Специализация функций в Рефале-5λ

Д. П. Сухомлинова

Руководитель:
А. В. Коновалов

МГТУ имени Н. Э. Баумана
28 июня 2019 года

Постановка задачи

- Целью данной работы является расширение возможностей оптимизаций компилятора Рефала-5λ добавлением специализаций функций

- Для этого необходимо:
  - определить правила объявления специализации функции по паттерну и ключи компиляции для включения данной оптимизации;
  - реализовать алгоритм специализации;
  - оценить изменение скорости компиляции и скорости работы программы при применении оптимизации специализации на примере самоприменимого компилятора языка

Специализация в общем и частном случае

- *«Специализация программ — это порождение по универсальной программе с множеством параметров специализированной программы, когда значения части параметров известны и фиксированы»*

(Андрей В. Климов «Введение в метавычисления и суперкомпиляцию»).

- Мы рассматриваем специализацию отдельно взятого определения функции для случаев её вызова в отдельных точках программы

Ограничения специализации функций. Количество аргументов

- **Проблема:** функции на Рефале-5λ принимают ровно один аргумент

- **Решение:** явное описание форматов функций в виде жёстких выражений и обязательной проверки каждого вызова функций на предмет его соответствия формату

Ограничения специализации функций. Формат аргумента

- **Проблема:** необходимо различать параметры, по которым ведётся специализация (*статические*), и по которым специализация не ведется (*динамические*)

- **Решение:** определение формата жесткого выражения, в котором специализируемые и обычные параметры промаркированы

Объявление специализации функции

- `$SPEC ИмяФункции ЖёсткоеВыражение;`

- ЖёсткоеВыражение — образец, не содержащий двух e-переменных на одном скобочном уровне

- Дополнительное ограничение:
  - Образец не должен содержать повторных переменных

- Индексы переменных в образце не имеют значения, кроме первого символа:
  - Если индекс переменной начинается с заглавной латинской буквы, то соответствующий ей параметр считается специализируемым
  - Если индекс начинается со строчной латинской буквы, цифры, дефиса (-) или нижнего подчёркивания (_), то соответствующий ей параметр считается обычным

Семантика специализации

- Для каждой функции допустимо не более одного объявления специализации.
- Специализируемые функции должны быть определены в области видимости текущей единицы трансляции.
- Если в образце спецификатора есть ссылки на функции, то они должны быть определены (например, если образец содержит АТД-скобки).
- Каждое образцовое выражение в определении функции должны быть уточнениями формата специализации, при этом в подстановках, переводящих шаблон специализации в каждое образцовое выражение:
  - специализируемые параметры должны заменяться на переменные соответствующего типа, при этом различные статическим параметрам должны соответствовать различные переменные;
  - обычные параметры в подстановках могут заменяться на любое выражение в соответствии с их типом, по определению уточнения.

Алгоритм специализации на примере модельного языка

```
Prog ::= Def*

Def ::= F(Params; Params) = Expr

Params ::= Var*

Expr ::= if Expr then Expr else Expr
       | F(Params; Params)
       | Value
       | let Var := Expr in Expr

Args ::= Value*

Value ::= Var | C(Args)
```

Алгоритм специализации. Исходная функция и вызов

```
replace (x, y; list) =
  if list is nil then
    Nil
  then
    let head := car(list) in
    let tail := cdr(list) in
    let new_tail := replace(x, y; tail)
    let new_head := (if head == x then y else head) in
      Cons(new_head, new_tail)

replace(C(x,y), C(y,x); x y z C(x,y) B(x,z) A(z, y))
```

Алгоритм специализации. Получение сигнатуры вызова

```
replace(C(x,y), C(y,x); x y z C(x,y) B(x,z) A(z, y))
```

```
  => C( v1, v2), C( v2, v1)
```

Алгоритм специализации. Формирование нового вызова

```
replace(C(x,y), C(y,x); x y z C(x,y) B(x,z) A(z, y))

  => replace@1(x, y, x y z C(x,y) B(x,z) A(z, y))
```

Алгоритм специализации. Формирование нового определения функции. Первая итерация

```
replace@1 (_v1, _v2, list) =
  if list is nil then
    Nil
  then
    let head := car(list) in
    let tail := cdr(list) in
    let new_tail := replace(C(_v1,_v2), C(_v2,_v1); tail)
    let new_head :=
      (if head == C(_v1,_v2) then C(_v2,_v1) else head) in
      Cons(new_head, new_tail)
```

Алгоритм специализации. Формирование нового определения функции. Вторая итерация

```
replace@1 (_v1, _v2, list) =
  if list is nil then
    Nil
  then
    let head := car(list) in
    let tail := cdr(list) in
    let new_tail := replace@1(_v1, _v2; tail)
    let new_head :=
      (if head == C(_v1,_v2) then C(_v2,_v1) else head) in
      Cons(new_head, new_tail)
```

Пример работы. Исходный код

```
$SPEC Replace (e.FROM) (e.TO) e.items;


Replace {
  (e.f) (e.t) e.f e.Tail = e.t <Replace (e.f) (e.t) e.Tail>;
  (e.f) (e.t) t.X e.Tail = t.X <Replace (e.f) (e.t) e.Tail>;
  (e.f) (e.t) /* empty */ = /* empty */;
}

F {
  s.1 s.2 e.X = <Replace (s.1 s.2) (s.2 s.1) e.X>;
}

$ENTRY Go {
  = <F 1 5 1 1 1 2 2 2 3 3 3>;
}
```

Пример работы. Исходное абстрактное синтаксическое дерево

```
$SPEC Replace (e.FROM#0) (e.TO#0) e.items#0;

Replace {
  (e.f#1) (e.t#1) e.f#1 e.Tail#1 = e.t#1 <Replace (e.f#1) (e.t#1) e.Tail#1>;

  (e.f#1) (e.t#1) t.X#1 e.Tail#1 = t.X#1 <Replace (e.f#1) (e.t#1) e.Tail#1>;

  (e.f#1) (e.t#1) = /* empty */;
}

F {
  s.1#1 s.2#1 e.X#1 = <Replace (s.1#1 s.2#1) (s.2#1 s.1#1) e.X#1>;
}

$ENTRY Go {
  /* empty */ = <F 1 5 1 1 1 2 2 2 3 3 3>;
}
```

Пример работы. Преобразованное абстрактное синтаксическое дерево. Первая итерация

```
Replace {
  (e.f#1) (e.t#1) e.f#1 e.Tail#1 = e.t#1 <Replace (e.f#1) (e.t#1) e.Tail#1>;
  (e.f#1) (e.t#1) t.X#1 e.Tail#1 = t.X#1 <Replace (e.f#1) (e.t#1) e.Tail#1>;
  (e.f#1) (e.t#1) = /* empty */;
}
F {
  s.1#1 s.2#1 e.X#1 = <Replace@1 s.1#1 s.2#1 (e.X#1)>;
}
$ENTRY Go {
  /* empty */ = <F 1 5 1 1 1 2 2 2 3 3 3>;
}
Replace@1 {
  s.1#1 s.2#1 (s.1#1 s.2#1 e.Tail#1)
    = s.2#1 s.1#1 <Replace (s.1#1 s.2#1) (s.2#1 s.1#1) e.Tail#1>;
  s.1#1 s.2#1 (t.X#1 e.Tail#1) = t.X#1 <Replace (s.1#1 s.2#1) (s.2#1 s.1#1) e.Tail#1>;
  s.1#1 s.2#1 () = /* empty */;
}
```

Пример работы. Преобразованное абстрактное синтаксическое дерево. Вторая итерация

```
Replace {
  (e.f#1) (e.t#1) e.f#1 e.Tail#1 = e.t#1 <Replace (e.f#1) (e.t#1) e.Tail#1>;
  (e.f#1) (e.t#1) t.X#1 e.Tail#1 = t.X#1 <Replace (e.f#1) (e.t#1) e.Tail#1>;
  (e.f#1) (e.t#1) = /* empty */;
}
F {
  s.1#1 s.2#1 e.X#1 = <Replace@1 s.1#1 s.2#1 (e.X#1)>;
}
$ENTRY Go {
  /* empty */ = <F 1 5 1 1 1 2 2 2 3 3 3>;
}
Replace@1 {
  s.1#1 s.2#1 (s.1#1 s.2#1 e.Tail#1)
    = s.2#1 s.1#1 <Replace@1 s.1#1 s.2#1 (e.Tail#1)>;
  s.1#1 s.2#1 (t.X#1 e.Tail#1) = t.X#1 <Replace@1 s.1#1 s.2#1 (e.Tail#1)>;
  s.1#1 s.2#1 () = /* empty */;
}
```

Тестирование

	Среднее значение	Доверительный интервал	
Компиляция без специализации	14.60297	13.45032	16.53564
Компиляция со специализацией	20.45314	19.60385	20.77855
Компиляция специализированным компилятором	14.24620	13.09759	15.45941

Заключение

- В результате был реализован алгоритм специализации функций в самоприменимом компиляторе Рефала-5λ

- Для специализации функции необходимо явно определить шаблон, по которому будут сопоставляться фактические значения параметров специализации

- В частном случае, при раскрутке компилятора, реализованный алгоритм не дал заметной выгоды при исполнении программы, но значительно замедлил процесс компиляции. Однако, реализация специализации открывает больше возможностей для других оптимизаций, и предполагается, что комбинирование оптимизации специализации и прогонки даст более заметное ускорение работы программы, чем их отдельное использование

Спасибо за внимание!