Декомпозиция вызовов функций во время суперкомпиляции путём построения выходных форматов

Александр В. Коновалов
МГТУ имени Н.Э. Баумана, Москва

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

8 июня 2021

Композиция функций при суперкомпиляции

- Известно, что суперкомпиляция неплохо справляется с вычислением композиции функций. Например, для композиции функций `<F <G …>>` исходной программы в остаточной программе может быть построена одна функция `<FG …>`.
- При этом промежуточные структуры данных, порождённые вызовом `<G …>` и потреблённые вызовом `<F …>`, в остаточной программе даже могут не строиться.
- Разновидность суперкомпиляции, нацеленная на оптимизацию подобных композиций, называется *дефорестацией* (Wadler, 1990).
- На вычислении подобной композиции основаны некоторые способы верификации программы при помощи суперкомпиляции.

Композиция функций при суперкомпиляции

- Есть более хитрые стратегии суперкомпиляции, позволяющие осуществлять *таплинг* — слияние нескольких параллельных вызовов в конфигурации, разделяющее общие переменные (Pettorossi, 1984, Chin, 1993, Secher, 2001).
- Например, если в конфигурации есть пара вызовов `<F e.X> … <G e.X>`, в остаточной программе для них построится один вызов `<FG e.X>`, вычисляющий пару значений:

  `<FG e.X> ≡ [<F e.X>, <G e.X>]`
- Подобное преобразование позволяет в некоторых случаях снижать порядок алгоритма.

Декомпозиция функций при суперкомпиляции?

- В классической статье (Burstall, Darlington, 1977) описывается набор правил для эквивалентной трансформации программ.
- Правила, описываемые в статье обратимые. Если программу P_2 можно получить из P_1 путём эквивалентных трансформаций, то можно сделать и обратное преобразование из P_2 в P_1.
- Для того, чтобы преобразовывать программы автоматически, нужна *стратегия* применения этих правил.
- Суперкомпиляция является одной из таких стратегий.

Декомпозиция функций при суперкомпиляции?

- Если исходная программа P_1 содержала композицию `<F <G …>>`, а полученная из неё эквивалентными преобразованиями, P_2 — один вызов `<FG …>`, то применив преобразования в обратном порядке, вызов `<FG …>` можно разложить на композицию `<F <G …>>`.
- Преобразование вызова <F …> в эквивалентную композицию <F′ <F″ …>> мы будем называть *декомпозицией*.
- Можно ли научить суперкомпиляцию делать декомпозицию?

Соглашение о языке

- В дальнейших примерах мы будем рассматривать программы на базисном Рефале в обозначениях Рефала-5.
- Так как при суперкомпиляции обычно повышается местность входной среды, удобно расширить модельный язык функциями с несколькими параметрами.
- Пустые выражения мы будем обозначать как ε.
- Отдельные параметры функций мы будем разделять запятыми

```
<F e.X, e.Y, e.Z>
```

Форматы функций

- Обозначим RANGE(`<F e.X>`) — область допустимых значений функции (конфигурации) `<F e.X>`, RANGE(`<F e.X>`) — множество объектных выражений.
- Пусть существует такое жёсткое выражение `OutF`, что
  ∀*oe* ∈ RANGE(`<F e.X>`).∃S.*oe* = `OutF` // S.
- Тогда `OutF` мы будем называть выходным форматом функции (конфигурации) `<F e.X>`.
- Количество переменных в `OutF` мы будем называть коместностью (коарностью) `<F e.X>`, сами переменные — выходными параметрами.

Форматы функций

- Если нам известен формат функции <F e.X>, то мы можем определить две вспомогательные функции

```
<fmt vars(OutF)> :: OutF
<unfmt OutF> :: vars(OutF).
```

- Функция `<unfmt OutF>` принимает формат и извлекает из него кортеж выходных параметров. Назовём её *функцией расформатирования*.

```
unfmt { OutF = vars(OutF) }
```

- Функция `<fmt vars(OutF)>` принимает кортеж выходных параметров и строит значение выходного формата функции. Назовём её *функцией форматирования*.

```
fmt { vars(OutF) = OutF }
```

- Очевидно, верно тождество

```
<F e.X>  ≡  <fmt <unfmt <F e.X>>>
```

Форматы функций

- В общем случае в формате может быть несколько переменных, а значит, функция может возвращать несколько результатов, формально — некоторый кортеж.
- В последующих примерах такие функции рассматриваться не будут для упрощения изложения.
- Но рассмотренные методы успешно обобщаются и на остальные случаи.

Форматы функций и суперкомпиляция

- Пусть известен выходной формат конфигурации `OutF`.
- Для данного формата мы можем определить функции форматирования `fmt` и расформатирования `unfmt`.
- Тогда мы можем конфигурацию `C` заменить на конфигурацию `<fmt <unfmt C>>`.
- Затем, это конфигурацию можно разбить на две `let out := <unfmt C> in <fmt out>`
- Конфигурации `<unfmt C>` и `<fmt out>` анализируются отдельно.
- Конфигурация `<unfmt C>` даст в остаточной программе функцию, в выходном формате которой отсутствуют избыточные конструкторы.
- Конфигурация `<fmt out>` навесит эти конструкторы на результат.

Онлайн-вычисление форматов конфигураций

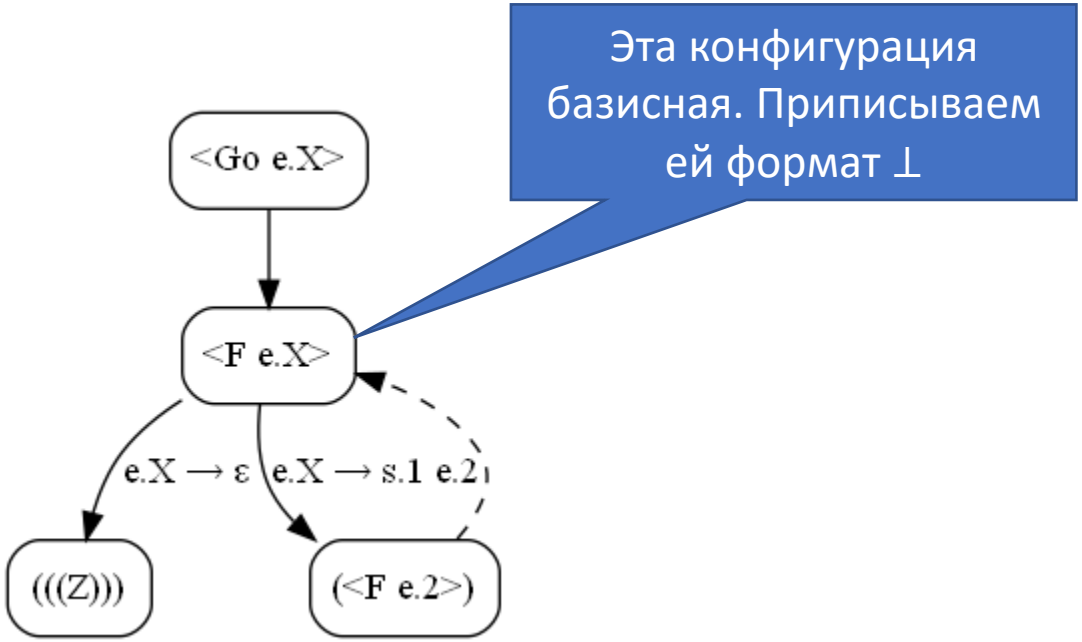
- Форматы базисных конфигураций можно вычислять онлайн.
- Когда конфигурация определяется как базисная (сработало условие остановки прогонки), ей приписывается тривиальный формат $\perp$, соответствующий функции, которая никогда не останавливается.
- При обнаружении выхода из рекурсии сверяется гипотетический выходной формат с фактическим.
- Если результат не вкладывается, происходит перестройка — форматная гипотеза ослабляется (обобщается).

Онлайн-вычисление форматов конфигураций. Пример

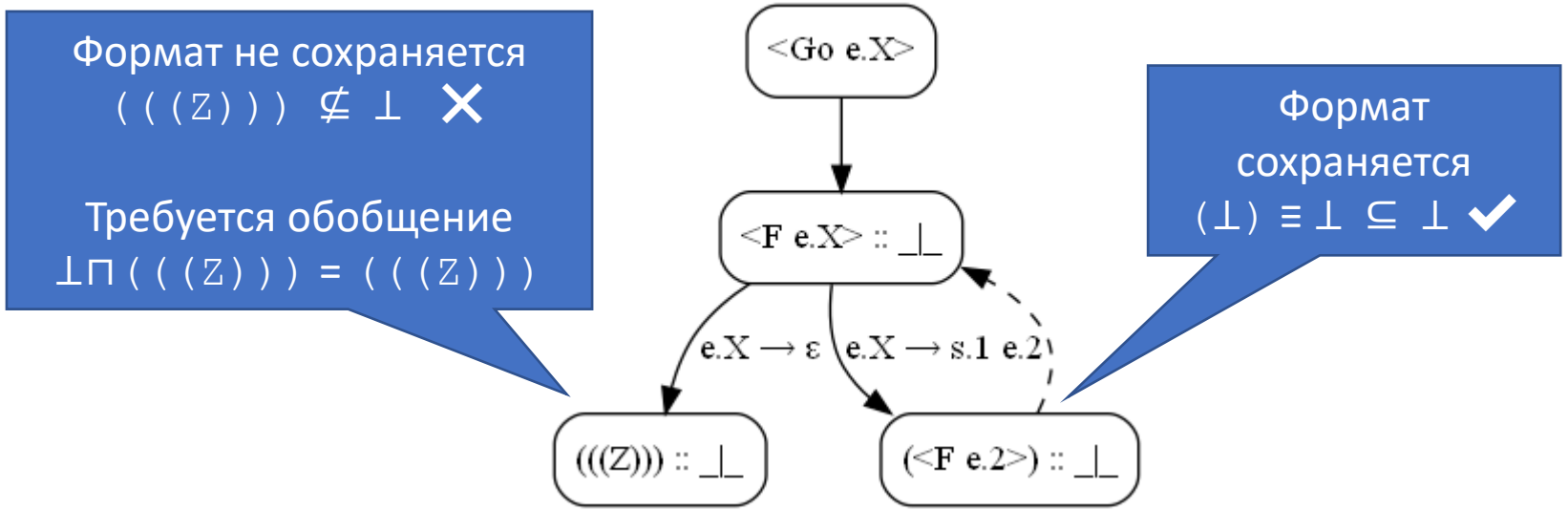
- Рассмотрим пример. Пусть дана следующая программа:

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

F {
  ε         =  (((Z)));
  s.1 e.2 = (<F e.2>);
}
```

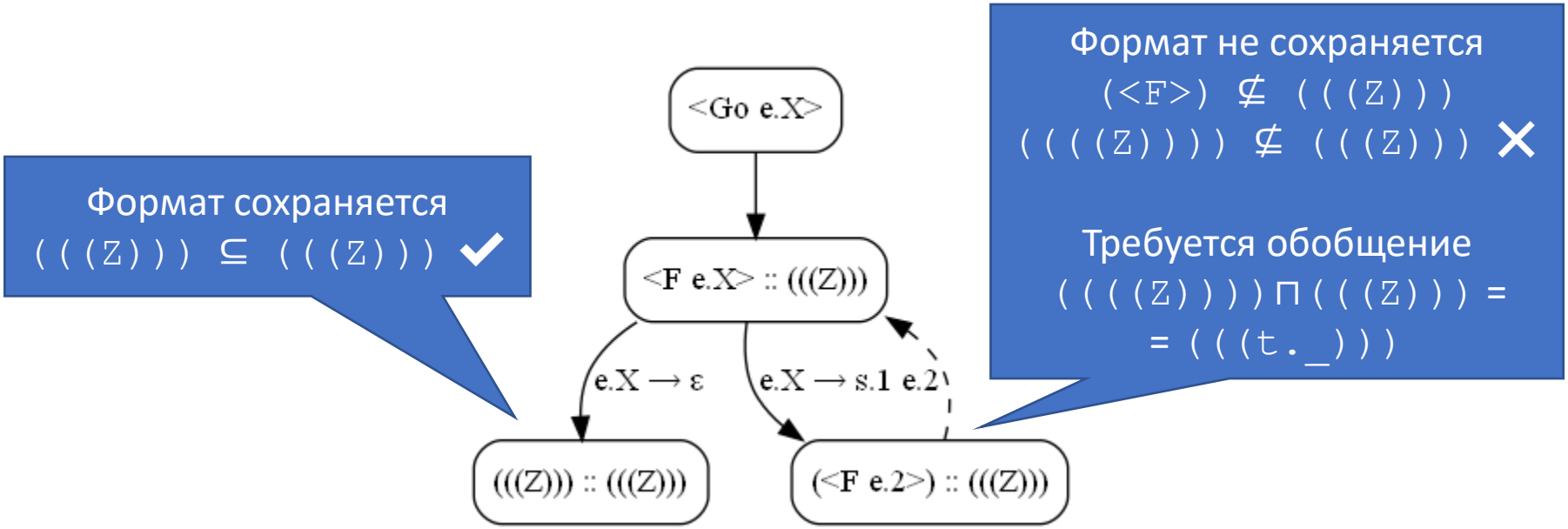
Онлайн-вычисление форматов конфигураций. Пример



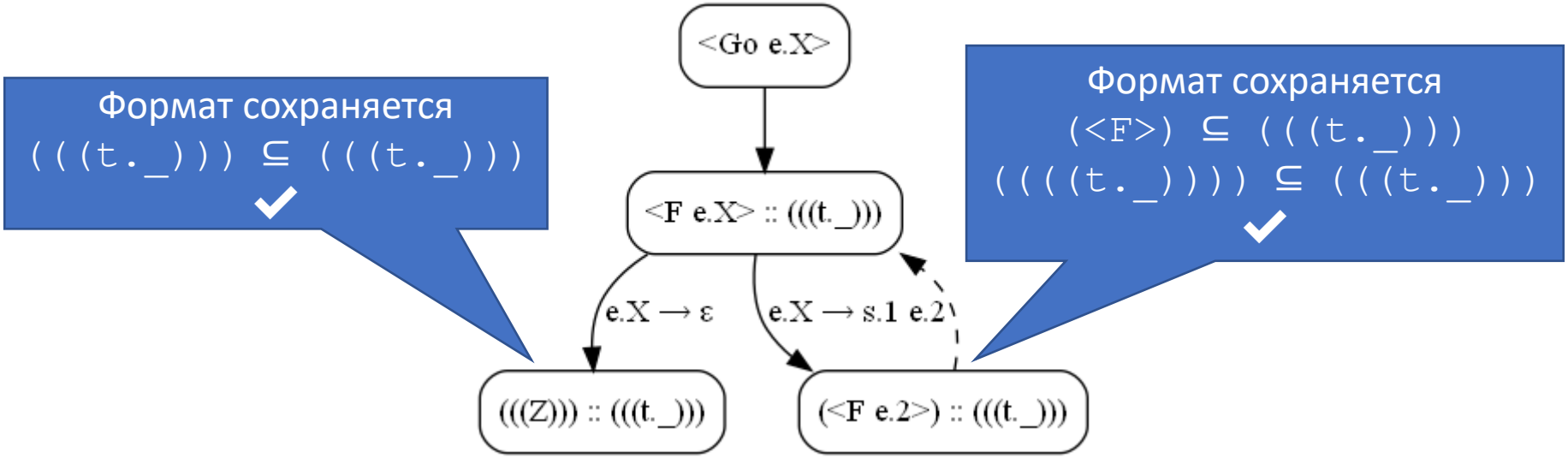
Онлайн-вычисление форматов конфигураций. Пример



Онлайн-вычисление форматов конфигураций. Пример



Онлайн-вычисление форматов конфигураций. Пример



Онлайн-вычисление форматов конфигураций

- Суперкомпилятор SCP4 умеет вычислять выходные форматы конфигураций, однако делает это офлайн, т.е. после завершения построения компоненты факторизации.

- В последующих примерах будет предполагаться, что форматы базисных конфигураций уже вычислены.

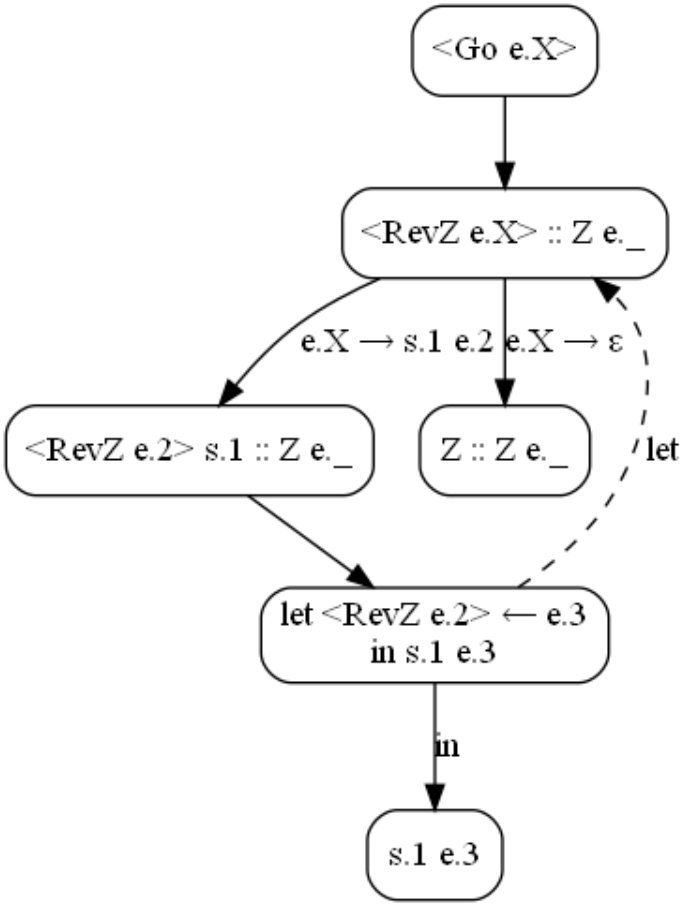
Пример использования форматов в суперкомпиляции

- Рассмотрим программу

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

RevZ {
   s.F e.X = <RevZ e.X> s.F;
   ε       = Z;
}
```

- Программа обращает строку и добавляет в её начало символ `Z`.

Пример использования форматов в суперкомпиляции




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

/* <RevZ e_> :: Z e._ */
RevZ {
   s.F e.X
      = <RevZ e.X> s.F;

   ε = Z;
}
```

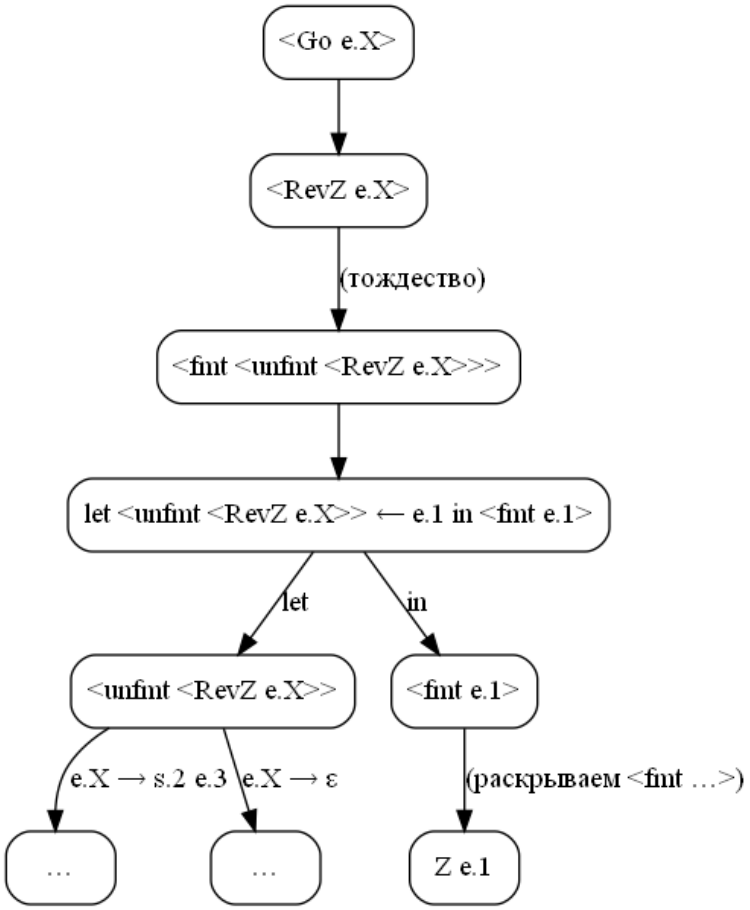
Пример использования форматов в суперкомпиляции

- Форматные функции для формата Z e._:

```
fmt {
  e.1 = Z e.1;
}

unfmt {
  Z e.1 = e.1;
}
```

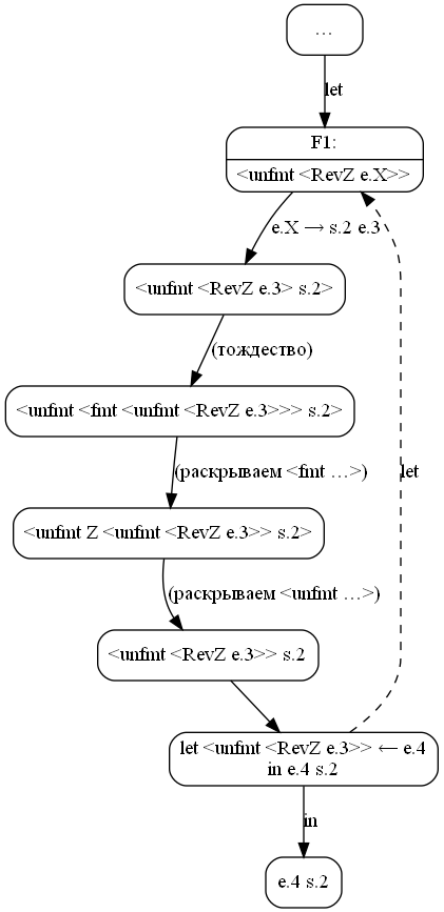
- Добавим их в граф суперкомпиляции

Пример использования форматов в суперкомпиляции



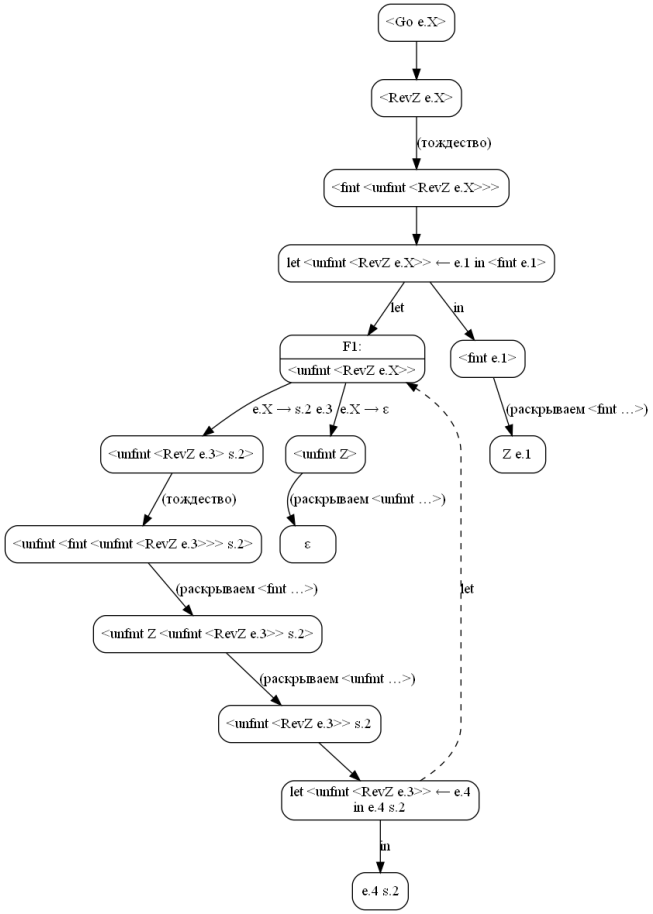

```
$ENTRY Go {
  e.X = <RevZ e.X>;
}

/* <RevZ e._> :: Z e._ */
RevZ {
  s.F e.X
    = <RevZ e.X> s.F;
  ε = Z;
}

fmt {
  e.1 = Z e.1;
}

unfmt {
  Z e.1 = e.1;
}
```

Пример использования форматов в суперкомпиляции




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

/* <RevZ e._> :: Z e._ */
RevZ {
  s.F e.X
    = <RevZ e.X> s.F;
  ε = Z;
}

fmt {
  e.1 = Z e.1;
}

unfmt {
  Z e.1 = e.1;
}
```

Пример использования форматов в суперкомпиляции




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

/* <RevZ e._> :: Z e._ */
RevZ {
   s.F e.X
      = <RevZ e.X> s.F;
   ε = Z;
}

fmt {
   e.1 = Z e.1;
}

unfmt {
   Z e.1 = e.1;
}
```

Пример использования форматов в суперкомпиляции

- Остаточная программа

```
$ENTRY Go {
  e.X = Z <F1 e.X>;
}

F1 {
  s.2 e.3 = <F1 e.3> s.2;
  ε       = ε;
}
```

- Конструктор `Z` из формата функции был вынесен наружу.

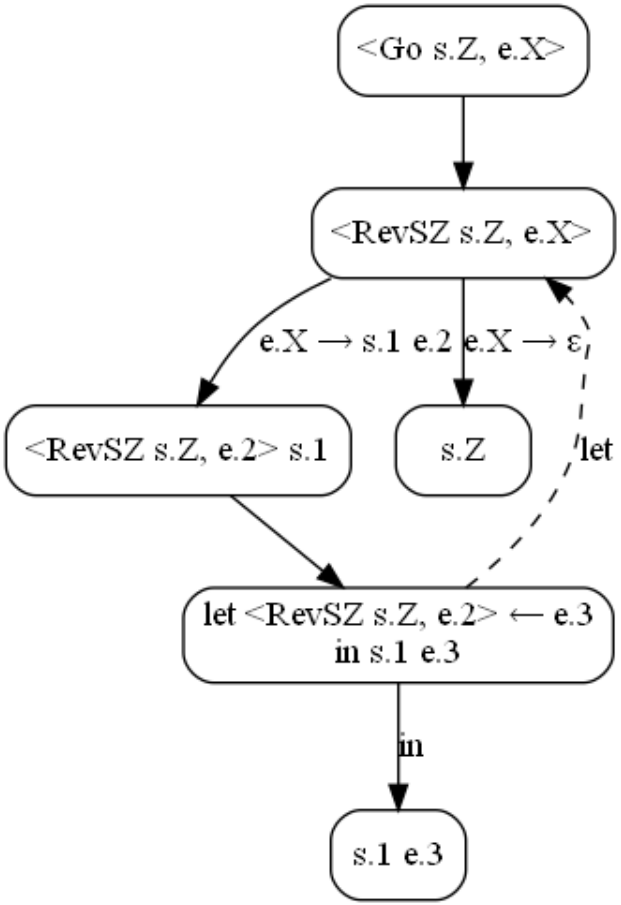
Форматы могут быть параметризованными

- Рассмотрим программу

```
$ENTRY Go {
   s.Z, e.X = <RevSZ s.Z, e.X>;
}

RevSZ {
   s.Z, s.F e.X = <RevSZ s.Z, e.X> s.F;
   s.Z, ε       = s.Z;
}
```

- Программа обращает строку и добавляет в её начало указанный символ.

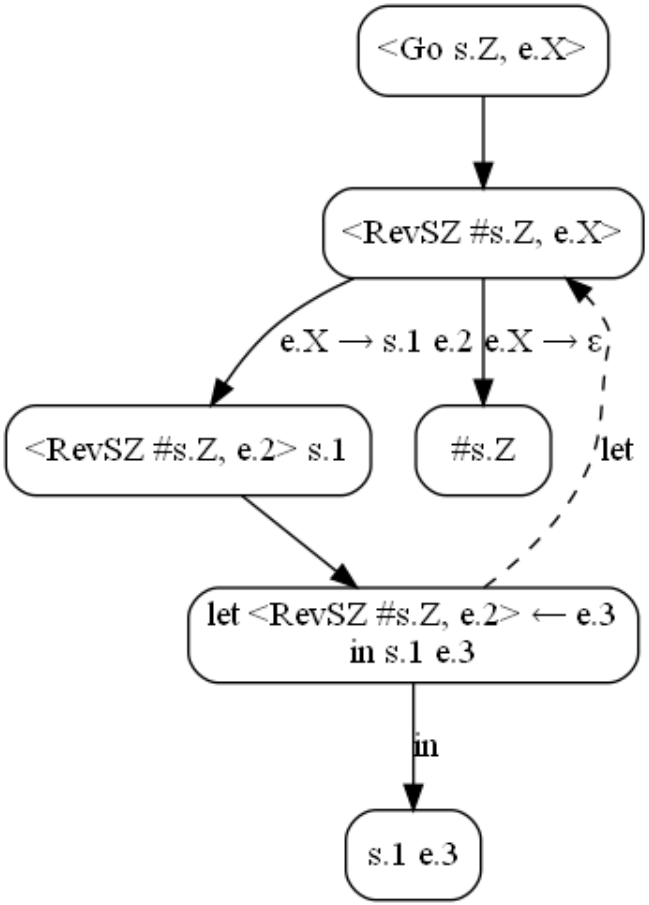
Форматы могут быть параметризованными




```
$ENTRY Go {
   s.Z, e.X
     = <RevSZ s.Z, e.X>;
}

RevSZ {
   s.Z, s.F e.X
     = <RevSZ s.Z, e.X> s.F;
   s.Z, ε = s.Z;
}
```

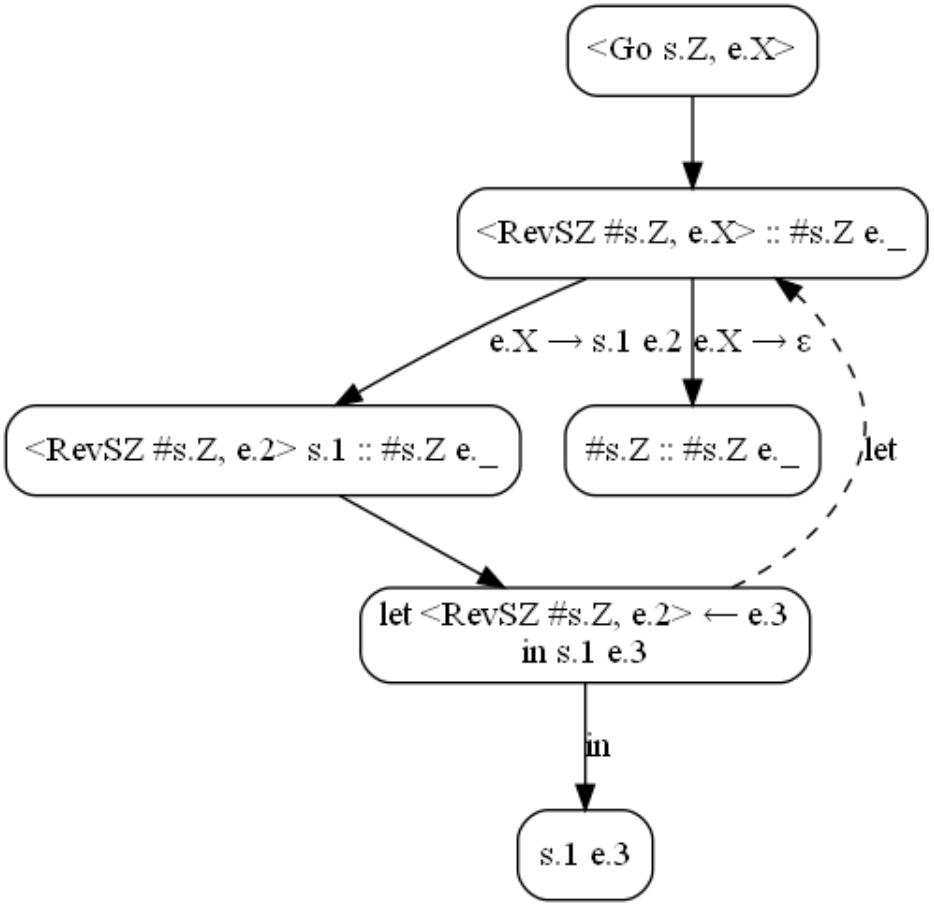
- Заметим, что параметр `s.Z` в вычислениях не участвует.
- Он возвращается неизменным в точке выхода из рекурсии.

Форматы могут быть параметризованными



- Пометим параметр `s.Z` знаком `#`.
- Будем считать, что параметры, помеченные `#`, в некоторой части графа суперкомпиляции рассматриваются как некоторые константы.
- В частности, здесь `#s.Z` — некоторая константа.
- Но, если `#s.Z` — константа, то её можно вынести в формат…

Форматы могут быть параметризованными



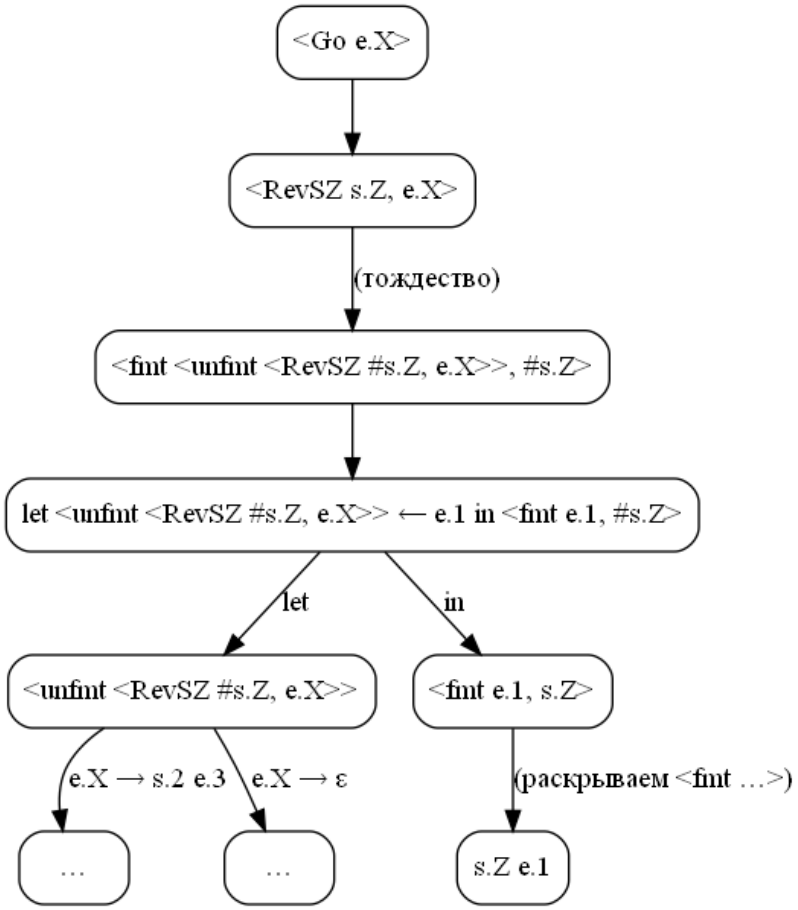
Форматы могут быть параметризованными

- Введём форматные функции для формата `#s.Z e._`:

```
fmt {
  e.1, s.Z = s.Z e.1;
}

unfmt {
  s.Z e.1 = e.1;
}
```

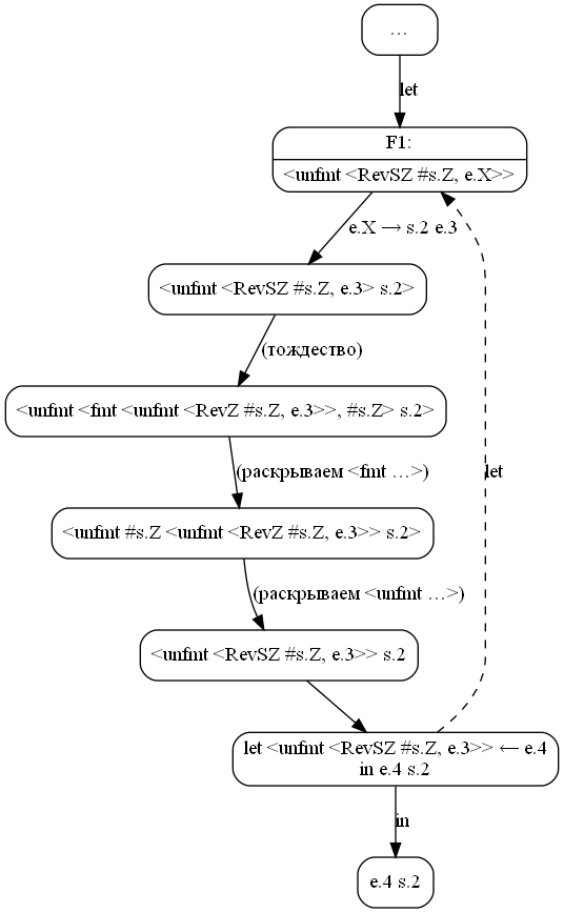
- Заметим, что функция `fmt` принимает дополнительный параметр.

Форматы могут быть параметризованными



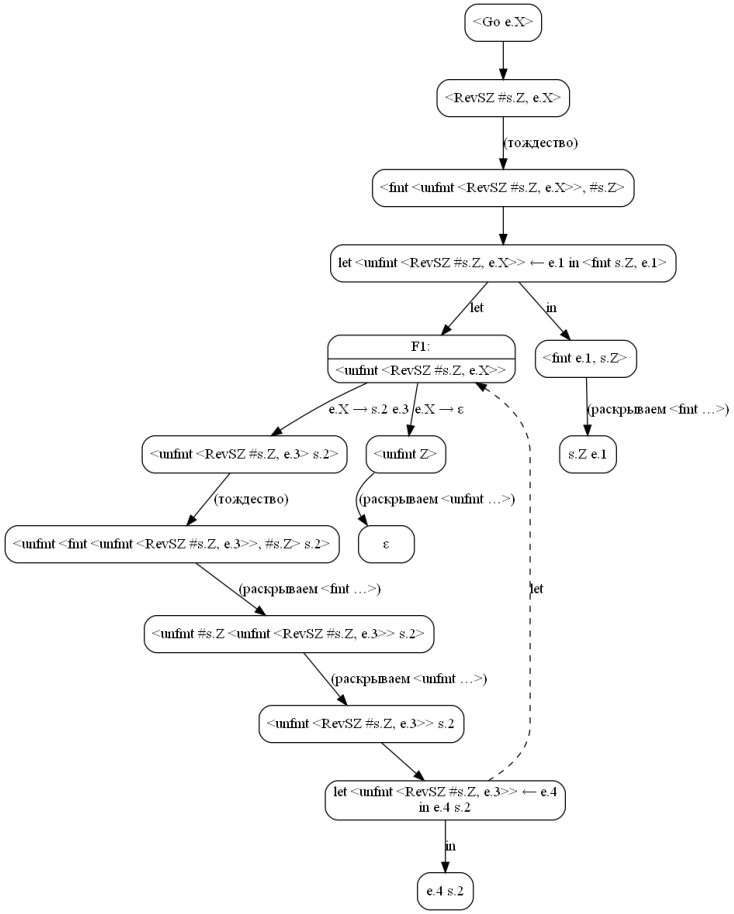

```
$ENTRY Go {
  s.Z, e.X
    = <RevSZ s.Z, e.X>;
}

* <RevSZ #s.Z, e._> :: #s.X e._
RevSZ {
  s.Z, s.F e.X
    = <RevSZ s.Z, e.X> s.F;
  s.Z, ε = s.Z;
}

fmt {
  e.1, s.Z = s.Z e.1;
}

unfmt {
  s.Z e.1 = e.1;
}
```

Форматы могут быть параметризованными




```
$ENTRY Go {
  s.Z, e.X
    = <RevSZ s.Z, e.X>;
}

* <RevSZ #s.Z, e._> :: #s.X e._
RevSZ {
  s.Z, s.F e.X
    = <RevSZ s.Z, e.X> s.F;
  s.Z, ε = s.Z;
}

fmt {
  e.1, s.Z = s.Z e.1;
}

unfmt {
  s.Z e.1 = e.1;
}
```

Форматы могут быть параметризованными




```
$ENTRY Go {
  s.Z, e.X
    = <RevSZ s.Z, e.X>;
}

* <RevSZ #s.Z, e._> :: #s.X e._
RevSZ {
  s.Z, s.F e.X
    = <RevSZ s.Z, e.X> s.F;
  s.Z, ε = s.Z;
}

fmt {
  e.1, s.Z = s.Z e.1;
}

unfmt {
  s.Z e.1 = e.1;
}
```

Форматы могут быть параметризованными

- Остаточная программа:

```
$ENTRY Go {
  s.Z, e.X = s.Z <F1 e.X>;
}

F1 {
  s.2 e.3 = <F1 e.3> s.2;
  ε       = ε;
}
```

- Параметр `s.Z`, не участвовавший в вычислениях, стал частью формата и вынесен из компоненты факторизации.
- Местность среды рекурсивной функции уменьшилась.

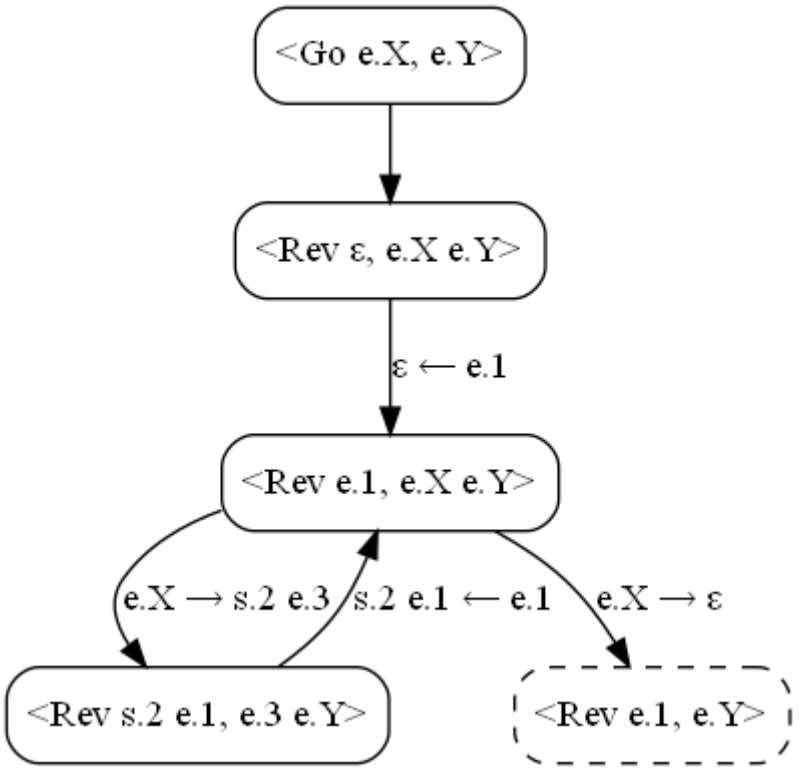
Форматы могут быть активными

- Рассмотрим программу

```
$ENTRY Go {
  e.X, e.Y = <Rev ε, e.X e.Y>;
}

Rev {
  e.A, s.F e.X = <Rev s.F e.A, e.X>;
  e.A, ε       = e.A;
}
```

- Программа обращает конкатенацию двух строк, использует переменную аккумулятор.
- Начнём строить граф суперкомпиляции.

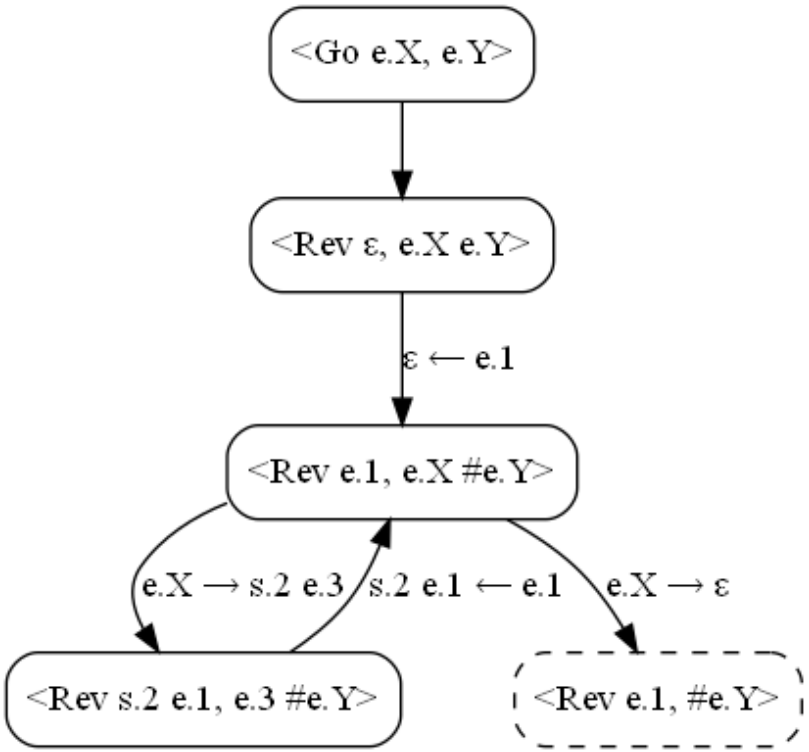
Форматы могут быть активными




```
$ENTRY Go {
  e.X, e.Y
    = <Rev ε, e.X e.Y>;
}

Rev {
  e.A, s.F e.X
    = <Rev s.F e.A, e.X>;
  e.A, ε = e.A;
}
```

- Заметим, что в цикле не участвует параметр `e.Y`.
- Попробуем его заморозить, т.е. будем считать его константой.

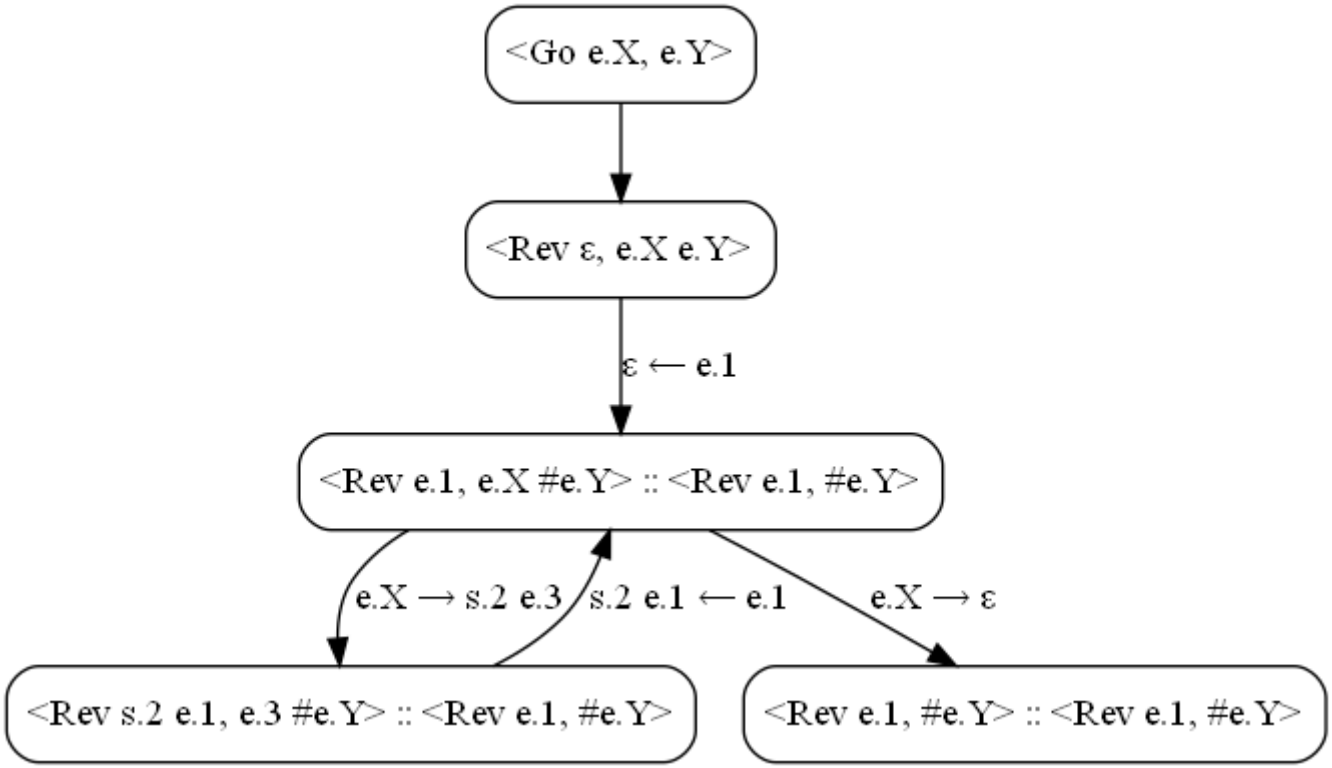
Форматы могут быть активными




```
$ENTRY Go {
   e.X, e.Y
     = <Rev ε, e.X e.Y>;
}

Rev {
   e.A, s.F e.X
     = <Rev s.F e.A, e.X>;
   e.A, ε = e.A;
}
```

- Получили конфигурацию <Rev e.1, #e.Y>, которую невозможно дальше прогнать. Что будем делать?
- Разморозить #e.Y? Не интересно.
- Будем считать её выходным форматом!

Форматы могут быть активными



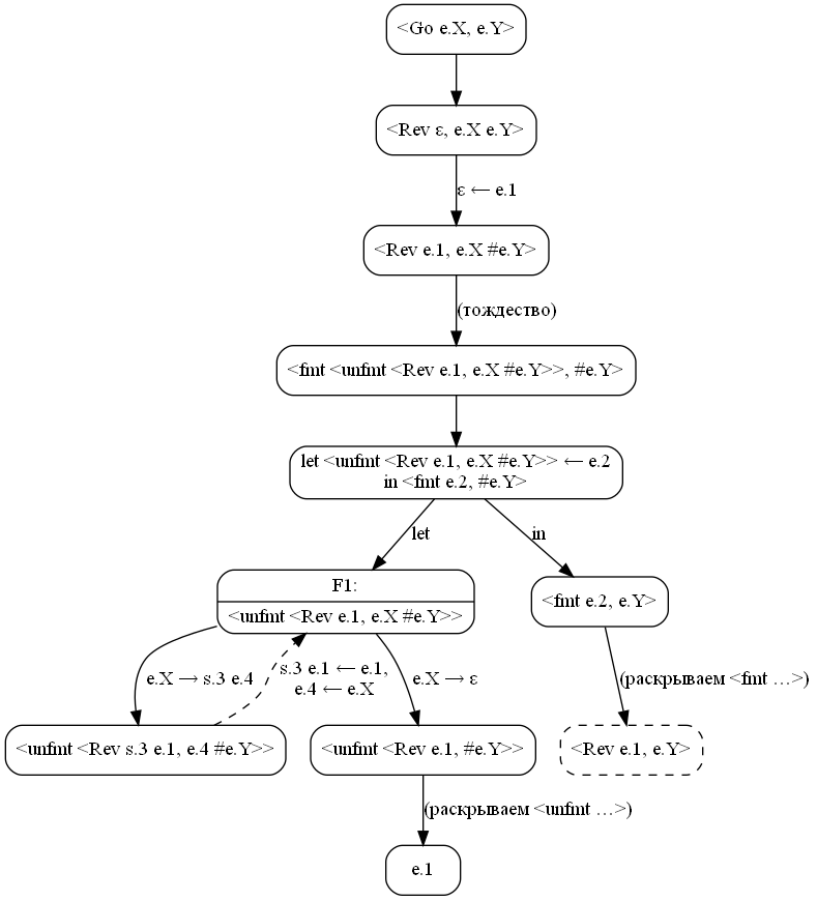
Форматы могут быть активными

- Построим функции форматирования и расформатирования:

```
fmt {
  e.1, e.Y = <Rev e.1, e.Y>;
}

unfmt {
  <Rev e.1, #e.Y> = e.1;
}
```

- Функция unfmt выглядит дико, т.к. в левой части стоит вызов функции. Но мы считаем этот вызов функции особого рода конструктором (примерно как во FLAC’е).

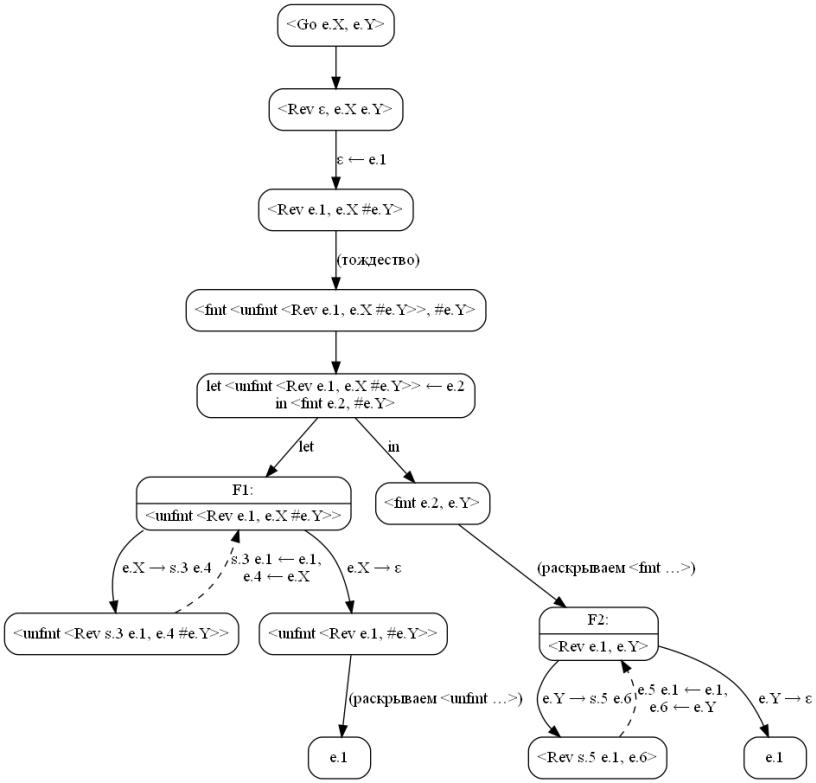
Форматы могут быть активными




```
$ENTRY Go {
  e.X, e.Y
    = <Rev ε, e.X e.Y>;
}

Rev {
  e.A, s.F e.X
    = <Rev s.F e.A, e.X>;
  e.A, ε = e.A;
}

fmt {
  e.1, e.Y = <Rev e.1, e.Y>;
}

unfmt {
  <Rev e.1, #e.Y> = e.1;
}
```

- Конфигурация `<Rev e.1, e.Y>` воспроизведёт исходную функцию `Rev`.

Форматы могут быть активными




```
$ENTRY Go {
  e.X, e.Y
    = <Rev ε, e.X e.Y>;
}

Rev {
  e.A, s.F e.X
    = <Rev s.F e.A, e.X>;
  e.A, ε = e.A;
}

fmt {
  e.1, e.Y = <Rev e.1, e.Y>;
}

unfmt {
  <Rev e.1, #e.Y> = e.1;
}
```

- Конфигурация `<Rev e.1, e.Y>` воспроизведёт исходную функцию `Rev`.

Форматы могут быть активными

- Остаточная программа примет вид

```
$ENTRY Go {
  e.X, e.Y = <F2 <F1 ε, e.X>, e.Y>;
}

F1 {
  e.1, s.3 e.4 = <F1 s.3 e.1, e.4>;
  e.1, ε = e.1;
}

F2 {
  e.1, s.5 e.6 = <F2 s.5 e.1, e.6>;
  e.1, ε = e.1;
}
```

- Исходный вызов `<Rev ε, e.X e.Y>` оказался разложен на композицию вызовов `<F2 <F1 ε, e.X>, e.Y>`.
- Заметим, что функции `F1` и `F2` идентичны и совпадают с `Rev`.
- Таким образом, доказали тождество `<Rev ε, e.X e.Y>` ≡ `<Rev <Rev ε, e.X>, e.Y>`.

Выводы

- Предложен способ онлайн-вычисления форматов конфигураций в процессе суперкомпиляции.
- Форматы могут быть параметризованными и могут быть активными.
- Использование активных параметризованных форматов позволяет выполнять декомпозицию функций в процессе суперкомпиляции.

Приложение

Специализация интерпретатора стекового языка

- Можно показать, что использование активных выходных форматов позволяет выполнять довольно необычные преобразования.
- Например, можно полностью проспециализировать «плоский» интерпретатор.
- Результирующая функция будет рекурсивной, в то время как исходный интерпретатор был «плоским».
- Простые методы суперкомпиляции для «плоских» исходных программ строят «плоские» остаточные программы.

Специализация интерпретатора стекового языка

```
Int {
  e.stack t.X, dup  e.code, e.defs
    = <Int e.stack t.X t.X, e.code, e.defs>;

  e.stack t.X, drop e.code, e.defs
    = <Int e.stack, e.code, e.defs>;

  e.stack, (e.N) e.code, e.defs
    = <Int e.stack (e.N), e.code, e.defs>;

  e.stack (e.X) (e.Y), '*' e.code, e.defs
    = <Int e.stack (<* e.X, e.Y>), e.code, e.defs>;

  e.stack (e.X) (e.Y), '-', e.code, e.defs
    = <Int e.stack (<- e.X, e.Y>), e.code, e.defs>;

  e.stack (0), if (e.T) else (e.F) e.code, e.defs
    = <Int e.stack, e.F e.code, e.defs>;

  e.stack (e.X), if (e.T) else (e.F) e.code, e.defs
    = <Int e.stack, e.T e.code, e.defs>;

  e.stack, s.Func e.code e.defs
    = <Int e.stack, <Lookup s.Func, e.defs> e.code, e.defs>;

  e.stack (e.Res), ε, e.defs = e.Res;
}

Lookup {
  s.Func, (s.Func e.Body) e.defs = e.Body;
  s.Func, t.other e.defs = <Lookup s.Func, e.defs>;
}
```

Специализация интерпретатора стекового языка

```
$ENTRY Fact {
  s.N
    = <Int
        (s.N), fact,
        (fact
          dup if (
            dup dec fact mul
          ) else (
            drop (1)
          )
        )
        (dec (1) '-')
      >;
}
```

Специализация интерпретатора стекового языка

- Использование активных параметризованных форматов позволит построить следующую остаточную программу:

```
$ENTRY Fact {
  s.N = <F1 s.N>;
}

F1 {
  0 = 1;
  e.1 = <* e.1, <F1 <- e.1, 1>>>;
}
```

- Терминология интерпретатора в остаточной программе отсутствует.
- Исходная программа была «плоской». Остаточная стала рекурсивной.