Трансформация процедурно–параметрических конструкций языка программирования C в промежуточное представление компилятора Clang

Косов П. В., Национальный исследовательский университет «Высшая школа экономики», г. Москва, Россия.

Разработка программ зачастую связана с инкрементальным расширением функциональности. Повышение надежности и качества в этом случае могут быть достигнуты минимизацией изменений в уже написанном коде. Для инструментальной поддержки эволюционной разработки предложен процедурно–параметрический подход к программированию, расширяющий возможности процедурного подхода. Он обеспечивает безболезненное расширение как данных, так функций, используя при этом статическую типизацию. В работе рассматривается реализация поддержки включения процедурно–параметрических механизмов в язык программирования C, широко используемый в системном программировании. Предлагаются решения, ориентированные на эффективную поддержку процедурно–параметрического полиморфизма. К конструкциям, обеспечивающим данный полиморфизм относятся: параметрические обобщения, специализации обобщений, обобщающие функции, обработчики специализаций. Рассматриваются возможности оптимизации дополнительного кода как на стороне компилятора, так и на стороне компоновщика. Описаны ситуации, при которых возможно повышение гибкости процесса разработки и надежности программ за счет использования процедурно–параметрического полиморфизма.

1. Введение

Разработка сложных программных продуктов сопряжена с внесением дополнений и изменений в требования к имеющемуся функционалу, которые приводят к необходимости перерабатывать уже реализованные программные модули. Это ведет к увеличению стоимости поддержки кодовой базы, так как чем больше кода нужно переписать, тем больше вероятность внести ошибки и больше нагрузка на тестировочную инфраструктуру.

Для снижения стоимости внесения изменений используются методы эволюционного расширения функциональности программы, при котором изменения в предварительно написанном коде минимальны. Данный подход реализуется в том числе за счет динамического связывания программных объектов, то есть, когда конкретная реализация вызываемой функции (метода) определяется не во время компиляции, а во время выполнения программы. Этот подход широко применяется при объектно-ориентированном программировании (ООП), что обусловило его высокую степень

применения в промышленных программных комплексах, а также большое распространение реализаций этого подхода в различных языках программирования (C#, Java [2] и т. д.). Стоит отметить, что подобный подход поддерживается не только в объектно-ориентированных (ОО) языках, но также, для обеспечения большей гибкости, и в мультипарадигменных и процедурных языках (C++ [1], Rust [4], Go [3]). Однако, этот подход тоже обладает некоторыми недостатками, например, в вышеперечисленных реализациях нет прямой поддержки множественного полиморфизма, которую приходится реализовать разработчикам, внедряя новые компоненты (применяя шаблоны проектирования), не связанные непосредственно с решением бизнес задач, что, в свою очередь, опять же ведет к увеличению объема кода и, как следствие, увеличению стоимости поддержки и тестирования.

Поддержка множественного полиморфизма решается в нескольких языках программирования (Julia [10], CLOS [12], и т. д.). Однако из-за особенностей реализации производительность таких решений не позволяет широкого применения множественного полиморфизма [11], в результате чего задачи, которые непосредственно решались с помощью множественного полиморфизма, решаются с помощью диспетчеризации (напр. ОО паттерн проектирования Посетитель [7]), словарей (напр. таблиц с хешированием) или реализацией вложенных проверок (напр. использование вложенных операторов switch в C/C++). Все эти перечисленные решения связаны с добавлением дополнительного кода, не связанного напрямую с решением задачи, и соответственно, увеличивается стоимость поддержки и расширения функциональности такого кода, как и вероятность внесения ошибок.

Другое решение реализации полиморфизма было предложено в процедурно-параметрическом (ПП) подходе [14]. Оно расширяет процедурное программирования возможностью добавления как новых данных, таки и обрабатывающих их функций без изменения ранее написанного кода. Данный подход был первоначально использован при разработке статически типизированного языка программирования O2M (являющегося расширением языка Oberon-2) [15]. Проведенные с его использованием исследования показали, что ПП парадигма программирования обеспечивает гибкое расширение функциональности программ и может быть интегрирована как в процедурные, так и в функциональные языки программирования[17]. В работе [16] было сделано сравнение ПП с другими подходами к реализации динамического полиморфизма.

В следующей работе [5] была рассмотрена интеграция ПП конструкций в язык программирования C, который является одним из распространенных языков программирования (напр. [18]). Он широко используется в областях, где накладные расходы от применения

ООП слишком высоки. Также показано, что использование ППП позволяет достичь большей безопасности кода и его гибкости, без негативного влияния на скорость выполнения за счет эффективной реализации множественного полиморфизма.

В данной работе рассматриваются детали реализации ПП конструкций в языке программирования C, за счет которой достигается высокая эффективность множественного полиморфизма (поиск необходимого обработчика происходит за константное время и не зависит от количества специализаций). Приводится описание трансформаций абстрактно-синтаксического дерева, внедрение дополнительных конструкций на этапе кодогенерации и добавление необходимой поддержки во время выполнения программы. В работе рассматривается реализация ПП конструкций в языке программирования C с использованием компилятора clang [6], исходный код проекта доступен на сайте github [8], примеры программ, использующих ПП конструкции языка C также представлены на сайте github [9].

2. Трансформация конструкций, поддерживающих процедурно–параметрическую парадигму

2.1 Общее описание процедурно-параметрических конструкций

В ходе компиляции кода, обеспечивающего поддержку процедурно–параметрической парадигмы (ППП), необходимо обеспечить трансформацию следующих программных объектов, формирующих ППП. (подробное описание каждой конструкции дано в работе [5]):

- Основа специализации. Данной конструкцией может быть любая структура языка C (либо одна из конструкций, о которых речь пойдет ниже: параметрическое обобщение или специализация обобщения);
- Параметрическое обобщение — структура языка C с дополнительной возможностью подключения специализаций;
- Специализация параметрического обобщения (для краткости — специализация обобщения) — параметрическое обобщение с подключенной специализацией;
- Экземпляр параметрического обобщения или специализированная переменная — объект типа параметрического обобщения или специализации обобщения;
- Обобщающая функция — функция, принимающая параметрические обобщениями (т. е. если она принимает один и более параметр типа параметрического обобщения);
- Обработчик параметрической специализации (или специализированная функция) — специализация обобщающей

функции, в которой параметрические обобщения заменены на специализации данного обобщения;

- Вызов обобщающей функции — передача в обобщающую функцию указателей на специализированные переменные. В данном случае обобщающая функция является диспетчеризующей — по типовым признакам специализированных переменных она определяет необходимый обработчик параметрической специализации и вызывает его, передав туда необходимые аргументы.

В последующих пунктах данной главы эти конструкции будут рассмотрены более полно.

Преобразование процедурно-параметрической программы в программу на стандартном языке С происходит на этапе формирования абстрактно-синтаксического дерева (АСД). А во время генерации промежуточного представления создаются необходимые данные и функции для обеспечения множественного полиморфизма. Основная идея трансформации АСД заключается в замене процедурно-параметрических конструкций, представленных на уровне исходных текстов, конструкциями АСД, представляющими эквивалентные подстановки конструкций языка С. Модифицированное дерево используется для последующей трансформации из С в LLVM. Для наглядности проводимых преобразований формируемые в АСД конструкции в тексте представлены на языке С. Примеры кода, порождаемые из АСД, а также напрямую формируемый дополнительный код, необходимый для поддержки процедурно-параметрической полиморфизма, представлены в с использованием упрощенного байткода LLVM IR.

В работе рассматривается реализация, при которой формирование необходимых процедурно-параметрических отношений осуществляется на этапе начальной загрузки программы и подключения системных библиотек. Представленное решение функционирует под управлением операционной системы Linux. Для инициализации и поддержки ПП полиморфизма используется функционал глобальных конструкторов и деструкторов (секций .init .fini в формате ELF) [13]. Возможны и другие варианты реализации, например, формирование необходимых параметрических отношений на этапе компоновки программы, что предполагается рассмотреть в ходе дальнейших исследований.

2.2Использование основ специализаций

Под основой специализации понимается любой атомарный (базовый) или составной именованный тип (структура), включая и параметрическое обобщение, которое тоже является составным типом. Понятие основы специализации не накладывает ограничений

на самостоятельное использование этих типов в программе и вводится только для терминологической стыковки с другими понятиями. В качестве примеров структур, используемых далее в роли основ специализаций, можно привести прямоугольник, треугольник, и круг:

```
typedef struct Rectangle {int x, y;} Rectangle;
typedef struct Triangle {int a, b, c;} Triangle;
typedef struct Circle {int r;} Circle;
```

Использование **typedef** не является обязательным и введено для более компактного представления в ходе дальнейшего изложения.

Так как в качестве основ специализаций используются стандартные типы данных языка С, то на во время компиляции не требуется их дополнительная обработка.

2.3 Трансформация параметрических обобщений

Параметрическое обобщение (далее — обобщение) первоначально может содержать от нуля до нескольких специализаций. Оно может расширяться новыми специализациями в различных единицах компиляции без изменения ранее написанного кода, что отличает его от объединения языка С. В предлагаемом расширении языка С параметрическое обобщение является модификацией структуры и называется также обобщенной структурой [5].

Базовой конструкцией, на которой можно рассмотреть особенности преобразования, является обобщение без структурных и обобщенных данных:

```
// Базовое (пустое) обобщение
struct Figure {}<>;
```

Данная конструкция преобразуется в структуру абстрактного синтаксического дерево, дополняясь полем целого типа, которое хранит признак специализации. На языке С эту конструкцию АСД можно представить следующим эквивалентом:

```
struct Figure { int tag; };
```

При трансформации в промежуточное представление LLVM данный тип будет представлен следующим образом:

```
%struct.Figure = type { i32 }
```

Помимо формирования обобщения, в единицу компиляции на этапе генерации LLVM добавляются следующие вспомогательные элементы:

1) Глобальная переменная, содержащая количество зарегистрированных специализаций данного обобщения. Изначально эта переменная инициализируется нулевым значением:

```
@tags_Figure = global i32 0
```

2) Вспомогательная функция увеличения значения глобальной переменной `tags_Figure`

```
define void @inc_tags_Figure() {
entry:
  %0 = load i32, ptr @tags_Figure
  %inc = add i32 %0, 1
  store i32 %inc, ptr @tags_Figure
  ret void
}
```

На языке программирования C аналог данной функции можно описать следующим образом:

```
void inc_tags_Figure() {
  tags_Figure++;
}
```

2.4 Трансформация специализаций параметрических обобщений

Под специализацией параметрического обобщения (далее — специализацией) понимается любая из основ специализации, включенная в качестве составной части в параметрическое обобщение. Специализация задает одну из альтернатив обобщения. Она близка по семантике альтернативному полю объединения (union) языка программирования C. Помимо этого в качестве специализаций могут использоваться неименованные структурные типы и указатели. Специализацию можно рассматривать как подтип параметрического обобщения.

Добавление специализаций в соответствующий список обобщения приводит к созданию в АСД структур, представляющих данные специализации. Каждая из этих структур включает в себя два поля: структуру, определяющую обобщение и структуру, задающую основу специализации.

Например, при наличии основ специализаций `Circle` и `Rectangle` обобщение `Figure` может быть расширено специализациями следующим образом:

```
struct Figure { }< circ: Circle; rect: Rectangle; >;
```

В ходе анализа в АСД сформируются две дополнительные структуры, в которых обобщение, определяющее фигуру, будет первым полем. Второе поле в каждой из этих структур будет задавать соответствующую специализацию. Эквивалент сформированных на языке C выглядит следующим образом:

```
// Представление обобщения struct Figure
struct Figure { int tag; };

// Представление специализации Figure<circ: Circle>
```

```
struct Figure_circ {
    struct Figure head;
    struct Circle tail;
};

// Представление специализации Figure<rect: Rectangle>
struct Figure_rect {
    struct Figure    head;
    struct Rectangle tail;
};
```

После генерации в LLVM данные структуры будут представлены следующими типами в промежуточном представлении компилятора:

```
%struct.Figure = type { i32 }
%struct.Figure_circ =   type {
                            struct.Figure,
                            struct.Circle
                        }

%struct.Figure_rect =   type {
                            struct.Figure,
                            struct.Rectangle
                        }
```

Для поддержки параметрического полиморфизма на этапе генерации LLVM дополнительно осуществляется три этапа генерации вспомогательных элементов и дополнительного кода:

1) Формирование признаков конкретных специализаций, которое задается через глобальные переменные. Для вышеописанной структуры `Figure_circ` данный признак будет выглядеть следующим образом:

```
@tag_Figure_circ = global i32 0
```

2) Для инициализации признаков конкретных типов формируются соответствующие функции в LLVM IR. Например, для вышеописанной структуры `Figure_circ` данный инициализатор будет выглядеть следующим образом:

```
define void @init_Figure_circ() {
  call void @inc_tags_Figure()
  %1 = load i32, ptr @tags_Figure
  store i32 %1, ptr @tag_Figure_circ
  ret void
}
```

На языке программирования C аналогичная функция может быть представлена в виде

```
void init_Figure_circ() {
  inc_tags_Figure();
  tag_Figure_circ = tags_Figure;
}
```

3) На следующем этапе кодогенерации инициализаторы добавляются в список функций, выполняющихся до запуска функции **main**. Это необходимо для обеспечения корректной инициализации значений признаков перед стартом приложения

```
@llvm.global_ctors = appending global
   [2 x { i32, ptr, ptr }]
   [{i32 101, ptr @init_Figure_circ, ptr null},
    {i32 101, ptr @init_Figure_rect, ptr null}]
```

2.4 Создание объектов параметрических специализаций

Объекты параметрических специализаций являются переменными специализированного типа, то есть, специализированными переменными. Выделение памяти для них осуществляется по тем же принципам, что и для других переменных языка С:

- автоматическое выделение памяти
- выделение глобальной и статической памяти
- выделение динамической памяти

Но дополнительно к выделению памяти, для специализированных переменных также происходит инициализация признаков типов.

2.4.1 Формирование объектов в автоматической памяти

При создании специализированной переменной в автоматической памяти (на стеке) происходит инициализация признака создаваемого объекта. В качестве примера можно рассмотреть следующую функцию, в которой создается локальная переменная специализированного типа:

```
void foo() {
    Figure.circ fc;
}
```

в этом случае на этапе генерации LLVM IR добавляется код, который инициализирует признак типа соответствующим значением:

```
define void @foo() {
   ; Выделение памяти под структуру
   %fc = alloca %struct.Figure_circ

   ; Начало блока инициализация признаками
```

```
  %0 = load i32, ptr @tag_Figure_circ
  %head = getelementptr inbounds %struct.Figure_circ,
          ptr %fc, i32 0, i32 0
  %spec_tag = getelementptr inbounds %struct.Figure,
              ptr %head, i32 0, i32 0
  store i32 %0, ptr %spec_tag
  ; Конец блока инициализации признаками

  ret void
}
```

На языке программирования С преобразованную функцию можно представить в следующем виде:

```
void foo() {
  struct Figure_circ fc;
  fc.head.tag = tag_Figure_circ;
}
```

2.4.2 Формирование объектов в динамической памяти

Для создании специализированной переменной в динамической памяти (на куче) предложена специальная функция **create_spec**, встроенная в компилятор. Пример ее использования:

```
void bar() {
    struct Figure.rect *pfr = create_spec(Figure.rect);
}
```

В этом случае компилятор на этапе формирования АСД сформирует вызов функции **create_spec_Fugure_rect**, тело которой будет сформировано на этапе генерации LLVM IR. Данная функция выделит память под объект типа **struct Figure.rect**, проинициализирует признак специализации и вернет указатель на данный объект. Выделение памяти происходит с помощью вызова функции **malloc**, поэтому освобождение памяти следует производить с помощью вызова функции **free**. Определение сгенерированной компилятором функции **create_spec_Fugure_rect** выглядит следующим образом:

```
define ptr @create_spec_Figure_rect() {
  %Size = alloca i64
  store i64 8, ptr %Size
  %0 = load i64, ptr %Size
  %malloc_res = call ptr @malloc(i64 noundef %0)
  call void @init_spec_Figure_rect(ptr noundef %malloc_res)
  ret ptr %malloc_res
```

```
}
```

Аналог данной функции на языке программирования C:

```
void* create_spec_Figure_rect() {
  void* malloc_res = malloc(sizeof(struct Figure_rect));
  init_spec_Figure_rect(malloc_res); // Описание дано ниже
  return malloc_res;
}
```

В данном коде происходит вызов функции `init_spec_Figure_rect`, которая также генерируется компилятором для инициализации признака данной специализации. Определение этой функции выглядит следующим образом:

```
define void @init_spec_Figure_rect(ptr noundef %0) {
   %pp_head = getelementptr inbounds
              %struct.Figure_rect, ptr %0, i32 0, i32 0
   %pp_spec_type = getelementptr inbounds
              %struct.Figure, ptr %pp_head, i32 0, i32 0
   %global_spec_tag = load i32, ptr @tag_Figure_rect
   store i32 %global_spec_tag, ptr %pp_spec_type
   ret void
}
```

Аналог данной функции на языке программирования C:

```
void init_spec_Figure_rect(struct Figure_rect* pfr) {
  pfr->head.tag = tag_Figure_rect;
}
```

Эта функция доступна разработчику, что может быть полезно, если программа работает с уже выделенной памятью, в которой нужно создать объекты параметрических специализаций. Например:

```
void foo() {
     struct Figure.rect *array_pfr =
               malloc(sizeof(struct Figure.rect) * ELEM_NUM);
     // Явная инициализация указанной области памяти
     // (в данном примере — только первого элемента массива)
    init_spec(Figure.rect, array_pfr);
}
```

2.4.3 Формирование объектов в глобальной памяти

Для создании специализированной переменной в глобальной (статической) памяти компилятором генерируются дополнительные

конструкторы (аналогично механизму инициализации глобальных объектов в C++). Например, имеется глобальная переменная вида:

```
struct Figure.rect gfr;
```

В этом случае компилятор на этапе формирования кодогенерации добавляет следующую функцию инициализации:

```
void init_gfr() {
   init_spec(Figure.rect, &gfr);
}
```

И затем идет добавление указателя на эту функцию в массив конструкторов (приоритет данной функции устанавливается с учетом того, что она должна выполняться после выполнения инициализаторов признаков типов)

```
@llvm.global_ctors = appending global
                     [3 x { i32, ptr, ptr }]

; Сначала инициализируются значения признаков специализаций
[{i32 101, ptr @init_Figure_circ, ptr null },
 {i32 101, ptr @init_Figure_rect, ptr null },

; После этого инициализация признаков
; глобальных переменных
 {i32 102, ptr @init_gfr, ptr null}]
```

2.5 Трансформация обобщающих функций

Обобщающие функции принимают непустой список параметрических обобщений (т. е. количество обобщений больше или равно единице) и используют признаки специализаций данных обобщений для выбора соответствующих обработчиков (инициализация самих признаков описана в п.2.4).

Синтаксис обобщающих функций и обработчиков специализаций подробно рассмотрен в работе [5].

В качестве примера рассмотрим добавление обобщающей функции без добавления обработчиков специализаций:

```
void PrintFigure<Figure* f>() {
   printf("Default case\n");
}
```

Данная функция принимает указатель на обобщение и для любых специализаций выводит текст

`"Default case\n"`. В АСД такая конструкция преобразуется в функцию с именем `pp_mm_PrintFigure`. На этапе генерации в LLVM осуществляются трансформации:

1) Добавление глобальной переменной — указателя на массив указателей на обработчики специализаций `initarr_pp_mm_PrintFigure`

2) Перемещение инструкций функции `pp_mm_PrintFigure` в специально созданную функцию `default_pp_mm_PrintFigure`, которая будет выполнять роль обработчика по умолчанию.

3) Добавление диспетчеризующего механизма в `pp_mm_PrintFigure`, который обеспечит вызов необходимого обработчика специализаций в зависимости от типа переданной в функцию специализации.

Пример сгенерированной диспетчеризующей функции:

```
define void @pp_mm_PrintFigure(ptr %f1, ptr %f2) {
entry:
  %0 = alloca ptr, align 8
  store ptr %f1, ptr %0, align 8
  %1 = getelementptr inbounds %struct.Figure,
                          ptr %f1, i32 0, i32 0
  %2 = load i32, ptr %1
  %3 = sub nsw i32 %2, 1
  %4 = sext i32 %3 to i64
  %5 = load ptr, ptr @mminitarr_pp_mm_PrintFigure
  %6 = getelementptr inbounds ptr, ptr %5, i64 %4
 %7 = load ptr, ptr %6
  call void %7(ptr noundef %f1)
  ret void
}
```

Пример сгенерированного обработчика по умолчанию:

```
define void @default_pp_mm_PrintFigure(ptr %f) {
entry:
  %call = call i32 (ptr, ...) @printf(ptr @str_default)
  ret void
}
```

Аналог данных функций на языке программирования C может быть представлен следующим образом:

```
void pp_mm_PrintFigure(struct Figure* f) {
   // Вычисление индекса соответвующего обработчика
   int idx = f->head.tag - 1;
   // Вызов обработчика
   (*initarr_pp_mm_PrintFigure)[idx](f1);
```

```
}

void default_pp_mm_PrintFigure(struct Figure* f) {
  printf("Default case\n");
}
```

На этапе кодогенерации также добавляется функция для выделения памяти под используемый массив обработчиков специализаций (в примере это `initarr_pp_mm_PrintFigure`). Эта функция заполняет все элементы массива указателем на обработчик по умолчанию (в примере это `default_pp_mm_PrintFigure`). Следует отметить, что в случае множественного полиморфизма, когда количество обобщенных аргументов больше одного, массив обработчиков обрабатывается как многомерная матрица, которая в памяти представляется как одномерный массив. За счет этого достигается постоянная сложность доступа к необходимому обработчику специализаций.

Для приведенного выше примера функция выделения памяти будет выглядеть следующим образом:

```
define void @alloc_pp_mm_PrintFigure() {
entry:
 %0 = load i32, ptr @tags_Figure
  %1 = sext i32 %0 to i64
  %2 = mul i64 8, %1
  %call_malloc = call ptr @malloc(i64 noundef %2) #2
  store ptr %call_malloc, ptr @initarr_pp_mm_PrintFigure
  %3 = load i64, ptr @initarr_pp_mm_PrintFigure
  %Size = alloca i64
  %Iter = alloca i64
  %4 = udiv i64 %2, 8
  %5 = load i64, ptr @initarr_pp_mm_PrintFigure
  store i64 %4, ptr %Size
  store i64 0, ptr %Iter
  br label %for.cond

for.cond:
  %6 = load i64, ptr %Iter
  %7 = load i64, ptr %Size
  %8 = icmp ult i64 %6, %7
  br i1 %8, label %for.body, label %for.end

for.body:
  %9 = load i64, ptr %Iter
  %10 = load ptr, ptr @initarr_pp_mm_PrintFigure
  %11 = getelementptr inbounds ptr, ptr %10, i64 %9
  %12 = load i64, ptr %11
  store ptr @default_pp_mm_PrintFigure, ptr %11
  %13 = load i64, ptr %11
  br label %for.inc
```

```
for.inc:
  %14 = load i64, ptr %Iter
  %15 = add i64 %14, 1
  store i64 %15, ptr %Iter
  br label %for.cond

for.end:
  %16 = load i64, ptr @initarr_pp_mm_PrintFigure
  ret void
}
```

На языке программирования C аналог этой функции можно представить следующим образом:

```
void alloc_pp_mm_PrintFigure() {
   // Получение количества элементов
   // (в данном случае это просто значение
   //  переменной  tags_Figure,
   //  а в случае нескольких обобщений должно быть вычисленно)
   int elems_count = tags_Figure;
   // Выделенение памяти
   initarr_pp_mm_PrintFigure =
                malloc(sizeof(void*) * elems_count);
   // Заполнение массива указателей обработчиком по умолчанию
   for (int i = 0; i < elems_count; ++i) {
      initarr_pp_mm_PrintFigure[i] =
                default_pp_mm_PrintFigure;
   }
}
```

Данная функция должна быть вызвана до начала выполнения функции **main**, но после инициализации признаков всех специализаций. Для этого используется помещение указателя на эту функцию в массив конструкторов с соответствующим приоритетом:

```
@llvm.global_ctors = appending global
                     [3 x { i32, ptr, ptr }]

; Сначала инициализируются значения признаков специализаций
[{i32 101, ptr @init_Figure_circ, ptr null },
 {i32 101, ptr @init_Figure_rect, ptr null },

; После этого следует выделение памяти
; под массив обработчиков
 {i32 102, ptr @alloc_pp_mm_PrintFigure, ptr null}]
```

2.6 Трансформация обработчиков специализации

Обработчик специализации с точки зрения разработчика — это обобщающая функция, принимающая указатель на специализированную переменную. После того как данный обработчик

определен пользователем (разработчиком), компилятор должен обеспечить сохранение указателя на данный обработчик в соответствующую ячейку массива обработчиков обобщающей функции. Это достигается следующим: для каждого обработчика специализации генерируется функция регистрации этого обработчика:

а) Шаг 1. Компилятор встречает описание обработчика специализации в коде. В этом случае имя этого обработчика в АСД меняется на `pp_mm_PrintFigure_Figure_circ__Figure_rect` (аналогичный подход используется в компиляторах C++ для именования перегрузок функций):

```
void PrintFigure<Figure.rect* f>() {
    printf("circ - rect case\n");
}
```

б) Шаг 2. Формируется функция регистрации обработчика специализации (которая запишет указатель на него в нужную ячеку памяти):

```
define void @record_pp_mm_PrintFigure_Figure_rect() {
entry:
  %0 = load i32, ptr @tag_Figure_rect
  %1 = sub nsw i32 %0, 1
  %2 = sext i32 %1 to i64
  %3 = load ptr, ptr @initarr_pp_mm_PrintFigure
  %4 = getelementptr inbounds ptr, ptr %3, i64 %2
  store ptr @pp_mm_PrintFigure_Figure_rect, ptr %4
  ret void
}
```

Аналогичная функция на языке программирования C может быть представлена в следующем виде:

```
void record_pp_mm_PrintFigure_Figure_rect() {
   // Вычисление индекса соответвующего обработчика
   // (в данном случае это значение tag_Figure_rect - 1,
   //  а в случае нескольких обобщений должно быть вычисленно)
   int idx = tag_Figure_rect - 1;
   // Запись обработчика
   initarr_pp_mm_PrintFigure[idx] =
     pp_mm_PrintFigure_Figure_rect;
}
```

в) Шаг 3. Сформированная функция регистрации должна быть выполнена раньше начала функции `main`, но после выделения памяти под массив обработчиков. Для этого функцию регистрации помещается в массив конструкторов с соответствующим приоритетом:

```
@llvm.global_ctors = appending global
```

```
                        [4 x { i32, ptr, ptr }]
[{i32 101, ptr @init_Figure_circ, ptr null },
 {i32 101, ptr @init_Figure_rect, ptr null },
 {i32 102, ptr @alloc_pp_mm_PrintFigure, ptr null},

; Запись обработчиков происходит после выделения памяти
 {i32 103, ptr @record_pp_mm_PrintFigure_Figure_rect,
  ptr null }]
```

3. Особенности процесса компоновки и запуска программы

После кодогенерации в процесс компиляции дополнительные изменения не вносятся. В итоге для каждой единицы трансляции генерируются соответствующие стандартные объектные файлы. После этого сформированные объектные файлы передаются стандартному компоновщику (в случае с компилятором clang и LLVM это LLD), который обеспечивает формирование одной секции для глобальных конструкторов и одной для глобальных деструкторов с учетом установленных приоритетов. Функции с одним приоритетом выполняются в условно-произвольном порядке. Это обеспечивает выполнение сгенерированных инициализаторов перед началом функции `main` в следующем порядке:

- Сначала инициализируются значения признаков специализаций.
- После этого происходит инициализация признаков в глобальных специализированных переменных.
- Затем следует выделение памяти под массив указателей на обработчики специализаций и заполнение этого массива значением указателя обработчика по умолчанию (либо определенного пользователем, либо добавленного компилятором обработчика, осуществляющего аварийное прерывание программы).

Данные инициализаторы могут быть оптимизированны во время компоновки программы (т. е. компоновщик может выполнить заложенный в них функционал и удалить их из списка конструкторов) и не выполняться перед стартом программы. Возможные оптимизации ПП конструкций на стороне компилятора и компоновщика являются объектом дальнейших исследований и будут представлены в следующих работах.

При использовании динамических библиотек конструкторы и деструкторы также выполняются в необходимом порядке благодаря соответствующей поддержки со стороны динамического загрузчика операционной системы. То есть, динамический загрузчик составляет граф зависимостей запускаемого процесса, формирует список необходимых конструкторов со всех зависимостей и запускает их в

определенной приоритетами последовательности. Таким образом, поддержка ПП функциональности при компоновке с использованием динамических библиотек не требует дополнительных изменений.

Если же динамическая библиотека с ПП конструкциями не является явной зависимостью приложения, а загружается с помощью функций dlopen (и выгружается с помощью функции dlclose), то в этом случае вызов необходимых инициализирующих функций происходит с помощью дополнительно сгенерированного кода, описание которого выходит за рамки данной статьи и будет представлено в дальнейших работах.

При необходимости добавить новую специализацию и (или) обработчик специализации можно обойтись без изменения предварительно написанного кода: нужно создать новую единицу трансляции и скомпоновать версию программы, используя объектный файл с описанием требуемой новой функциональности. Таким образом, изменения в уже существующие единицы трансляции вносить не нужно. Более того, даже объектные файлы существующих единиц трансляции не требуют перекомпиляции. Данная особенность делает ПП подход довольно гибким к расширению функциональности и открывает дополнительные возможности использования по сравнению с ООП и другими парадигмами (например, для применения на мобильных платформах).

4. Заключение

В результате данной работы описана внутренняя реализация механизма ПП полиморфизма для языке программирования C в компиляторе clang. Показаны возможности, позволяющие сделать расширение функционала более гибким по сравнению с другими подходами (в частности с ООП).

Дополнительно следует указать, что данный механизм может быть реализован различными способами. Например, вместо представленного в п.2.5 индескируемого массива обработчиков специализаций можно использовать такие решения, как контейнеры с использованием пар ключ-значение, условные переключатели и др. Выбор используемого решения может быть определен в зависимости от доли зарегистрированных обработчиков специализаций, количества возможных обработчиков, требований по размеру кода и производительности, либо от других условий, в том числе решений в дизайне языка программирования, в котором реализуется ПП полиморфизм. Это также возможное направление для дальнейших исследований.

Также в ходе дальнейших исследований предполагается более детально рассмотреть вопросы, связанные с оптимизаций ПП конструкций, что позволит генерировать более производительный

код, по сравнению с использованием прямой реализации методов ПП полиморфизма. Это возможно за счет того, что при знании семантики ПП конструкций компилятор и компоновщик имеют больше возможностей для различных дополнительных трансформаций, таких как, например, встраивание кода обработчика специализаций в место вызова диспетчеризующей функции; инициализация поля признака типа специализированной переменной соответствующим значением (которое может быть вычислено на этапе компоновки программы); вычисление необходимого размера для массива обработчиков и выделение памяти для него на этапе компоновки программы, и другие оптимизационные решения.

Литература

1. M. Gregoire, Professional C++, John Wiley & Sons. 2018, p. 1122.
2. E. Sciore, Java Program Design, Apress Media. 2019, p. 1122.
3. A. Freeman, Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang, Apress. 2022, p. 1105.
4. J. Blandy, J. Orendorff, and L. F. Tindall, Programming Rust, O'Reilly Media. 2021, p. 735.
5. Легалов А.И., Косов П.В. Расширение языка С для поддержки процедурно–параметрического полиморфизма. Моделирование и анализ информационных систем. 2023;30(1):40-62. doi: 10.18255/1818-1015-2023-1-40-62.
6. Clang: A c language family frontend for llvm. [Online]. Available: https://clang.llvm.org/, accessed 24.01.2025.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Professional. 1994, p. 416.
8. Размещение проекта с процедурно–параметрической версией языка программирования С на Гитхаб: https://github.com/kpdev/llvm-project/tree/pp-extension-v2, accessed 24.01.2025.
9. Примеры на Гитхаб, написанные с использованием процедурно–параметрической версии языка С: https://github.com/kreofil/evo-situations, accessed 24.01.2025.
10. Julia Documentation [Online]. Available: https://docs.julialang.org/en/v1/, accessed 24.01.2025.
11. Julia Documentation. The dangers of abusing multiple dispatch [Online]. Available: https://docs.julialang.org/en/v1/manual/performance-tips/#The-dangers-of-abusing-multiple-dispatch-(aka,-more-on-types-with-values-as-parameters), accessed 24.01.2025.
12. The Common Lisp Cookbook — Fundamental of CLOS [Online]. Available: https://lispcookbook.github.io/cl-cookbook/clos.html, accessed 24.01.2025.
13. Tool Interface Standard (TIS). Executable and Linking Format (ELF) Specification [Online]. Available: https://refspecs.linuxfoundation.org/elf/elf.pdf, accessed 24.01.2025.
14. Легалов А.И. Процедурно–параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? - Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. - 43 с. Доступна в сети Интернет: http://www.softcraft.ru/ppp/pppfirst/, accessed 24.01.2025.

15. Легалов А.И. Швец Д.А. Процедурный язык с поддержкой эволюционного проектирования. --- Научный вестник НГТУ, № 2 (15), 2003. С. 25-38.
16. Легалов А.И., Косов П.В. Эволюционное расширение программ с использованием процедурно–параметрического подхода // Вычислительные технологии. 2016. Т. 21. № 3. С. 56–69.
17. Легалов И.А. Применение обобщенных записей в процедурно–параметрическом языке программирования // Научный вестник НГТУ, 2007. № 3 (28). С. 25–38.
18. TIOBE Index Programming Languages [Online]. Available: https://www.tiobe.com/tiobe-index/ , accessed 24.01.2025.

References in Cyrillics

1. Primery' na Gihab, napisanny`e s ispolzovaniem procedurno-parametricheskoy versii yazy`ka C: https://github.com/kreofil/evo-situations, accessed 24.01.2025.
2. Legalov A.I Procedurno-parametricheskaya paradigma programmirovaniya. Vozmozhna li al'ternativa ob'ectno-orientirovannomu stilyu? Krasnoyarsk: 2000. Del. Ruk. №662-V00 Dep. v VINITI 13.03.2000. - 43 s. [Online]: http://www.softcraft.ru/ppp/pppfirst/, accessed 24.01.2025.
3. Legalov A.I, Shvets D. A. Procedurny'y yazy'k s podderzhkoy e'volyucionnogo programmirovaniya. --- Nauchny'y vestnik NGTU, № 2 (15), 2003. S. 25-38.
4. Legalov A.I, Kosov P.V. E'volyutsionnoe rasshirenie programm s ispol'zovaniem procedurno-parametricheskogo podhoda // Vy'chislitel'ny'e tehnologii. 2016. T. 21. № 3. S. 56–69.
5. Legalov I.A. Primenenie obobschenny'h zapisey v procedurno-parametricheskom yazy'ke programmirovaniya // Nauchny'y vestnik NGTU, 2007. № 3 (28). S. 25–38.

Ключевые слова

язык программирования; компиляция; процедурно–параметрическое программирование; полиморфизм; эволюционная разработка программного обеспечения; надежность программного обеспечения.

Косов Павел Владимирович
Национальный исследовательский университет
«Высшая школа экономики»,
Россия, 101000, г. Москва, ул. Мясницкая, д. 20.
ORCID: 0000-0002-9035-312X,
pvkosov@hse.ru

Pavel Kosov, Transformation of procedural-parametric constructions of C programming language to intermediate representation of Clang

DOI: TBD

Keywords

programming language; compilation; procedural-parametric programming; polymorphism; evolutionary software development; software reliability.

Abstract

Software development is often associated with incremental expansion of functionality. In this case, reliability and quality can be improved by minimizing changes in the already written code. A procedural-parametric programming paradigm is proposed for instrumental support of evolutionary development. It expands the capabilities of the procedural approach and ensures flexible expansion of both data and functions, using static typing instead of type dereferencing. The paper considers the inclusion of procedural-parametric mechanisms in the C programming language, which is widely used in system programming. Syntactic constructions aimed at supporting the procedural-parametric paradigm are proposed. These include: parametric generalizations, specializations of generalizations, generalizing functions, specialization handlers. Their capabilities are considered. Situations are described in which it is possible to increase the flexibility of the development process and the reliability of programs through the use of procedural-parametric polymorphism.

Информация об авторе / Information about author

Павел Владимирович КОСОВ – Руководитель группы разработки компиляторов, аспирант факультета компьютерных наук НИУ «Высшая школа экономики». Его научные интересы связаны с разработкой языков программирования и компиляторов, оптимизацией LLVM, эволюционной разработкой программного обеспечения.

Pavel Vladimirovich KOSOV is a technical leader of compiler development team and a graduate student of the Faculty of Computer Science at the Higher School of Economics. His research interests are related to the development of programming languages and compilers, LLVM optimization, and evolutionary software development.