Конструктор компиляторов LiME

М.О. Бахтерев

12 ноября 2014 г.

1. Общее описание

Ниже приведено общее описание системы и указаны мотивы для того, почему конструкция именно такая.

1.1. Ещё один компилятор

Попытки использовать существующие системы компиляции промышленного уровня для разработки **оптимизирующего** компилятора для процессоров MultiClet (семейство MCp) к успеху не привели. Эти средства слишком тесно привязаны к ISA процессоров, с основным обменом данными через регистры. Были испытаны:

- GCC;
- LCC, который можно приспособить, теряя в эффективости;
- TCC;
- LLVM, в перенацеливание которого было вложено особенно много сил.

1.1.1. Проблемы. Коммутатор

Одно из основных предположений этих традиционных систем о целевых процессорах:

- у любой ячейки памяти есть фиксированный адрес или имя R;
- ячейка с именем R хранит записанное в него значение до следующей записи;
- все ссылки в разных инструкциях на значения для чтения по имени R выдают одинаковые результаты между записями.

Ссылки вида `@N`, используемые в МСр для обращения к результатам выполнения предыдущих инструкций, таким свойством не обладают. В инструкциях

```
add     @1, @1
add     @1, @1
```

`@1` ссылаются на разные значения.

1.1.2. Проблемы. Параграфы

Другое предположение этих систем заключается в том, что обмен данными между процессами исполнения различных инструкций осуществляется *только через именованные ячейки памяти*

- и в линейных участках программы,
- и при исполнении процессором переходов между ними.

Для процессоров МСр это не выполняется. Есть два способа передавать данные между процессами исполнения инструкций:

- именованные ячейки памяти;
- коммутатор.

В рамках одного параграфа МСр, описывающего, обычно, линейный участок программы, обмены между исполнениями инструкций возможны *только через коммутатор*.

1.1.3. Проблемы. Типичный LLVM

```
  %wide.load.1 = load <4 x i32>* %28, align 4, !tbaa !1
  %.sum12.1 = or i64 %index.next, 4
  %29 = getelementptr i32* %A, i64 %.sum12.1
  %30 = bitcast i32* %29 to <4 x i32>*
  %wide.load5.1 = load <4 x i32>* %30, align 4, !tbaa !1
  %31 = add nsw <4 x i32> %wide.load.1, %25
  %32 = add nsw <4 x i32> %wide.load5.1, %26
  %index.next.1 = add i64 %index.next, 8
  %33 = icmp eq i64 %index.next.1, %n.vec
  br i1 %33,
        label %middle.block.loopexit.unr-lcssa,
        label %vector.body,
        !llvm.loop !5

middle.block.loopexit.unr-lcssa:
  %.lcssa14.ph = phi <4 x i32> [ %32, %vector.body ]
  %.lcssa13.ph = phi <4 x i32> [ %31, %vector.body ]
  br label %middle.block.loopexit
```

В приведённой выдержке из кода LLVM для исходного кода

```
int fn(const int A[], unsigned N)
{
        int sum = 0;
        for(unsigned i = 0; i< N; i += 1)
        {
                sum += A[i];
        }

        return sum;
}
```

значения, присвоенные «регистрам» `%31` и `%32` переживают инструкцию ветвления `br`. Что плохо вписывается в поведение МСр.

1.2. Основные цели разработки

Основное, но не единственное – это, конечно же, достичь такого представления программы, которое было бы удобно для последующей генерации кода для процессоров МСр. И позволяло бы применять необходимые алгоритмы оптимизации.

Кроме этого, хотелось бы упростить разработку за счёт следующего.

- Повторного использования реализаций языковых конструкций. Во многих языках код «`x + y`» означает одно и то же.
- Использования событийной модели трансляции кода. Можно ожидать некоторого упрощения работы за счёт этого, потому что *событийные модели параллельных процессов* (например, CSP) похожи на *модели рекурсивных типов* из теории языков программирования (μ-типы). Сами же событийные модели часто упрощают разработку сложных систем (например, каналы UNIX).
- Освобождения разработчиков от необходимости заниматься управлением ресурсами на низком уровне за счёт предоставления им специального языка для описания семантики языковых конструкций.
- Поддержки процессоров с различными архитектурами для привлечения к работам сторонних разработчиков.

Кроме этого хотелось бы обеспечить самим себе относительно простой механизм наращивания семантики исходного языка для использования архитектурных особенностей МСр на соответствующем языковом уровне.

1.3. Общая структура системы

Можно описать примерно такой формулой из мира UNIX

```
frontend < source-file \
 | lime-kernel -f forms-1.lk ... -f forms-N.lk \
 | codegen
```

Всё разбито на независимые процессы. Цели:

- изолировать домены ошибок (failure domains) и утечек памяти (потому что давление заставляет нас херачить код подобно бешеным кроликам круглосуточно, горы ошибок и постоянные утечки в таких условиях просто неизбежны);
- через определение форматов входных данных можно разделить работу (и это получилось);
- открыть разработчикам возможность описывать компоненты на наиболее удобных языках программирования (вообще, имело бы смысл это всё писать на Python или JavaScript, но где бы мы нашли тогда разработчиков?).

Указанные в коде выше программы для соответствующих процессов должны выполнять следующие роли.

- `frontend` – группа программ, которая должна превратить текст на исходном языке в определённое описание синтаксического дерева программы.
- `lime-kernel` – программа, которая по набору форм, описывающих правила вывода графа программы, и по исходному дереву программы, выдаёт граф программы.
- `codegen` – генератор кода, естественно.

1.3.1. Фасад (frontend)

Основная задача фасада преобразовать исходный текст на ЯП в описание синтаксического дерева в некотором формате. Например, для нашего компилятора Си

```
mcpp < source-code.c | mc-cfe
```

В этой структуре `mcpp` – стандартный препроцессор для обработки директив препроцессора Си таких как `#include` для подстановки в текущий исходный файл других файлов, или как `#define` для определения макросов. Кроме этого, задача препроцессора вычищать комментарии, обрабатывать три-графы, сращивание строк. Подробности здесь: `http://en.wikipedia.org/wiki/C\_preprocessor`

`mc-cfe` – специальный транслятор с «чистого» Си в язык синтаксических деревьев:

- `A hint.size."bytes"` – листья дерева, например, `A 02.1."a"` – идентификатор Си;
- `L hint.size."bytes"` – начало «бинарного» узла в дереве, левым поддеревом которого является предыдущее выражение, для Си, например, `A 01.1."a"; L 14.1."+"` – описывает «частичное» выражение `a +`;
- `U hint.size."bytes"` – начало «унарного» узла в дереве, например,

  ```
  U 0c.3."p++"
  ```

  оператор пре-инкремента в Си;
- `E hint.size."bytes"` – конец описания «унарного» или «бинарного» узла в дереве, описывает конец поддерва (правого, если узел «бинарный»), например,

  ```
  A 02.1."a"; L 14.1."+"; A.02.1."b"; E 14.1."+"
  ```

  описывает выражение `a + b`.

Так, например, программу

```
fn () { return a + b; }
```

фасад превращает в дерево

```
A       0.0     01.3."int"
L       0.4     06.11."@–func–decl"
A       0.0     02.2."fn"
L       0.3     06.6."@–argv"
A       0.0     ff.7."NOTHING"
E       0.0     06.6."@–argv" // 2
L       0.3     06.11."@–func–body"
U       0.0     0a.1."{"
A       0.2     01.6."return"
L       0.7     06.9."@retvalue"
A       0.0     02.1."a"
L       0.2     14.1."+"
A       0.2     02.1."b"
E       0.0     14.1."+" // 2
E       0.0     06.9."@retvalue" // 5
E       0.0     0a.1."{" // 8
E       0.0     06.11."@–func–body" // 10
E       0.0     06.11."@–func–decl" // 16
```

Атомами в системе описываются все токены исходной программы.

Вторая колонка в примере выше – это позиции токенов в исходном тексте программы, заданные своими смещениями относительно предыдущих в формате `line.offset`.

1.3.2. Генератор кода

Генератор на вход получает такие примерно представления программ (`a + b`; `a` и `b` – *новые переменные*):

```
(
  .L n0
  (
    .T     n0  ('00.1."P"; 4);
    .T     n1  ('00.1."I"; 4);
    .T     n2  ('00.5."world"; n0; n1);
    .TDef  n3  (n1; ('00.5."hello"; '00.4."test"; n2));
    .S     n4  ('02.1."a"; n1);
    .S     n5  ('02.1."b"; n1);
    .addr  n6  (n0; n4);
    .rd    n7  (n1; n6);
    .addr  n8  (n0; n5);
    .rd    n9  (n1; n8);
    .add   n10 (n1; n7; n9)
   );
  .Label n1 (01.2."L3"; n0)
)
```

- В атрибутах узлов типа `L` содержатся деревья линейных участков.
- «Недеревянные» связи между линейными участками генератор воспроизводит по узлам `Label` (метки) и узлым `br` (ветвления). Это «*внутренние знания*» генератора.

Используемые графы, конечно, являются деревьями. Их текстовое представление описывается грамматикой:

$G \to$ «`()`» | «`(`» A «`)`»	граф
$A \to a \mid a$ «`;`» A	список атрибутов
$a \to G \mid n \mid num \mid atom \mid id$	атрибут
$n \to$ «`.`» $id\ (G \mid id\ G)$	узел, например: `.S n5 ('02.1."b"; n1)`
num	натуральное число в десятичной записи
$atom$	атом
id	идентификатор

Семантика этих конструкций тоже не особо выдающаяся и очевидная. Так выражение

```
.add n10 (n1; n7; n9)
```

описывает узел графа, в который ведут дуги из узлов `n1`, `n7`, `n9` (порядок этих узлов важен только для генератора кода, ядро системы оперирует всеми подобными узлами общим способом).

Имя узла, если присутствует, указывается идентификатором после его типа – выражения вида «`.id`». Имена узлов можно не плодить и указывать дугу из узла α в узел β, указав узел α в списке атрибутов β. Поэтому выражение

```
.add n10 (n1; n7; n9)
```

эквивалентно

```
.add n10 (n1; .rd (n1; n6); .rd (n1; .S ('02.1."b"; n1)))
```

При загрузке графов генератор кода может пользоваться библиотечными функциями системы. Например,

```
eval(C, NULL, D, 0, NULL, EMGEN)
```

обрабатывает входной граф `D` в режиме генерации кода, то есть, с восстановлением по узлам `E`, `T`, `TDef`, `TEnv`, `S` таблиц окружений, типов и символов.

1.4. Ядро

Задача ядра по синтаксическому дереву построить граф программы. Этот граф выводится в соответствии с математической моделью RiDE. Причины такого решения:

- RiDE – это система динамического построения графа вычисления, а графы – везде графы;
- эксперименты с RiDE показывают, что объём кода, описывающий продолжения вычислений, можно существенно сократить (прямая

аналогия: *продолжение распределённого вычисления – продолжение графа программы*, в обоих случаях неопределённость по необходимым для этого данным);

- философские рассуждения позволяют говорить: активация продолжения графа вычисления соответствуют *динамическому построению сопоставлений по шаблону* – основы компиляторов, и мы знаем, что динамические графы удобнее статических;
- системы управляемые событиями логически проще, чем функциональные и императивные системы, например, конвейер процессов `a | b | c` *проще тройного вложенного цикла*;
- модель событийной системы CSP совпадает с моделью рекурсивных μ-типов – одного из оснований теории языков программирования.

1.4.1. Основной цикл

Ядро считывает синтаксические команды и на их основе перестраивает стек областей вывода. Области вывода описывают текущие графы программ для тех или иных подвыражений в исходном коде.

В каждой области есть (**1**) два «реактора» (на самом деле, это просто две дочерние области вывода, но так как нас всё время пинают и не дают нормально подумать, то получаются вот такие вот ad-hoc решения); (**2**) одна таблица связности с другими областями (на самом деле, можно было бы и без этого усложнения, но пинки под зад заставляют принимать достаточно корявые решения; это всё надо исправлять в следующей версии); и (**3**) накопленный для этой области вывода граф программы.

Каждый реактор представляет собой структуру:

- таблица из пар имя:значние, в которой накапливается информация об опубликованных под определёнными именами значениях: узлах графа, типах, символах и выражениях из атомов, номеров, узлов, типов, символов;

- список ожидающих своей *активации* форм.

Таблица связности между областями вывода – это просто таблица из пар вида имя:«ссылка на область».

Граф программы – это граф программы. Чисто технически он приписывается к 1-му реактору области вывода.

Именами в этих разных таблицах служат выражения из типов, атомов и чисел. Выражения структурно являются подмножеством выражений для описания атрибутов узлов. Например,

```
(0; 1; ('0.1."a"; .T (34)); .T (.T (34); 76; '3.4."5678"))
```

Теперь можно описать основной цикл вывода.

- Встречая $s =$ `{A|U} hint.size."bytes"`, ядро размещает на стеке новый контекст, «засевая» его 0-й реактор парой $s{:}s$ и формой, найденной по одному из имён (порядок поиска имеет значение):
  - `({A|U}; hint.size."bytes")`,
  - `({A|U}; .T (hint.0.""))`.
- Встречая `L hint.size."bytes"`, ядро выталкивает из стека контекст c_L, размещает на вершине новый контекст c_t, «засевает» его, и заносит в таблицу связанных с c_t окружений окружение c_l под именем `0.4."LEFT"`.
- Встречая `E hint.size."bytes"`, ядро выталкивает из стека контекст c_R, проверяет, что это `{U|L}`-контекст с подходящим атомом, «засевает» контекст на вершине стека c_t, и заносит в таблицу известных в c_t окружение c_R под именем `0.5."RIGHT"`.

Обработав очередной синтаксический элемент ядро переходит к циклу вывода в различных областях. Модель вычислений RiDE позволяет этот вывод осуществлять в произвольном порядке, даже сохраняя контексты во внешнюю память.

Вывод продолжается, пока ядро не встретит в одной из активированных на вершине стека узел `.Go`. *Таким образом ядро создаёт дерево взаимодействующих процессов вывода, отражающее структуру синтаксического дерева*.

1.4.2. Цикл вывода и формы

После манипуляции областями на стеке и «засева» их данными ядро переходит к циклу вывода графов программы, соответствующих различным узлам деревьев.

Основной сущностью в процессе ввывода графов является форма. Формы – это параметризованный кусочек графа. Параметризован этот кусочек с «двух сторон». С одной стороны, со стороны входов, к некоторым узлам, определённым внутри этого кусочка можно протянуть дуги от некоторых уже существующих в графе программы узлов и значений отмеченных в области вывода некоторыми именами.

С другой стороны, стороны выходов, этот кусочек после интерпретации ядром сам может стать источником именованных узлов и значений в области вывода.

Кроме того, интерпретация формы может привести к публикации в некоторых областях вывода новых форм, ожидающей своей активации.

- Форма может быть *опубликована* в контексте вывода. При публикации формы задаются имена, появление которых в соответствующей области вывода к активации формы и к «протягиванию» в её граф дуг от этих узлов.

- Форма может быть *активирована*. В этом процессе ядро на основании анализа тела формы:

  - *достраивает граф программы*, связанный с текущим контекстом вывода,
  - *публикует формы*, по описаниям в теле активированной формы,

- *именует некоторые узлы и значения* (не обязательно новые) по описаниям в теле активированой формы.

С появлением новых имён у прежних или новых узлов графа и новых опубликованных форм процесс вывода продолжается.

Далее пара примеров форм. Начнём с простой

```
.FEnv (("binop"; '14.1."+"); .F
(
  .Nth l (.FIn (); (1; 0));
  .Nth r (.FIn (); (1; 1));

  .add result (.T ("I"; 4); l; r);
  .FOut (0; (("result"; result)))
));
```

Эта простая форма описывает кусочек графа с единственным узлом `.add`, задающим операцию целочисленного (в терминах Си-99 `int`) сложения. То, что именно такой узел описывает именно такое сложение «знает» только генератор кода.

Телом формы служит граф, описанный в списке атрибутов узла `.F`

Форма описывает наращивание графа узлом `.add`, соответствующим машинной операции, этот узел «привязывается» к двум другим узлам, которые задаются узлами `.Nth`. Несмотря на «дикий» синтаксис конструкции относительно простые.

- `FIn` – ссылка на входы формы;
- `Nth` – позволяет назвать необходимые узлы из входов формы;
- `FOut` – опубликовывает определённое имя для определённого узла;
- `FEnv` – регистрация формы с определённой сигнатурой в окружении (в частности, ядро ищет по этим сигнатурам формы для «посева»).

2. **Фасад Си-99. Готовность 0.9**
3. **Генератор кода. Готовность 0.9**
4. **Ядро. Готовность 0.9**
5. **Семантика Си-99. Готовность 0.1**
6. **Разбор примера**