Проблемы ссылочной эквивалентности замыканий при эквивалентных преобразованиях программ в Рефале-5λ

Александр В. Коновалов

МГТУ имени Н.Э. Баумана

г. Москва

Семинар по метавычислениям им. В.Ф. Турчина — онлайн — 17 мая 2021

План доклада

- Язык программирования Рефал-5λ и архитектура его компилятора.
- Гарантии равенства для замыканий, описанные в официальном руководстве, реальное поведение компилятора.
- Оптимизация прогонки вызовов функций.
- Оптимизация специализации вызовов функций.
- Специализация замыканий.
- Проблемы с равенством замыканий при наивной реализации оптимизаций.
- Пути решения проблем, их преимущества и недостатки.

Рефал-5λ: расширение Рефала-5

Более свободное использование блоков:

```
F {
  e.X e.Name e.Y
    , e.Name : { 'Саша' = T; 'Маша' = T; e._ = F } : T
    = (e.X) e.Name (e.Y);

  e._ = NoName;
}
```

А это безымянная переменная, тоже расширение Рефала-5

Блоки можно использовать после любого результатного выражения, в том числе и в условиях.

Рефал-5λ: расширение Рефала-5

Присваивания — «безоткатные условия». При неудаче сопоставления справа от «=» имеем аварийный останов программы.

```
71  UpdateDriveInfo {
72    s.OptDrive s.OptIntrinsic ((e.KnownNames) e.KnownFunctions) e.AST
73      = <ExtractLabels Drive e.AST> : (e.Drives) e.AST^
74      = <ExtractLabels Inline e.AST> : (e.Inlines) e.AST^
75      = <ExtractLabels Intrinsic e.AST> : (e.Intrinsics) e.AST^
76      = <ExtractMetatableNames e.AST> : (e.Metatables) e.AST^
77
78      = <SetNames-Reject (e.Drives) (e.Intrinsics)> : e.Drives^
79      = <SetNames-Reject (e.Inlines) (e.Intrinsics)> : e.Inlines^
80
81      = <HashSet-AsChain e.KnownNames> : e.KnownNames^
82
83      = <SetNames-Reject (e.Drives) (e.KnownNames)> : e.Drives^
84      = <SetNames-Reject (e.Inlines) (e.KnownNames)> : e.Inlines^
85      = <SetNames-Reject (e.Intrinsics) (e.KnownNames)> : e.Intrinsics^
86      = <SetNames-Reject (e.Metatables) (e.KnownNames)> : e.Metatables^
```

А это сокрытие переменной

Рефал-5λ: расширение Рефала-5

Безымянные вложенные функции (они и дали букву λ в названии диалекта):

```
193 ExtractBaseNames {
194   e.OptNames
195     = <Map
196         {
197           (s.Label e.Name)
198             = <BaseName e.Name> : e.BaseName s.Num
199             = (s.Label e.BaseName);
200         }
201         e.OptNames
202       >
203     : e.BaseOptNames
204     = <Unique e.BaseOptNames>;
205 }
```

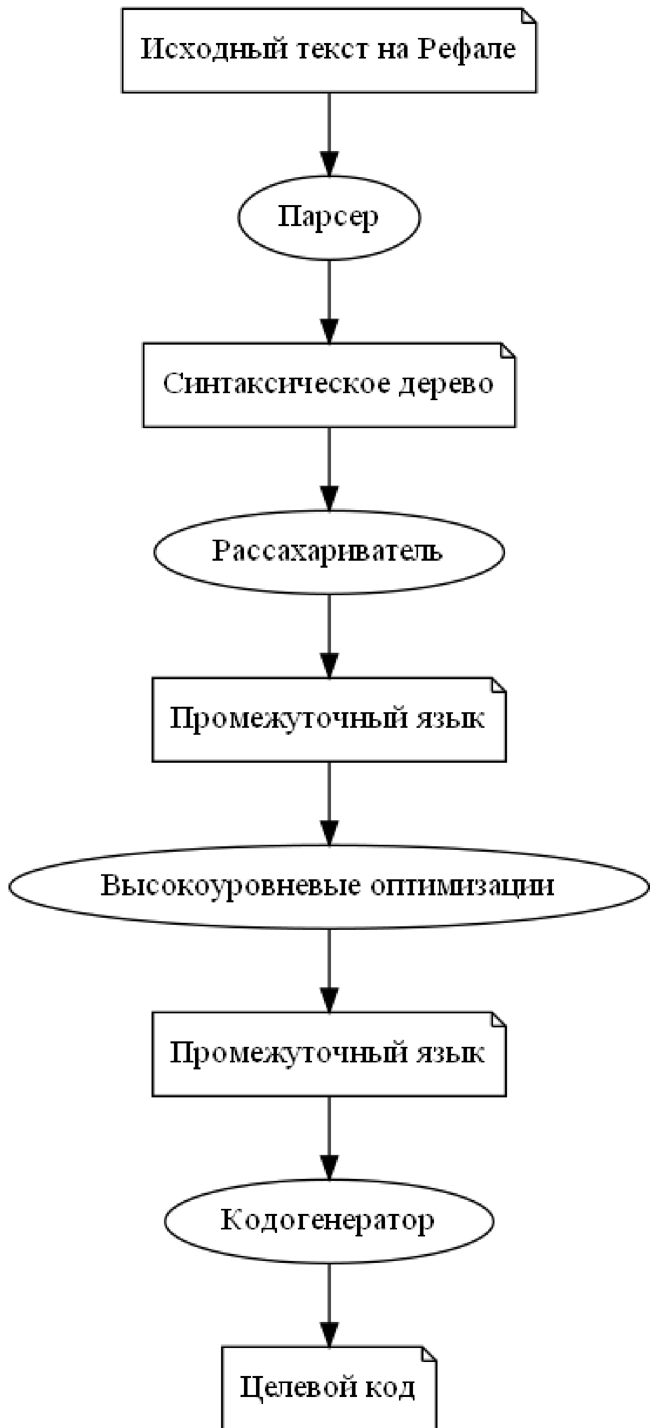
(тут, кстати, есть два присваивания)

Рефал-5λ: расширение Рефала-5

- Абстрактные скобки — инструмент осуществления инкапсуляции. Записываются они как `[FuncName …]`, причём `FuncName` должно быть именем функции.
- Если `FuncName` является именем локальной (не `$ENTRY`) функции, то доступ к содержимому скобок есть только в том файле, где эта функция объявлена.
- В других единицах трансляции терм абстрактных скобок может быть сопоставлен только с t-переменной. Узнать что внутри нельзя за одним исключением — можно только сравнить на равенство две t-переменные.

Рефал-5λ: архитектура компилятора

- Блоки, присваивания, вложенные скобки, безымянные переменные (`e._`) и сокрытия переменных (`e.Var^`) являются синтаксическим сахаром.
- Безымянные и сокрывающие переменные переименовываются.
- Присваивания трактуются как блоки из одного предложения.
- Блоки трактуются как вызовы вложенных безымянных функций.
- Вложенные функции транслируются внутри компилятора в примитивы более низкого уровня.



Рефал-5λ: архитектура компилятора

- Весь этот синтаксический сахар устраняется проходом рассахаривания.
- На входе этого прохода имеем синтаксическое дерево, полученное от парсера, на выходе программу на промежуточном языке.
- Высокоуровневые оптимизации выполняют эквивалентные преобразования на промежуточном языке.

Промежуточный язык

Программа в промежуточном языке представляет собой последовательность глобальных функций.

```
F {
    …предложения…
}

G {
    …предложения…
}
```

Промежуточный язык

Предложения в функциях состоят из образца, результата и нуля или нескольких условий между ними:

```
F {
  образец = результат;
  образец, результат′ : образец′ … = результат;
  …
}
```

Промежуточный язык

- В образцовых выражениях допустимы привычные нам термы Рефала: символы (включая символы-указатели на функции `&Func`), переменные, круглые и квадратные скобки.
- В результатных выражениях допустимы те же термы, что и в образцовых, плюс скобки активации <…> и так называемые *конструкторы замыканий* `{{ … }}`.
- При рассахаривании вложенной функции создаётся вспомогательная глобальная функция, вхождение вложенной функции заменяется либо на указатель на функцию, либо на конструктор замыкания.
- Блоки и присваивания сначала преобразовываются в вызовы вложенных функций, а затем те — во вспомогательные функции.

Промежуточный язык

Исходный текст

```
CartProd {
   (e.X) (e.Y)
     = <Map
          {
            t.X
               = <Map
                    { t.Y = (t.X t.Y) } e.Y
                  >
          }
          e.X
        >
}
```

транслируется в такой промежуточный код:

```
CartProd {
   (e.X) (e.Y)
     = <Map {{ &CartProd\1 (e.Y) }} e.X>;
}

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

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

Промежуточный язык

- Объект замыкания содержит указатель на вспомогательную функцию, реализующую замыкание, и значения захваченных переменных.
- Вспомогательная функция принимает значения захваченных переменных при помощи дополнительных подпараметров, передаваемых в начале аргумента. s- и t-переменные передаются как есть, e-переменные заключаются в скобки.
- Семантика выполнения объекта замыкания может быть описана так:

```
<{{ &Func e.Context }} e.Arg>  →  <Func e.Context e.Arg>
```

- Фактически, конструктор замыкания можно считать конструктором каррирования.

Равенство замыканий

- Объекты замыканий являются символами, т.е. сопоставимы с s-переменными.
- Копирование символов должно выполняться за константное время, поэтому объекты замыканий копируются по ссылке.
- Для управления памятью используется счётчик ссылок.
- Однако, вызов замыкания в общем случае может выполняться не за константу — может потребоваться копирование захваченных переменных в поле зрения.
- Равенство замыканий также должно проверяться за константное время, поэтому они сравниваются на равенство по ссылке.

Равенство замыканий

Инвариант равенства копий. Если два выражения получены путём копирования (подстановки одного и того же значения в кратные переменные в результатном выражении), то они равны (могут быть сопоставлены с кратными переменными в образцовом выражении).

Пример:

```
F { e.X = <Eq (e.X) (e.X)> }

Eq {
  (e.E) (e.E) = True;
  (e.L) (e.R) = False;
}
```

В соответствии с инвариантом функция `F` всегда возвращает `True`.

Равенство замыканий

- В руководстве пользователя определены следующие, довольно слабые требования к равенству замыканий:
  - В соответствии с инвариантом все копии одного и того же замыкания всегда равны.
  - Два замыкания, построенные из вложенных функций, расположенных в исходном тексте по разным координатам, не равны.
  - Два замыкания, построенные из вложенных функций в разных единицах трансляции, не равны. *Пояснение.* В разные единицы трансляции может включаться (`$INCLUDE`) один и тот же файл, который порождает замыкание. Эти порождённые замыкания не равны.
  - Два замыкания, захватывающие неравный контекст, не равны.
  - Остальное не определено.
- Почему требования такие слабые? Потому что если вложенная функция не захватывает переменных, она компилируется не в замыкание с пустым контекстом, а просто в указатель на глобальную функцию.

Оптимизация прогонки

- Впервые прогонка для Рефала была описана Турчиным 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. Стр. 31-42.
- Прогонка по Турчину может быть применена только к подмножеству Рефала, т.н. *ограниченному Рефалу*. Это базисное подмножество, в котором разрешены образцы только определённого вида.

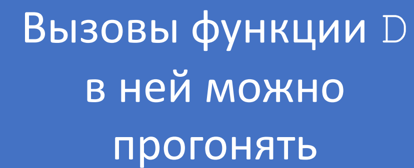
Оптимизация прогонки

Рассмотрим программу.

```
$DRIVE D;

$ENTRY F {
  e.X = <Prout <D e.X>>;
}

D {
  X = A;
  s.X = S;
  t.X = T;
  e.X = E;
}
```



Оптимизация прогонки

Прогоним вызов функции `D`:

```
$DRIVE D;

$ENTRY F {
  X = <Prout A>;
  s.X = <Prout S>;
  t.X = <Prout T>;
  e.X = <Prout E>;
}

D {
  X = A;
  s.X = S;
  t.X = T;
  e.X = E;
}
```

Оптимизация прогонки

Особенности текущей реализации:

- Оптимизируемые функции помечаются ключевым словом `$DRIVE` или `$INLINE`.
- Для функций, помеченных как `$INLINE`, сужения при прогонке запрещены.
- Все безымянные функции (включая вспомогательные функции для блоков и присваиваний) неявно помечены как `$DRIVE`.
- Прогонка применима только к функциям ограниченного Рефала в предложениях без условий и с L-образцами.

Оптимизация специализации

- Будем говорить, что данные известны *статически,* если они известны на стадии компиляции. Если данные известны только во время выполнения, будем говорить, что они известны *динамически.*
- Пусть нам дан вызов `<F ARG>`. *Преобразованием специализации* назовём замену этого вызова на вызов `<F' ARG'>` и построение новой функции `F'` на основе `F` такое, что тело функции `F'` учитывает статически известную информацию из исходного аргумента `ARG`, а новый аргумент `ARG'` эту статически известную информацию не содержит.
- Функцию `F'` будем называть *экземпляром* функции `F`.
- *Сигнатурой* экземпляра будем называть информацию из `ARG`, учтённую при построении экземпляра. Разные вызовы с одной сигнатурой будут вызывать один и тот же экземпляр.
- Простейший случай специализации — вызываемая функция имеет несколько аргументов, один из аргументов в вызове является статическим. В теле экземпляра все вхождения этого параметра заменены на соответствующие константы.

Оптимизация специализации

Особенности текущей реализации

- Для специализируемых функций необходимо явно задавать входной формат и обозначать в нём статические и динамические параметры:
  `$SPEC Map s.FUNC e.arg;`
- Специализация ведётся только по статическим параметрам.
- Экземпляры специализированных функций получают суффиксы `@n`: `Map@1`, `Map@2` и т.д.
- Безымянные функции неявно специализируются по контексту.

Особенности обеих оптимизаций

- Обе оптимизации не умеют работать с ссылочными данными, такими как замыкания.
- Для них конструктор замыкания — это всего лишь ещё один тип скобок `{{ … }}`, который, в отличие от `(…)` и `[X …]`, может сопоставляться с s-переменной и его вызов обрабатывается особым образом.
- Если образец прогоняемой функции имел кратное вхождение s-переменной, а фактическим значением были замыкания, то сопоставление будет успешным, если записи замыканий текстуально равны.
- Собственно, отсюда все проблемы.

Пример работы обеих оптимизаций

Рассмотрим программу

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

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

Пример работы обеих оптимизаций

Эта же программа на промежуточном языке

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

$ENTRY CartProd {
  (e.X) (e.Y) = <Map {{ &CartProd\1 (e.Y) }} e.X>
}

CartProd\1 {
  (e.Y) t.X = <Map {{ &CartProd\1\1 t.X }} e.Y>
}

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

Пример работы обеих оптимизаций

Выполним специализации

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

$ENTRY CartProd { (e.X) (e.Y) = <Map@1 (e.Y) e.X> }

CartProd\1 { (e.Y) t.X = <Map@2 t.X e.Y> }

CartProd\1\1 { t.X t.Y = (t.X t.Y); }

Map@1 {
   (e.Y) t.Next e.Rest
     = <{{ &CartProd\1 (e.Y) }} t.Next> <Map {{ &CartProd\1 (e.Y) }} e.Rest>;

   (e.Y) /* пусто */ = /* пусто */;
}

Map@2 {
  t.X t.Next e.Rest
     = <{{ &CartProd\1\1 t.X }} t.Next> <Map {{ &CartProd\1\1 t.X }} e.Rest>;

  t.X /* пусто */ = /* пусто */;
}
```

Пример работы обеих оптимизаций

Выполним специализации

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

$ENTRY CartProd { (e.X) (e.Y) = <Map@1 (e.Y) e.X> }

CartProd\1 { (e.Y) t.X = <Map@2 t.X e.Y> }

CartProd\1\1 { t.X t.Y = (t.X t.Y); }

Map@1 {
   (e.Y) t.Next e.Rest = <{{ &CartProd\1 (e.Y) }} t.Next> <Map@1 (e.Y) e.Rest>;
   (e.Y) /* пусто */ = /* пусто */;
}

Map@2 {
  t.X t.Next e.Rest = <{{ &CartProd\1\1 t.X }} t.Next> <Map@2 t.X e.Rest>;
  t.X /* пусто */ = /* пусто */;
}
```

Пример работы обеих оптимизаций

Выполним прогонки

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

$ENTRY CartProd { (e.X) (e.Y) = <Map@1 (e.Y) e.X> }

CartProd\1 { (e.Y) t.X = <Map@2 t.X e.Y> }

CartProd\1\1 { t.X t.Y = (t.X t.Y); }

Map@1 {
  (e.Y) t.Next e.Rest = <Map@2 t.Next e.Y> <Map@1 (e.Y) e.Rest>;
  (e.Y) /* пусто */ = /* пусто */;
}

Map@2 {
  t.X t.Next e.Rest = (t.X t.Next) <Map@2 t.X e.Rest>;
  t.X /* пусто */ = /* пусто */;
}
```

Специализация замыканий

- На выходе рассахаривателя конструкторы замыканий содержат захваченные переменные лишь в общем виде:
  `{{ &F\1 (e.X) t.Y s.Z }}`
- Однако, при преобразованиях программ (прогонка, специализация) аргументы конструктора замыкания могут оказаться частично известными:
  `{{ &F\1 ('AB' s.1) (e.2) 'C' }}`
- Возникает соблазн создать для функции `&F\1` специализированный экземпляр, учитывающий статически известную информацию:
  `{{ &F\1@1 s.1 (e.2) }}`
- Такая оптимизация в компиляторе реализована.

Специализация замыканий

Пример. Исходная программа

```
$ENTRY F {
  e.X = <D e.X> { e.Y = e.X e.Y };
}

$DRIVE D;

D {
  s.X = S;
  t.X = T;
  e.X = E;
}
```

Специализация замыканий

После рассахаривания:

```
$ENTRY F {
  e.X = <D e.X> {{ &F\1 (e.X) }};
}

F\1 { (e.X) e.Y = e.X e.Y; }

$DRIVE D;

D {
  s.X = S;
  t.X = T;
  e.X = E;
}
```

Специализация замыканий

После прогонки:

```
$ENTRY F {
  s.X = S {{ &F\1 (s.X) }};
  t.X = T {{ &F\1 (t.X) }};
  e.X = E {{ &F\1 (e.X) }};
}

F\1 { (e.X) e.Y = e.X e.Y; }

$DRIVE D;

D {
  s.X = S;
  t.X = T;
  e.X = E;
}
```

Специализация замыканий

После специализации замыканий:

```
$ENTRY F {
  s.X = S {{ &F\1@1 s.X }};
  t.X = T {{ &F\1@2 t.X }};
  e.X = E {{ &F\1 (e.X) }};
}

F\1 { (e.X) e.Y = e.X e.Y; }

$DRIVE D;

D { … }

F\1@1 { s.X e.Y = s.X e.Y; }
F\1@2 { t.X e.Y = t.X e.Y; }
```

Пример нарушения семантики № 1

Рассмотрим программу

```
$ENTRY Go {
  e.X = <Prout <Eq <Clo e.X> <Clo e.X>>>
}

$INLINE Clo, Eq;

Clo { e.X = { = e.X } }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Неоптимизированная программа напечатает `False`, т.к. два вызова функции `Clo` создадут два разных объекта замыканий по разным адресам. Рассмотрим оптимизированный случай.

Пример нарушения семантики № 1

Программа после рассахаривания:

```
$ENTRY Go {
  e.X = <Prout <Eq <Clo e.X> <Clo e.X>>>
}

$INLINE Clo, Eq;

Clo { e.X = {{ &Clo\1 (e.X) }} }

Clo\1 { (e.X) = e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Пример нарушения семантики № 1

Первые два шага прогонки:

```
$ENTRY Go {
  e.X = <Prout <Eq {{ &Clo\1 (e.X) }} {{ &Clo\1 (e.X) }}>>
}

$INLINE Clo, Eq;

Clo { e.X = {{ &Clo\1 (e.X) }} }

Clo\1 { (e.X) = e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Пример нарушения семантики № 1

При прогонке первого предложения `Eq` будет выполнено сравнение на равенство. Два замыкания будут признаны равными, т.к. текстуально совпадают.

```
$ENTRY Go {
  e.X = <Prout True>
}

$INLINE Clo, Eq;

Clo { e.X = {{ &Clo\1 (e.X) }} }

Clo\1 { (e.X) = e.X }

Eq { … }
```

Оптимизированная программа напечатает `True`. Семантика программы изменилась, но с точки зрения документации это нормально, т.к. поведение программы изначально было не определено.

Пример нарушения семантики № 2

Рассмотрим программу

```
$ENTRY Go {
  e.X = <Prout <Eq <Dup { = e.X }>>>
}

$INLINE Dup;

Dup { e.X = e.X e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Неоптимизированная версия будет печатать `True`, т.к. функция `Dup` создаст копию ссылки, оба символа будут равны. Рассмотрим, как на неё повлияет оптимизация.

Пример нарушения семантики № 2

После рассахаривания и прогонки получим

```
$ENTRY Go {
  e.X = <Prout <Eq {{ &Go\1 (e.X) }} {{ &Go\1 (e.X) }}>>
}

Go\1 { (e.X) = e.X }

$INLINE Dup;

Dup { e.X = e.X e.X }

Eq { … }
```

Оптимизированная версия будет распечатывать `False`, т.к. в аргументе функции `Eq` построятся два отдельных замыкания, ссылки на них будут не равны.

Пример нарушения семантики № 3

Рассмотрим программу

```
$ENTRY Go {
  e.X = <S { = e.X }>;
}

$SPEC S s.STAT;

S { s.X = <Prout <Eq s.X s.X>> }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Неоптимизированная версия напечатает `True`. Посмотрим, что сделает оптимизация.

Пример нарушения семантики № 3

Программа после рассахаривания:

```
$ENTRY Go {
  e.X = <S {{ &Go\1 (e.X) }}>;
}

Go\1 { (e.X) = e.X }

$SPEC S s.STAT;

S { s.X = <Prout <Eq s.X s.X>> }

Eq { … }
```

Функция `S` может быть проспециализирована.

Пример нарушения семантики № 3

Программа после рассахаривания:

```
$ENTRY Go {
  e.X = <S@1 e.X>;
}

Go\1 { (e.X) = e.X }

$SPEC S s.STAT;

S { s.X = <Prout <Eq s.X s.X>> }

Eq { … }

S@1 { e.X = <Prout <Eq {{ &Go\1 (e.X) }} {{ &Go\1 (e.X) }}>> }
```

Оптимизированная версия будет печатать `False`.

Пример нарушения семантики № 4

Рассмотрим программу

```
$ENTRY Go {
  e.X
    = { e.Y = e.X e.Y } : s.Func
    = <Eq s.Func <S s.Func e.X>>;
}

$SPEC S s.FUNC e.ARG;

S { s.F e.X = s.F <Prout <D e.X>>; }

$DRIVE D;

D {
  'abc' = abc;
  'a' e._ = a_;
  e._ = any;
}

Eq {
  s.Eq s.Eq = True;
  s._ s._ = False;
}
```

Пример нарушения семантики № 4

После рассахаривания:

```
$ENTRY Go { e.X = <Go=1 {{ &Go=1\1 (e.X) }}> }

Go=1 { s.Func = <Eq s.Func <S s.Func e.X>> }

Go=1\1 { (e.X) e.Y = e.X e.Y }

$SPEC S s.FUNC e.ARG;

S { s.F e.X = s.F <Prout <D e.X>>; }

$DRIVE D;

D {
  'abc' = abc;
  'a' e._ = a_;
  e._ = any;
}

Eq {
  s.Eq s.Eq = True;
  s._ s._ = False;
}
```

Вызов `Go=1` можно прогнать (вложенные функции неявно имеют метку `$DRIVE`)

Пример нарушения семантики № 4

После прогонки Go=1:

```
$ENTRY Go { e.X = <Eq {{ &Go=1\1 (e.X) }} <S {{ &Go=1\1 (e.X) }} e.X>> }

Go=1 { s.Func = <Eq s.Func <S s.Func e.X>> }

Go=1\1 { (e.X) e.Y = e.X e.Y }

$SPEC S s.FUNC e.ARG;

S { s.F e.X = s.F <Prout <D e.X>>; }

$DRIVE D;

D {
  'abc' = abc;
  'a' e._ = a_;
  e._ = any;
}

Eq {
  s.Eq s.Eq = True;
  s._ s._ = False;
}
```

Пример нарушения семантики № 4

После специализации S:

```
$ENTRY Go { e.X = <Eq {{ &Go=1\1 (e.X) }} <S@1 e.X>> }

Go=1 { s.Func = <Eq s.Func <S s.Func e.X>> }

Go=1\1 { (e.X) e.Y = e.X e.Y }

$SPEC S s.FUNC e.ARG;

S { s.F e.X = s.F <Prout <D e.X>>; }

$DRIVE D;

D {
  'abc' = abc;
  'a' e._ = a_;
  e._ = any;
}

Eq { … }

S@1 { e.X = {{ &Go=1\1 (e.X) }} <Prout <D e.X>> }
```

Пример нарушения семантики № 4

После прогонки `D` внутри экземпляра `S@1`:

```
$ENTRY Go { e.X = <Eq {{ &Go\1 (e.X) }} <S@1 e.X>> }

Go=1 { s.Func = <Eq s.Func <S s.Func e.X>> }

Go=1\1 { (e.X) e.Y = e.X e.Y }

$SPEC S s.FUNC e.ARG;

S { s.F e.X = s.F <Prout <D e.X>>; }

$DRIVE D;

D {
  'abc' = abc;
  'a' e._ = a_;
  e._ = any;
}

Eq { … }

S@1 {
  'abc' =     {{ &Go=1\1 ('abc') }}   <Prout abc>;
  'a' e.1 = {{ &Go=1\1 ('a' e.1) }} <Prout a_>;
  e.X =     {{ &Go=1\1 (e.X) }}     <Prout any>;
}
```

Замыкания `Go=1\1` можно проспециализировать

Пример нарушения семантики № 4

После специализации замыканий внутри `S@1`:

```
$ENTRY Go { e.X = <Eq {{ &Go\1 (e.X) }} <S@1 e.X>> }

Go=1\1 { (e.X) e.Y = e.X e.Y }

$SPEC S s.FUNC e.ARG;

S { s.F e.X = s.F <Prout <D e.X>>; }

$DRIVE D;

D {
  'abc' = abc;
  'a' e._ = a_;
  e._ = any;
}

Eq { … }

S@1 {
  'abc'    =    &Go=1\1@1          <Prout abc>;
  'a' e.1  = {{ &Go=1\1@2 (e.1) }} <Prout a_>;
  e.X      = {{ &Go=1\1 (e.X) }}   <Prout any>;
}

Go=1\1@1 { e.Y = 'abc' e.Y }
Go=1\1@2 { (e.1) e.Y = 'a' e.1 e.Y }
```

У новых замыканий изменилось их внутреннее содержимое. У первого замыкания даже изменился тип (с объекта замыканя на указатель на функцию).

Про фазы луны

Рассмотрим программу

```
$EXTERN MoonPhase;

$ENTRY RandomCall {
  s.F s.G
    , s.F s.G : s.Eq s.Eq  /* s.F и s.G равны */
    = <MoonPhase>
    : {
        '+' = <s.F>;       /* луна растёт */
        '-' = <s.G>;       /* луна убывает */
      };

  s.F s.G = "!=";
}
```

При правильном описании семантики результат работы `RandomCall` не должен зависеть от фазы луны.

Простые неудачные решения

- Запретить сравнение замыканий на равенство вообще. При попытке сравнения на равенство программа аварийно останавливается.
  - ❌ Абсолютно контринтуитивно.
- Считать все замыкания не равными друг другу.
  - ❌ Нарушится инвариант: копии значений будут не равны друг другу.
- Считать все замыкания равными.
  - ❌ Программа будет зависеть от фаз луны.
- Сравнивать замыкания не по ссылке, а по значению.
  - ❌ s-переменные будут сравниваться не за константное время.
  - ❌ Пример № 4 работать всё равно не будет.
- Вообще не давать никаких гарантий равенства.
  - ❌ Неопределённого поведения в высокоуровневых языках лучше избегать.

Решение № 1: создание замыкания в два этапа

Создание замыкания будет выполняться в два этапа:

- Создание пустого объекта замыкания (пустого динамического ящика). В псевдокоде операцию создания ящика обозначать ключевым словом `$NEW`.
- Присвоение этому динамическому ящику значения замыкания. Такой конструктор замыкания будем обозначать

```
{{ s.R := &F\1 e.Content }}
```

- При прогонке сравниваются на равенство переменные перед `:=`.
- Семантика вызова конструктора замыкания остаётся неизменной:

```
<{{ s.R := &F\1 e.Cnt }} e.Arg> → <F\1 e.Cnt e.Arg>
```

Решение № 1: создание замыкания в два этапа

- «Старый» конструктор замыкания `{{ &F\1 e.Cnt }}` каждый раз создавал новую ссылку. Поэтому копирование при прогонке приводило к созданию новой ссылки.
- «Новый» конструктор `{{ s.R := &F\1 e.Cnt }}` инициализирует значение ссылки. Копирование при прогонке приведёт лишь к повторной инициализации (тем же) значением.
- Прогонщик сравнивает два «старых» конструктора замыкания на равенство по содержимому — это было решение ad hoc.
- Два новых замыкания будут сравниваться по ссылкам.

Решение № 1:
создание замыкания в два этапа

Рассмотрим пример

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

$ENTRY Scale {
  e.Values s.Scale
    = <Map { s.X = <Mul s.X s.Scale> } e.Values>;
}
```

.

Решение № 1: создание замыкания в два этапа

Эта программа рассахарится так:

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

$ENTRY Scale {
  e.Values s.Scale
    , $NEW : s.R
    = <Map {{ s.R := &Scale\1 s.Scale }} e.Values>;
}

Scale\1 { s.Scale s.X = <Mul s.X s.Scale> }
```

Решение № 1: создание замыкания в два этапа

Аргументами экземпляра функции Map будут `s.R` и `s.Scale`:

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

$ENTRY Scale {
  e.Values s.Scale, $NEW : s.R = <Map@1 s.R s.Scale e.Values>;
}

Scale\1 { s.Scale s.X = <Mul s.X s.Scale> }

Map@1 {
  s.R s.Scale t.Next e.Rest
    = <{{ s.R := &Scale\1 s.Scale }} t.Next> <Map {{ s.R := &Scale\1 s.Scale }} e.Rest>;
  s.R s.Scale /* пусто */ = /* пусто */;
}
```

Решение № 1: создание замыкания в два этапа

Первый вызов в `Map@1` можно прогнать, второй специализировать:

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

Map {
  s.Func t.Next e.Rest = <s.Func t.Next> <Map s.Func e.Rest>;
  s.Func /* пусто */ = /* пусто */;
}

$ENTRY Scale {
  e.Values s.Scale, $NEW : s.R = <Map@1 s.R s.Scale e.Values>;
}

Scale\1 { s.Scale s.X = <Mul s.X s.Scale> }

Map@1 {
  s.R s.Scale s.X e.Rest = <Mul s.X s.Scale> <Map@1 s.R s.Scale e.Rest>;
  s.R s.Scale /* пусто */ = /* пусто */;
}
```

Решение № 1: создание замыкания в два этапа

```
…

$ENTRY Scale {
  e.Values s.Scale, $NEW : s.R = <Map@1 s.R s.Scale
e.Values>;
}

Map@1 {
  s.R s.Scale s.X e.Rest = <Mul s.X s.Scale> <Map@1 s.R
s.Scale e.Rest>;
  s.R s.Scale /* пусто */ = /* пусто */;
}
```

Очевидный недостаток полученной программы — создаётся новый динамический ящик `s.R`, который передаётся на каждой итерации и никак не используется.

Для его удаления потребуется нетривиальный анализ программы.

Решение № 1: создание замыкания в два этапа

Покажем, что пример № 1 теперь работает правильно:

```
$ENTRY Go {
  e.X = <Prout <Eq <Clo e.X> <Clo e.X>>>
}

$INLINE Clo, Eq;

Clo { e.X = { = e.X } }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Решение № 1: создание замыкания в два этапа

Рассахариваем:

```
$ENTRY Go {
  e.X = <Prout <Eq <Clo e.X> <Clo e.X>>>
}

$INLINE Clo, Eq;

Clo { e.X, $NEW : s.R = {{ s.R := &Clo\1 (e.X) }}; }

Clo\1 { (e.X) = e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Решение № 1: создание замыкания в два этапа

Прогоняем вызовы `Clo`:

```
$ENTRY Go {
  e.X
    , $NEW : s.R1, $NEW : s.R2
    = <Prout <Eq {{ s.R1 := &Clo\1 (e.X) }} {{ s.R2 := &Clo\1 (e.X) }}>>
}

$INLINE Clo, Eq;

Clo { e.X, $NEW : s.R = {{ s.R := &Clo\1 (e.X) }}; }

Clo\1 { (e.X) = e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Решение № 1: создание замыкания в два этапа

Прогоняем вызов `Eq`. При прогонке функции `Eq` будут сравниваться `s.R1` и `s.R2`. Они не равны, следовательно вызов `Eq` заменится на `False`

```
$ENTRY Go {
  e.X, $NEW : s.R1, $NEW : s.R2 = <Prout False>
}

$INLINE Clo, Eq;

Clo { e.X, $NEW : s.R = {{ s.R := &Clo\1 (e.X) }}; }

Clo\1 { (e.X) = e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Поведение программы не изменилось.

Решение № 1:
создание замыкания в два этапа

Покажем, что пример № 2 тоже работает правильно:

```
$ENTRY Go {
  e.X = <Prout <Eq <Dup { = e.X }>>>
}

$INLINE Dup;

Dup { e.X = e.X e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Решение № 1: создание замыкания в два этапа

Рассахариваем:

```
$ENTRY Go {
  e.X, $NEW : s.R = <Prout <Eq <Dup {{ s.R := &Go\1 (e.X) }}>>>
}

Go\1 { (e.X) = e.X }

$INLINE Dup;

Dup { e.X = e.X e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Решение № 1: создание замыкания в два этапа

Прогоняем вызов `Dup`:

```
$ENTRY Go {
  e.X
    , $NEW : s.R
    = <Prout <Eq {{ s.R := &Go\1 (e.X) }}{{ s.R := &Go\1 (e.X) }}>>
}

Go\1 { (e.X) = e.X }

$INLINE Dup;

Dup { e.X = e.X e.X }

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Решение № 1: создание замыкания в два этапа

```
$ENTRY Go {
  e.X
    , $NEW : s.R
    = <Prout <Eq {{ s.R := &Go\1 (e.X) }} {{ s.R := &Go\1 (e.X) }}>>
}

…

Eq {
  s.X s.X = True;
  s.X s.Y = False;
}
```

Замыкания будут равны во время выполнения, т.к. будут представлены одной и той же ссылкой `s.R`.

Решение № 1: создание замыкания в два этапа

- Нетрудно убедиться, что пример № 3 тоже будет работать правильно.
- Пример № 4 будет работать правильно, если разрешить создавать объекты замыканий с пустым контекстом:

```
$ENTRY Go {
  e.X, $NEW : s.R = <Eq {{ s.R := &Go\1 (e.X) }} <S@1 s.R e.X>>
}

Go=1\1 { (e.X) e.Y = e.X e.Y }
…

S@1 {
  s.R 'abc' =    {{ s.R := &Go=1\1@1 }}         <Prout abc>;
  s.R 'a' e.1 =  {{ s.R := &Go=1\1@2 (e.1) }}  <Prout a_>;
  s.R e.X =      {{ s.R := &Go=1\1 (e.X) }}    <Prout any>;
}

Go=1\1@1 { e.Y = 'abc' e.Y }
Go=1\1@2 { (e.1) e.Y = 'a' e.1 e.Y }
```

Решение № 1: создание замыканий в два этапа

- ✔ Обеспечивает сохранение поведения программы.
- ✔ Сохраняет работоспособными все три оптимизации.
- ✔ Сохраняет обратную совместимость (замыкания как s-переменные и ссылочные типы).
- ✖ В специализированных экземплярах будут передаваться дополнительные бесполезные параметры.
- ✖ Требует усложнения компилятора:
  - расширение синтаксиса промежуточного языка,
  - усложнение логики древесных оптимизаций,
  - поддержка новых конструкций на уровне генератора кода и рантайма.

В примерах псевдокода операции `$NEW` для наглядности были записаны как условия. Но с точки зрения и логики, и реализации, это не условия, это часть результатного выражения.

Адекватной была бы конструкция $\nu\ r_1 \ldots r_N . expr$, использованная Анд. В. Климовым на предыдущем семинаре. (Анд. В. Климов взял это обозначение у школы Эндрю Питтса).

Решение № 2: отказаться от ссылочных типов

- Если ссылочные типы нам доставляют такие неудобства, почему бы нам от них не отказаться?
- Тогда замыкания должны будут сравниваться по значению, что контринтуитивно: s-переменные должны сравниваться за малое константное время.
- А почему вообще замыкания должны быть s-переменными?
- s-переменные — значения, которые в Рефале невозможно разбить на составные части при помощи сопоставления с образцом.
- Но в языке есть и t-значения, которые в некоторых случаях тоже нельзя разбить на составные части — абстрактные скобки.
- Что получится, если мы сделаем замыкания квадратными скобками?

Решение № 2: отказаться от ссылочных типов

- Сейчас конструктор замыкания имеет вид `{{ &F\1 e.Context }}`, где `F\1` — имя неявно сгенерированной функции.
- Предлагается вместо конструктора замыкания создавать абстрактный терм `[F\1 e.Context]`.
- Семантика вызова функции будет иметь вид

```
<[F\1 e.Cnt] e.Arg>  →  <F\1 e.Cnt e.Arg>
```

- Функция `F\1` локальная и неявно сгенерированная. Пользователь не сможет разобрать замыкание на составные части, оно сможет быть сопоставлено только с t-переменной (в том числе и повторной).

Решение № 2: отказаться от ссылочных типов

- Нетрудно показать, что примеры № 1, № 2 и № 3 не будут менять своё поведение при включённых оптимизациях. Действительно, там будут создаваться скобочные термы с идентичным содержимым, которые будут равны и во время компиляции (при прогонке) и во время выполнения.
- Пример № 4 работать не будет — если будет выполнена специализация замыкания, то вместо терма
  `[Go=1\1 ('a' e.1)]`
  будет построен терм
  `[Go=1\1@2 (e.1)]`
  который будет заведомо не равен исходному.
- Специализация замыканий ставит и другой вопрос. Если пользователь объявил непустую локальную функцию `F` и стал использовать её имя для АТД-термов, должен ли компилятор эти термы преобразовывать. Понятно, что не должен. Либо должен, только если пользователь пометит функцию `F` как `$SPEC`.

Решение № 2: отказаться от ссылочных типов

- ✔ Существенное упрощение семантики языка и реализации компилятора:
  - ссылочных типов нет,
  - меньше понятий на уровне языка — меньше кода в компиляторе,
  - рантайм не должен подсчитывать ссылки на объекты.
- ✔ Не требуется расширять компилятор и рантайм новыми средствами.
- ✔ Сравнительно просто реализовать именованные (взаимно) рекурсивные вложенные функции — они будут упрощаться на стадии рассахаривания.
  В противоположность, в актуальной реализации и в рамках решения № 1 взаимнорекурсивные именованные вложенные функции реализовать довольно сложно, т.к. используется подсчёт ссылок.
- ✕ Теряется обратная совместимость.
- ✕ Одно средство языка используется для двух существенно разных целей. Своего рода «ложка-вилка», которая и как ложка неудобна, и как вилка.
- ✕ Специализация замыканий всё равно нарушает семантику. Придётся или пожертвовать этой оптимизацией, или смириться с тем, что проблема не решена.

Выводы

- Эквивалентные трансформации программ гораздо проще реализовать для типов-значений, нежели для ссылочных типов.
- В докладе были показаны проблемы, которые возникают в наивной реализации трёх оптимизаций при использовании ссылочных типов.
- Предложено два варианта решения этих проблем, каждый из которых имеет свои преимущества и недостатки.
- Для данной проблемы есть заявка на GitHub: Древесные оптимизации нарушают семантику · Issue #276 · bmstu-iu9/refal-5-lambda (github.com)