта может быть значение целочисленного поля объекта, и нужно отобрать те объекты, у которых значение этого поля лежит в заданном интервале.

Обычное решение такой проблемы заключается в написании метода, в цикле проверяющего свойства каждого объекта. Но если нужно отбирать объекты по разным логическим условиям, то в программе возникает много методов, тела которых очень похожи: в каждом из методов требуется «прокручивать» цикл по всем объектам.

Структуру программы можно улучшить, вынеся общие части таких методов в новый метод, который принимает на вход предикат, представленный в виде объекта и описывающий логическое условие, по которому следует провести поиск. Таким образом, цикл, проходящий по всем объектам и применяющий предикат к каждому объекту, появляется в теле только одного метода.

Часто бывает нужно не только построить несколько предикатов, но и уметь их комбинировать с помощью логический операций И, ИЛИ, НЕ. Тогда можно реализовать не только объекты-предикаты, но и объекты-операции.

В [23] рассмотрен следующий пример на языке C# [28] (рис. 9, 10). Имеется класс Product для описания продуктов, имеющий два поля-критерия: цвет и цена. И есть абстрактный класс Spec с виртуальным методом-предикатом IsSatisfiedBy(Product) для описания предикатов. Описаны два класса-предиката ColorSpec и BelowPriceSpec — для поиска товара по цвету и по цене. А также два класса-операции AndSpec и NotSpec (для взятия логического И и НЕ для предикатов). Тогда, чтобы найти все товары с цветом, отличным от заданного, и ценой, менее заданной, достаточно создать следующий объект: new AndSpec(new BelowPriceSpec(price), new NotSpec(new ColorSpec(color))).

Такой способ более удобен для чтения, анализа и преобразования программы. Но такая программа может работать менее эффективно, чем вариант с явными проверками полей объектов, поскольку

```
class Product {
  public int color;
  public double price;
  public Product (int color, double price) {
    this.color = color; this.price = price;
} }
abstract class Spec {
  [Inline]
  public Spec () {}
  [Inline]
  public abstract bool IsSatisfiedBy (Product product);
}
class AndSpec : Spec {
  Spec x, y;
  [Inline]
  public AndSpec (Spec x, Spec y)
    {this.x = x; this.y = y;}
  [Inline]
  public override bool IsSatisfiedBy (Product product) {
    return this.x.IsSatisfiedBy(product) &&
           this.y.IsSatisfiedBy(product);
} }
class NotSpec : Spec {
  Spec x;
  [Inline]
  public NotSpec (Spec x) {this.x = x;}
  [Inline]
  public override bool IsSatisfiedBy (Product product) {
    return ! this.x.IsSatisfiedBy(product);
} }
class ColorSpec : Spec {
  int color;
  [Inline]
  public ColorSpec (int color) {this.color = color;}
  [Inline]
  public override bool IsSatisfiedBy (Product product) {
    return product.color == this.color;
} }
```

Рис. 9. Исходная программа (1)

```
class BelowPriceSpec : Spec {
  double price;
  [Inline]
  public BelowPriceSpec (double price)
    {this.price = price;}
  [Inline]
  public override bool IsSatisfiedBy (Product product) {
    return product.price < this.price;
} }
class ProductFinder {
  IEnumerable repository;
  public ProductFinder (IEnumerable repository) {
   this.repository = repository;
  }
  [Inline]
  public IList SelectBy (Spec spec) {
    IList foundProducts = new ArrayList();
    IEnumerator products =
                          this.repository.GetEnumerator();
    while (products.MoveNext()) {
      Product product = (Product) products.Current;
      if (spec.IsSatisfiedBy(product))
        foundProducts.Add(product);
    }
    return foundProducts;
  }
  [Specialize]
  public IList BelowPriceAvoidingAColor
                (double price, int color) {
    Spec spec = new AndSpec(
                        new BelowPriceSpec(price),
                        new NotSpec(new ColorSpec(color)));
    return SelectBy(spec);
} }
```

Рис. 10. Исходная программа (2)

при каждой проверке истинности предиката происходит обход дерева объектов, описывающих предикат. Т.е., вычисление предиката происходит «в режиме интерпретации».

Для получения эффективной программы в данном случае уместно использовать методы специализации. Частичный вычислитель для объектно-ориентированных языков должен уметь специализировать такого рода программы, убирая создание и использование временных объектов, создание и использование которых полностью прослеживается в период специализации.

В данном примере, специализатор CILPE уберает создание «лишних» объектов AndSpec, BelowPriceSpec, NotSpec и ColorSpec, а в том месте, где был вызов метода IsSatisfiedBy, поставит явную проверку полей объекта (рис. 11).

```
class Product {
  public int color;
  public double price;
  public Product (int color, double price) {
    this.color = color; this.price = price;
} }
class ProductFinder {
  IEnumerable repository;
  public ProductFinder (IEnumerable repository)
    {this.repository = repository;}
  public IList BelowPriceAvoidingAColor
                                (double price, int color) {
    IList foundProducts = new ArrayList();
    IEnumerator products =
                          this.repository.GetEnumerator();
    while (products.MoveNext()) {
      Product product = (Product) products.Current;
      if (product.price<price&&!(product.color==color))
        foundProducts.Add(product);
    }
    return foundProducts;
} }
```

Рис. 11. Результат специализации.

Данный пример показывает, что методы специализации в общем и специализатор CILPE в частности позволяют использовать высокоуровневые методы программирования без потери эффективности.

Заключение

В работе рассмотрены существующие в данный момент специализаторы, основанные на методе частичных вычислений и применимые к программам, написанным на императивных и объектно-ориентированных языках, и специализатор CILPE, разработанный автором. Показано, что специализатор CILPE обладает более широкими возможностями, по сравнению с другими известными специализаторами для програм на объектно-ориентированных языка типа C# или Java.

Список литературы

[1] Климов Ю. А. *Возможности специализатора CILPE и примеры его применения к программам на объектно-ориентированных языках* : Препринт : ИПМ им.М.В. Келдыша, 2008, № 30. — 28 с. ↑[], 5, 6.4

[2] Климов Ю. А. *Генератор остаточной программы и корректность специализатора объектно-ориентированного языка* // Научный сервис в сети Интернет: технологии параллельного программирования. Труды Всероссийской научной конференции (18–23 сентября 2006 г., г. Новороссийск). — Москва : Изд-во МГУ, 2006. — ISBN 5–211–05296–X, с. 137–140. ↑1, 4

[3] Климов Ю. А. *Метод частичных вычислений, позволяющий преобразовывать объектно-ориентированные программы в императивные* // Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность. Труды Всероссийской суперкомпьютерной конференции (21–26 сентября 2009 г., г. Новороссийск). — Москва : Изд-во МГУ, 2009. — ISBN 978–5–211–05697–8, с. 241–246. ↑

[4] Климов Ю. А. *О поливариантном анализе времен связывания в специализаторе объектно-ориентированного языка* // Научный сервис в сети Интернет: технологии распределенных вычислений. Труды Всероссийской научной конференции (19–24 сентября 2005 г., г. Новороссийск). — Москва : Изд-во МГУ, 2005. — ISBN 5–211–05141–6, с. 89–91. ↑1, 3

[5] Климов Ю. А. *Особенности применения метода частичных вычислений к специализации программ на объектно-ориентированных языках* : Препринт : ИПМ им. М.В. Келдыша, 2008, № 12. — 27 с. ↑5, 6.4

[6] Климов Ю. А. *Поливариантный анализ времен связывания в специализаторе CILPE для Common Intermediate Language платформы Microsoft.NET* // Технологии Microsoft в теории и практике программирования. Труды Всероссийской конференции студентов, аспирантов и молодых ученых. Центральный регион (Москва, 17–18 февраля 2005 г.). — Москва : Изд-во МГТУ им. Н.Э. Баумана, 2005. — ISBN 5–7038–2668–3, с. 128. ↑1, 3

[7] Климов Ю. А. *Преобразование объектно-ориентированных программ в императивные методом частичных вычислений* // Программные продукты и системы, 2009, № 2 (86), с. 71–74. ↑

[8] Климов Ю. А. *Специализатор CILPE: анализ времен связывания* : Препринт : ИПМ им. М.В. Келдыша, 2009, № 7. — 28 с. ↑1, 3, 6.4

[9] Климов Ю. А. *Специализатор CILPE: генерация остаточной программы* : Препринт : ИПМ им. М.В. Келдыша, 2009, № 8. — 26 с. ↑1, 4, 6.4

[10] Климов Ю. А. *Специализатор CILPE: доказательство корректности* : Препринт : ИПМ им. М.В. Келдыша, 2009, № 33. — 32 с. ↑6.4

[11] Климов Ю. А. *SOOL: объектно-ориентированный стековый язык для формального описания и реализации методов специализации программ* : Препринт : ИПМ им. М.В. Келдыша, 2008, № 44. — 32 с. ↑[], 6.4

[12] Affeldt R., Masuhara H., Sumii E., Yonezawa A. *Supporting objects in runtime bytecode specialization* // Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation : ACM Press, 2002, p. 50–60. ↑6.3.1

[13] Andersen L. O. December 1991. *C program specialization*, Master's thesis, DIKU, University of Copenhagen, Denmark, DIKU Student Project 91–12–17. ↑6.2.1

[14] Andersen L. O. *Partial Evaluation of C* // Chapter 11 in «Partial Evaluation and Automatic Compiler Generation» by N.D. Jones, C.K. Gomard, P. Sestoft ed. C.A.R. Hoare : Prentice-Hall, 1993, p. 229–259. ↑

[15] Andersen L. O. May 1994. *Program Analysis and Specialization for the C Programming Language*, DIKU Technical Report, Computer Science Department, University of Copenhagen, PhD thesis. ↑6.2.1

[16] Asai K., Masuhara H., Yonezawa A. *Partial evaluation of call-by-value lambda-calculus with side-effects* // Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97 ed. Consel C. : ACM Press, 1997, p. 12–21. ↑6.1

[17] Asai K. *Binding-time analysis for both static and dynamic expressions* // Static Analysis Symposium. Lecture Notes in Computer Science ed. Cortesi A., File G. — Berlin–Heidelberg : Springer-Verlag, 1999. Vol. **1694**, p. 117–133. ↑6.1

[18] Asai K. *Offline Partial Evaluation for Shift and Reset* // ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '04), 2004, p. 3–14. ↑6.1

[19] Bertelsen P. *Binding-time analysis for a JVM core language*, 1999, http://www.dina.kvl.dk/~pmb. ↑6.3.2

[20] Chepovsky A. M., Klimov A. V., Klimov A. V., Klimov Yu. A., Mishchenko A. S., Romanenko S. A., Skorobogatov S. Y. *Partial Evaluation for Common Intermediate Language* // Perspectives of Systems Informatics. 5th International Andrei Ershov Memorial Conference, PSI 2003 (Akademgorodok, Novosibirsk, Russia, July 9-12, 2003) : Revised Papers Lecture Notes in Computer Science ed. Broy M., Zamulin A. V. — Berlin–Heidelberg : Springer-Verlag, 2003. Vol. **2890**. — ISBN 978–3–540–20813–6, p. 171–177. ↑[], 6.4

[21] Consel C., Lawall J. L., Le Meur A.–F. *A Tour of Tempo: A Program Specializer for the C Language*, 2003. ↑6.2.2

[22] Jones N. D., Gomard C. K., Sestoft P. Partial Evaluation and Automatic Compiler Generation / ed.C.A.R. Hoare : Prentice-Hall, 1993. ↑[], 1, 6

[23] Kerievsky J. Refactoring to Patterns : Addison Wesley, 2004. ↑7, 7

[24] Klimov Yu. A. *An Approach to Polyvariant Binding Time Analysis for a Stack-Based Language* // First International Workshop on Metacomputation in Russia. Proceedings of the First International Workshop on Metacomputation in Russia

(Pereslavl-Zalessky, Russia, July 2-5, 2008) ed. Nemytykh A. P. — Pereslavl-Zalessky : Ailamazyan University of Pereslavl, 2008. — ISBN 978–5–901795–12–5, p. 78–84. ↑[], 6.4

[25] Masuhara H., Yonezawa A. *Run-time Program Specialization in Java Bytecode* // Proceedings of the JSSST SIGOOC 1999 Workshop on Systems for Programming and Applications (SPA'99), 1999. ↑6.3.1

[26] Schultz U. P. December 2000. *Object-Oriented Software Engineering Using Partial Evaluation*, PhD thesis, University of Rennes I, Rennes, France. ↑6.3.3

[27] Schultz U. P., Lawall J. L., Consel C. *Automatic program specialization for Java* // ACM Transactions on Programming Languagesand Systems, 2003. **25 (4)**, p. 452–499. ↑6.3.3

[28] C# programming language, http://msdn.microsoft.com/vcsharp/. ↑[], 6.4, 7

[29] Common Language Infrastructure (CLI), http://www.ecma-international.org/publications/standards/Ecma-335.htm. ↑6.4

[30] Microsoft .NET Framework, http://www.microsoft.com/net/. ↑[], 6.4

Yu. A. Klimov. *Specializer CILPE: partial evaluator for object-oriented languages.*

Abstract. Program specialization for object-oriented languages by the method of partial evaluation is considered. An overview of the features of known specializes for this class of languages is given. The features of specializer CILPE and an example of it application are discussed.

Key Words and Phrases: program specialization, partial evaluation, object-oriented languages, specializer CILPE.

Образец ссылки на статью:

Ю. А. Климов. *Специализатор CILPE: частичные вычисления для объектно-ориентированных языков* // Программные системы: теория и приложения : электрон. научн. журн. 2010. № 3(3), с. 13–36. URL: http://psta.psiras.ru/read/psta2010_3_13-36.pdf