Древесные оптимизации в компиляторе Рефала-5λ: итоги и перспективы

А. В. Коновалов

МГТУ имени Н. Э. Баумана

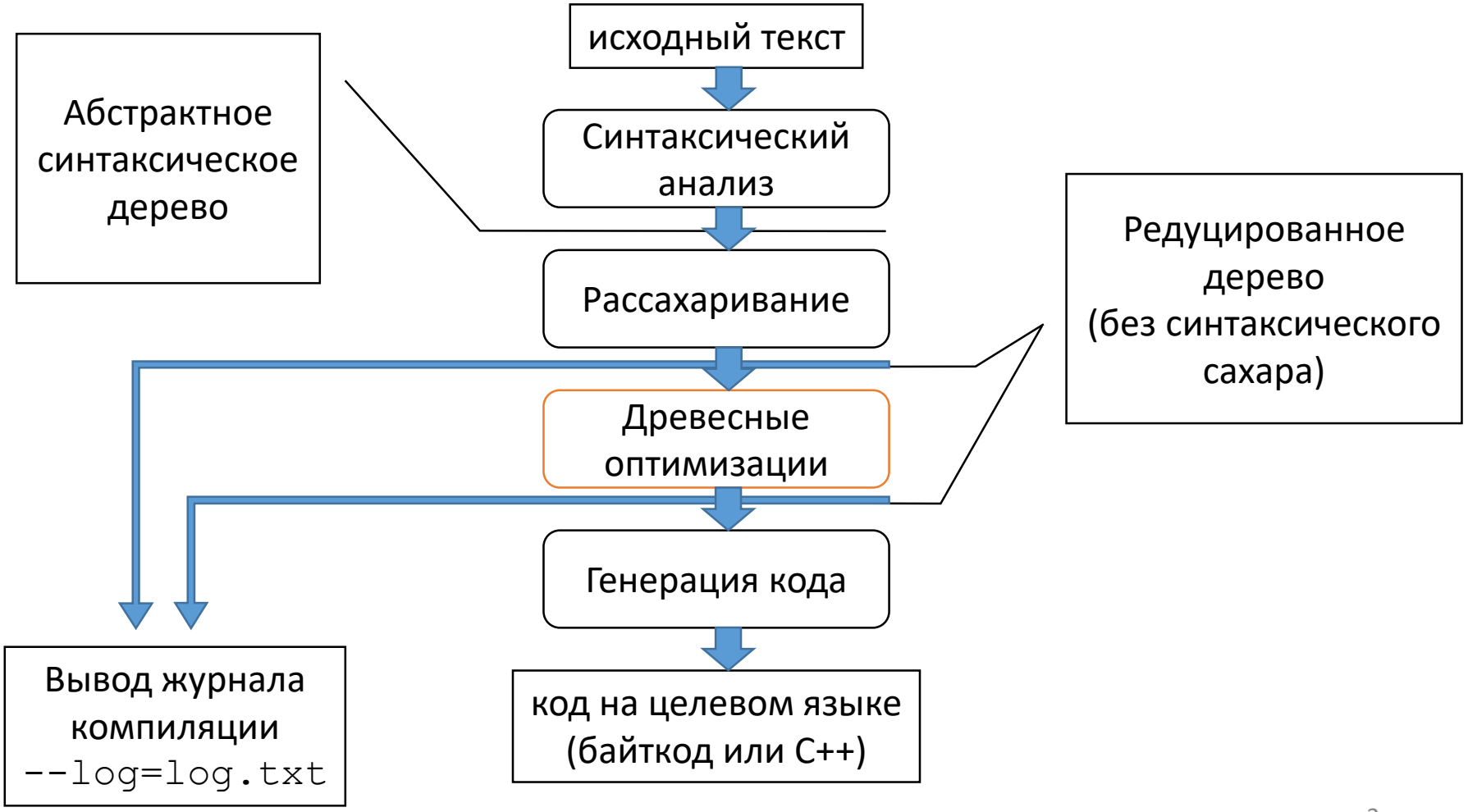
Третье совместное рабочее совещание
ИПС имени А. К. Айламазяна РАН и МГТУ имени Н. Э. Баумана
по функциональному языку программирования Рефал

12 июня 2020 года

Оптимизации в компиляторе Рефала-5λ

- Оптимизации на уровне синтаксического дерева, преобразуют программу в терминах языка Рефал:
  - **прогонка** (`-OD`, `-OI`),
  - **специализация** (`-OS`),
  - глобальная оптимизация (`-OG`),
  - автоматический поиск оптимизируемых функций (`-OA`) *(under construction)*,
  - оптимизация intrinsic-функций (`-Oi`) *(under construction)*.
- Оптимизации на уровне языка сборки:
  - оптимизация совместного сопоставления с образцом (`-OP`),
  - оптимизация построения результатных выражений (`-OR`).
- Оптимизация на уровне генерации кода:
  - компиляция не в интерпретируемый код, а в код на C++ (`-Od`).

Место древесных оптимизаций в архитектуре компилятора



Синтаксис редуцированного («рассахаренного») дерева

```
Program = Unit*
Unit = Function | Declaration
Function = ['$ENTRY'] Name '{' Sentence* '}'
Sentence = Pattern Condition* Result ';'
Condition = ',' Result ':' Pattern
Pattern, Result = Expression
Expression = Term*
Term = Symbol
     | '(' Expression ')'
     | '[' NAME Expression ']'
     | '<' Expression '>'
     | '{{' Pointer Expression '}}'
Symbol = NAME | Pointer | CHAR | NUMBER
Pointer = '&' NAME
```

Вызовы функций и замыкания допустимы только в результатных выражениях

Пример рассахаривания программы — декартово произведение

```
$ENTRY CartProd {
  e.Xs e.Ys
    = <Map
        {
          t.X = <Map { t.Y = (t.X t.Y) } e.Ys>
        }
        e.Xs
      >
}
```

```
CartProd\1\1 {
  t.X#2 t.Y#3 = (t.X#2 t.Y#3);
}

CartProd\1 {
  (e.Ys#1) t.X#2 = <Map {{&CartProd\1\1 t.X#2}} e.Ys#1>;
}

$ENTRY CartProd {
  e.Xs#1 e.Ys#1 = <Map {{&CartProd\1 (e.Ys#1)}} e.Xs#1>;
}
```

Оптимизация прогонки

- Прогонка функций была описана Турчиным ещё в 1972 году:
  - В. Ф. Турчин. *Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ.* В сб.: Труды симпозиума «Теория языков и методы построения систем программирования», Киев-Алушта: 1972
  - В. Ф. Турчин. *Эквивалентные преобразования программ на РЕФАЛе.* В сб.: Труды ЦНИПИАСС «Автоматизированная система управления строительством», выпуск 6, М: 1974. Стр. 36-68.
- При прогонке устраняется один из вызовов функции в правой части одного из предложений.
- Логика вызывающей функции «сливается» с логикой «вызываемой». Можно сказать, что происходит встраивание одной функции в другую.
- Имя функции, чьи вызовы должны прогоняться, должно быть записано в объявлении `$DRIVE` или `$INLINE`. Вложенные функции, функции-блоки и функции-присваивания прогоняются по умолчанию.
- Прогонка рекурсивной функции может зациклить компилятор.

Прогонка: пример

```
$ENTRY Test {
  e.Y = <Last 'A' e.Y>;
}

$DRIVE Last;

Last {
  e.Begin s.Last = s.Last;
}
```

```
$ENTRY Test {
  /* empty */ = 'A';
  e.#0 s.Last#1 = s.Last#1;
  e.Y#1 = <Last*1 'A' e.Y#1>;
}

Last {
  e.Begin#1 s.Last#1 = s.Last#1;
}

Last*1 {
}
```

Ограничения прогонки

```
F {
  …
  Pat = E1 <D E2> E3
  …
}

$DRIVE D;

D {
  Pat1 = Res1;
  …
  PatN = ResN;
}
```

Должно быть пассивным

Должны быть L-выражениями

- Условия в образцах запрещены
- L-выражение — образцовое выражение, в котором запрещены
  - открытые e-переменные,
  - повторные t- и e-переменные.

Специализация функций

- Специализируемая функция компилятором рассматривается как шаблон — для каждого её вызова строится *экземпляр* функции, учитывающий статически известную информацию из вызова.
- Для специализируемых функций описывается их входной формат при помощи объявления $SPEC:

```
$SPEC FuncName формат;
```

- В формате выделяются *статические* и *динамические* параметры.
- Вызов должен соответствовать формату (иначе вызов не оптимизируется).
- Значения статических аргументов учитываются при создании экземпляра, динамических — нет.
- Образцы предложений функции должны соответствовать формату. При этом статические параметры могут отображаться в образцах только на переменные того же типа.
- Специализация рекурсивных функций может приводить к зацикливанию компилятора.

Пример специализируемой функции

Типичная функция, предназначенная для специализации — Map. Она принимает указатель на функцию и вызывает его с каждым элементом последовательности.

```
$SPEC Map s.FUNC e.items;

Map {
   s.Func t.Next e.Rest
      = <s.Func t.Next>
        <Map s.Func e.Rest>;

   s.Func /* пусто */ = /* пусто */;
}
```

Статический параметр — с большой буквы

Динамический — с маленькой буквы

Статические параметры отображаются на переменные, динамическе — нет.

Пример использования специализации

- Задача: есть последовательность вида

```
e.Items ::= t.Item*
t.Item ::=
      (Valid t.Val)
    | (Message e.Msg)
    | (Invalid t.Val)
```

- Элементы типа `Valid` нужно оставить.
- Элементы типа `Message` — распечатать.
- Элементы типа `Invalid` — удалить.

Пример использования специализации

Решение на Рефале-5:

```
Scan {
  (Valid t.Val) e.Rest
    = t.Val <Scan e.Rest>;

  (Message e.Msg) e.Rest
    = <Prout e.Msg> <Scan e.Rest>;

  (Invalid t.Val) e.Rest
    = /* пусто */ <Scan e.Rest>;

  /* пусто */ = /* пусто */;
}
```

Пример использования специализации

Решение на Рефале-5λ:

```
Scan {
  e.Items
    = <Map
        {
          (Valid t.Val) = t.Val;
          (Message e.Msg)
            = <Prout e.Msg>;
          (Invalid t._) = /* пусто */;
        }
        e.Items
      >;
}
```

Пример использования специализации

Псевдокод синтаксического дерева:

```
$SPEC Map s.FUNC#0 e.items#0;

Map {
  s.Func#1 t.Next#1 e.Rest#1
    = <s.Func#1 t.Next#1> <Map s.Func#1 e.Rest#1>;

  s.Func#1 = /* empty */;
}

Scan\1 {
  (Valid t.Val#2) = t.Val#2;
  (Message e.Msg#2) = <Prout e.Msg#2>;
  (Invalid t._#2) = /* empty */;
}

$DRIVE Scan\1;

Scan {
  e.Items#1 = <Map &Scan\1 e.Items#1>;
}
```

Пример использования специализации

После прохода специализации:

…

```
Scan\1 {
   (Valid t.Val#2) = t.Val#2;
   (Message e.Msg#2) = <Prout e.Msg#2>;
   (Invalid t._#2) = /* empty */;
}

Scan {
  e.Items#1 = <Map@1 e.Items#1>;
}

Map@1 {
  t.Next#1 e.Rest#1
    = <Scan\1 t.Next#1> <Map &Scan\1 e.Rest#1>;

  /* empty */ = /* empty */;
}
```

Пример использования специализации

После прохода прогонки:

…

```
Scan {
  e.Items#1 = <Map@1 e.Items#1>;
}

Map@1 {
  (Valid t.0#0) e.Rest#1 = t.0#0 <Map &Scan\1 e.Rest#1>;

  (Message e.0#0) e.Rest#1
    = <Prout e.0#0> <Map &Scan\1 e.Rest#1>;

  (Invalid t.0#0) e.Rest#1 = <Map &Scan\1 e.Rest#1>;

  t.Next#1 e.Rest#1
    = <* &Scan\1*3 t.Next#1*> <Map &Scan\1 e.Rest#1>;

  /* empty */ = /* empty */;
}

Scan\1*3 {
}
```

Пример использования специализации

После второго прохода специализации:

…

```
Scan {
  e.Items#1 = <Map@1 e.Items#1>;
}

Map@1 {
  (Valid t.0#0) e.Rest#1 = t.0#0 <Map@1 e.Rest#1>;

  (Message e.0#0) e.Rest#1
    = <Prout e.0#0> <Map@1 e.Rest#1>;

  (Invalid t.0#0) e.Rest#1 = <Map@1 e.Rest#1>;

  t.Next#1 e.Rest#1
    = <Scan\1*3 t.Next#1> <Map@1 e.Rest#1>;

  /* empty */ = /* empty */;
}
```

Пример использования специализации

- В результате оптимизаций мы получили функцию, эквивалентную той, которую мы бы написали вручную без Map.
- Функция, написанная вручную, сложнее читается и более подвержена ошибкам.
- Функция, написанная с использованием Map, проще для восприятия, но без оптимизаций была бы медленнее.
- Оптимизации позволяют программисту писать более выразительный код без ущерба для производительности.

Ограничения специализации

- Необходимость задавать шаблон функции.
- Статические переменные должны отображаться на переменные в предложениях.

Развитие прогонки

- Прогонка вызовов функций в условиях, прогонка вызовов функций в предложениях с условиями.
- Прогонка функций с условиями.
- Прогонка в предложениях с не-L-образцом.
- Разрешить аргументам прогоняемых вызовов быть активными.
- **Расширение алгоритма обобщённого сопоставления.**

Развитие алгоритма обобщённого сопоставления с образцом

- Алгоритм обобщённого сопоставления с образцом — «сердце» прогонки.
- Алгоритм решает уравнение вида

  E : P

  - E — некоторое выражение с переменными,
  - P — некоторый образец.
- Решением уравнения является набор подстановок в E — *сужений* и подстановок в P — *присваиваний*, такой, что уравнение обращается в тождество.
- Одно уравнение может иметь несколько решений.
- Алгоритм обобщённого сопоставления является разновидностью алгоритма унификации.
- **Классический турчинский алгоритм умеет решать уравнения только для случая, когда P — L-выражение.**

Развитие алгоритма обобщённого сопоставления с образцом

- Можно выделить три ситуации, когда уравнение E : P можно решить:
  - E — произвольное выражение, P — L-выражение,
  - E — объектное выражение (т.е. без переменных), P — произвольный образец,
  - E — e-переменная e.X, P — произвольный образец, решение в этом случае — сужение e.X → P и пустой набор присваиваний.
- Нужно обобщить алгоритм таким образом, чтобы он не только давал решение в этих трёх граничных точках, но и «в их окрестности».

Развитие специализации

- Очевидное направление развития — специализация без шаблона.
- Функцию вида

```
$SPEC Func;

Func {
  ⟨тело функции⟩
}
```

- можно условно трактовать как

```
$SPEC Func e.ARG;

Func {
  e.X = <Func' e.X>;
}

$DRIVE Func';

Func' {
  ⟨тело функции⟩
}
```

Развитие специализации

- При таком подходе грань между прогонкой и специализацией размывается.
- При прогонке вызова `<F ARG>` решается набор уравнений
  - `ARG : Pat1`
  - …
  - `ARG : PatN`
- и их решения применяются к предложению с вызовом.
- При специализации вызова <F ARG> решается тот же набор уравнений, но их решения превращаются в предложения экземпляра.

Развитие специализации

- Исходная программа:

```
… <Rot 'A' e.X e.Y> …

$SPEC Rot;

Rot {
  e.1 s.2 = s.2 e.1;
}
```

- Специализированная программа:

```
… <Rot@1 (e.X) e.Y> …

Rot@1 {
  (/*ε*/) /*ε*/ = 'A';

  (e.1 s.2) /*ε*/
    = s.2 'A' e.1;

  (e.1) e.2 s.3
    = s.3 'A' e.1 e.2;
}
```

«Динамическое обобщение»

- Допустим, мы хотим проспециализировать вызов

```
... <S ARG> ...

$SPEC S;
S {
  Pat = Res;
}
```

- Нам необходимо решить уравнение
  - `ARG : Pat`
- Но, что если его решить невозможно?
- Предлагается искать такое выражение `ARG'`, являющееся обобщением `ARG` (существует подстановка `S`, такая что `ARG'/S ≡ ARG`) и уравнение `ARG' : Pat` разрешимо.

Объединение оптимизаций

- Вместо меток `$DRIVE`, `$INLINE` и `$SPEC` можно будет использовать метку `$OPT`.
- Если функцию, помеченную как `$OPT`, можно прогонять (она не является рекурсивной), то её вызовы прогоняются.
- Если нельзя — вызовы специализируются.
- После специализации вновь построенные экземпляры проверяются на возможность прогонки: экземпляр рекурсивной функции может быть нерекурсивным и его безопасно будет прогонять.

Выводы

- Прогонка и специализация позволяют осуществлять глубокие оптимизации программ, но имеют ряд ограничений.
- Многие из этих ограничений устранимы.
- Задачи по развитию оптимизаций довольно интересные.