«Суперкомпилятор простого функционального языка, анализирующий выходные форматы конфигураций»

Никита Горбатов
Науч. рук.: Коновалов. А. В.

01.07.2024

Цели и задачи работы

Суперкомпиляция

Набор методов анализа и преобразования программ, основанный на построении вычислительного графа с последующей трансформацией в остаточную программу.

Суперкомпиляция позволяет:

- анализировать свойства программ;
- осуществлять специализацию программ;
- находить композицию программ или функций;
- оптимизировать программы;
- решать задачу верификации;

Целью работы является разработка алгоритма нахождения выходных форматов функций в процессе построения вычислительного графа; разработка алгоритма генерации остаточной программы; проектирование и реализация экспериментального суперкомпилятора, оснащённого новым методом анализа.

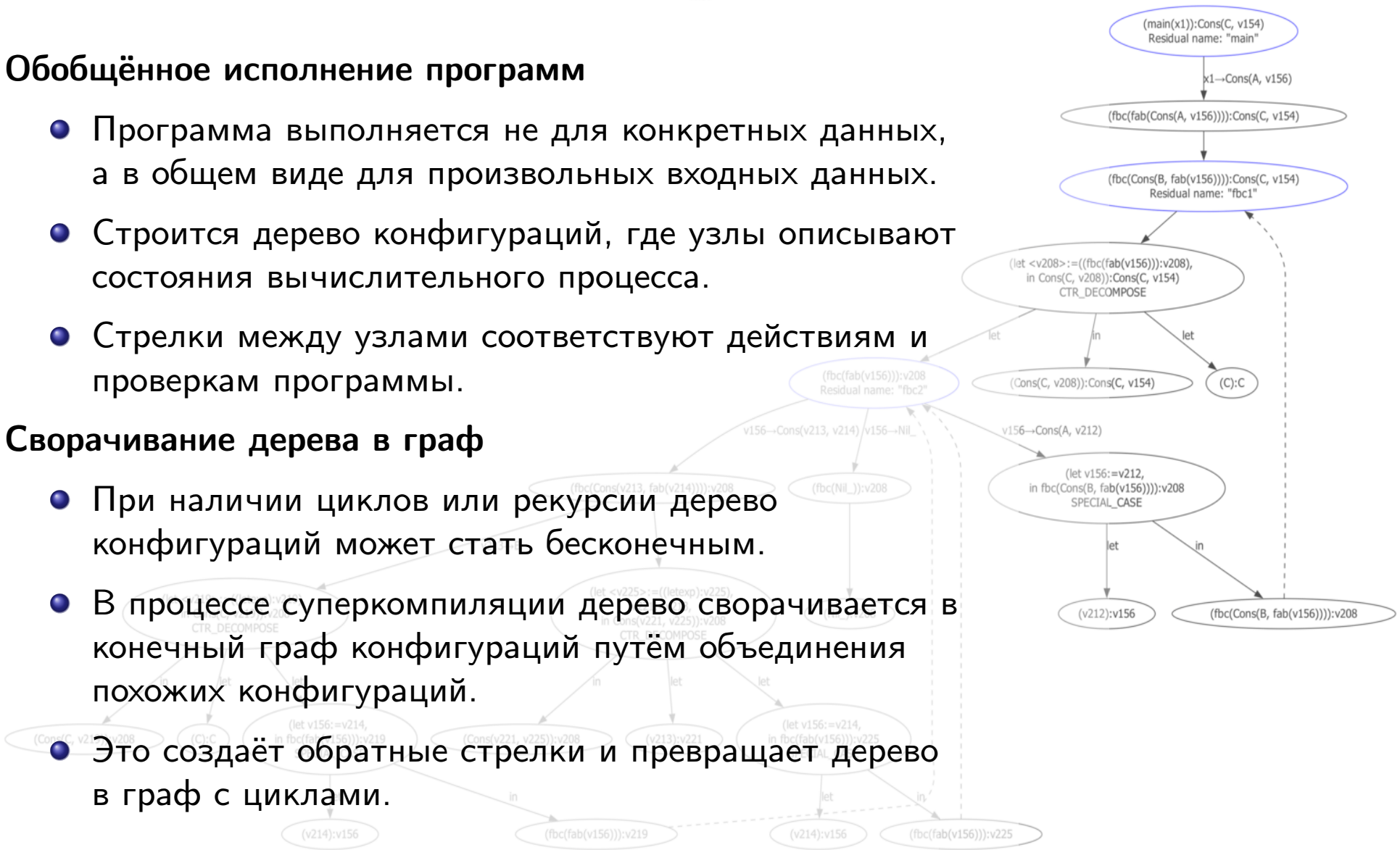
Цель и составные части суперкомпиляции (1)

Обобщённое исполнение программ

- Программа выполняется не для конкретных данных, а в общем виде для произвольных входных данных.
- Строится дерево конфигураций, где узлы описывают состояния вычислительного процесса.
- Стрелки между узлами соответствуют действиям и проверкам программы.

Сворачивание дерева в граф

- При наличии циклов или рекурсии дерево конфигураций может стать бесконечным.
- В процессе суперкомпиляции дерево сворачивается в конечный граф конфигураций путём объединения похожих конфигураций.
- Это создаёт обратные стрелки и превращает дерево в граф с циклами.

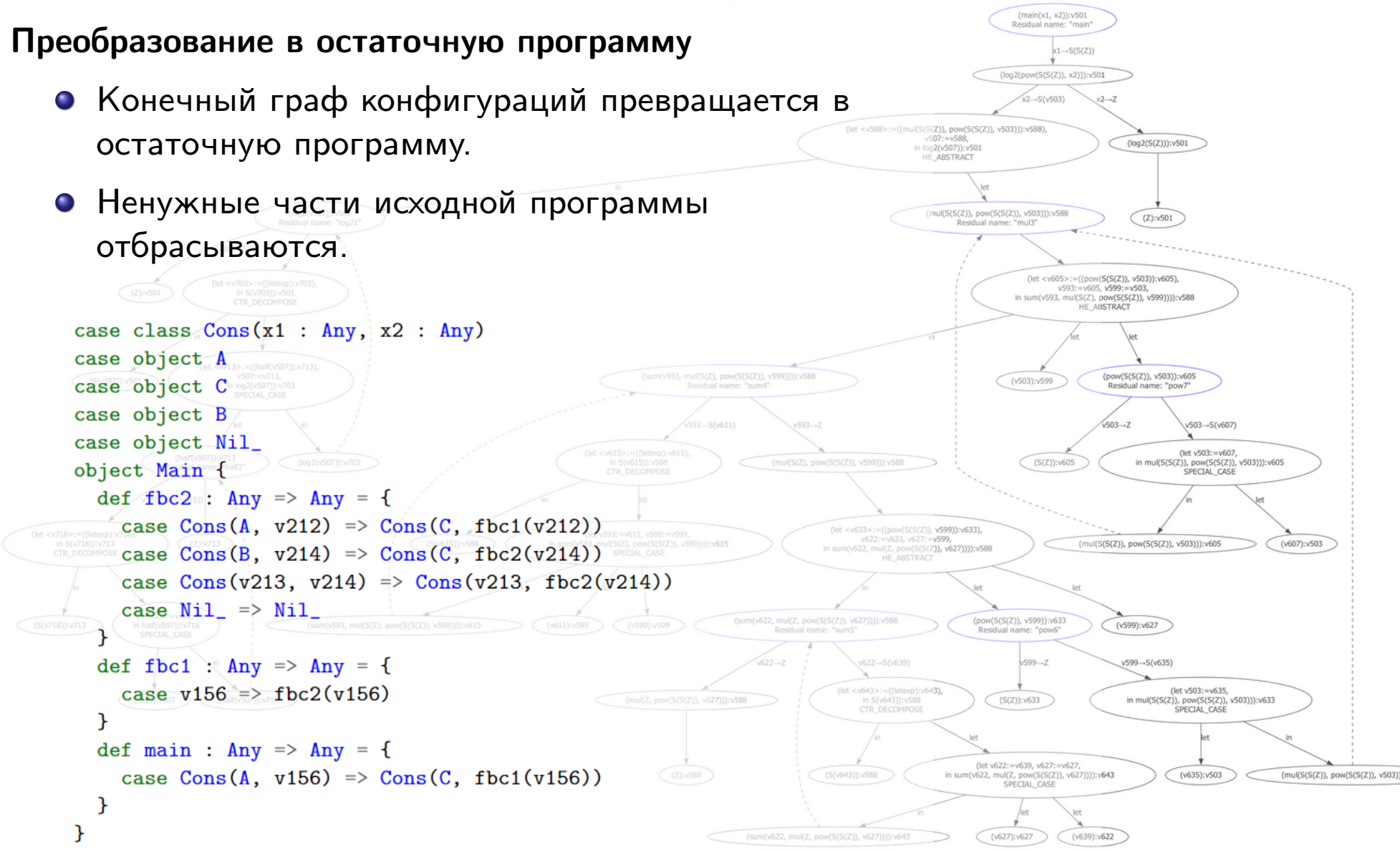


Цель и составные части суперкомпиляции (2)

Преобразование в остаточную программу

- Конечный граф конфигураций превращается в остаточную программу.
- Ненужные части исходной программы отбрасываются.

```
case class Cons(x1 : Any, x2 : Any)
case object A
case object C
case object B
case object Nil_
object Main {
  def fbc2 : Any => Any = {
    case Cons(A, v212) => Cons(C, fbc1(v212))
    case Cons(B, v214) => Cons(C, fbc2(v214))
    case Cons(v213, v214) => Cons(v213, fbc2(v214))
    case Nil_ => Nil_
  }
  def fbc1 : Any => Any = {
    case v156 => fbc2(v156)
  }
  def main : Any => Any = {
    case Cons(A, v156) => Cons(C, fbc1(v156))
  }
}
```



Модельный язык

- Программы на простом функциональном языке.
- Входные и выходные данные — композиции конструкторов (именованных кортежей).
- Функции определяются правилами переписывания вида ОБРАЗЕЦ => РЕЗУЛЬТАТ_ПЕРЕПИСЫВАНИЯ

```
// Объявление двуместного конструктора
case class Pair(x: Any, y: Any)

// Объявление нульарных конструкторов - констант
case object Apple
case object Half

// Объявление функции
def cut: Any => Any = {
  case Apple => Pair(Half, Half)   // Первое правило переписывания
  case x => x                      // Второе правило переписывания
}
```

Пример программы на модельном языке

Один из способов определить неотрицательные целые числа:

- `Z` — нульместный конструктор (константа), означающий 0;
- `S(x)` — одноместный конструктор, означающий инкремент;
- `S(Z)` — единица, `S(S(Z))` — двойка и т.д.

```
case class S(x: Any)
case object Z

def sum: (Any, Any) => Any = {
  case (S(x), y) => S(sum(x, y))
  case (Z, y)    => y
}
```

$$\texttt{sum(S(S(Z)), S(S(Z)))} \xrightarrow{\text{I}} \texttt{S(sum(S(Z), S(S(Z))))} \xrightarrow{\text{I}} \texttt{S(S(sum(Z, S(S(Z)))))} \xrightarrow{\text{II}} \texttt{S(S(S(S(Z))))}$$

$$\texttt{sum(S(S(Z)), b)} \xrightarrow{\text{I}} \texttt{S(sum(S(Z), b))} \xrightarrow{\text{I}} \texttt{S(S(sum(Z, b)))} \xrightarrow{\text{II}} \texttt{S(S(b))}$$

Специализация sum(a, b) по a = S(S(Z))

$$\texttt{sum(S(S(Z)), b)} \xrightarrow{\text{I}} \texttt{S(sum(S(Z), b))} \xrightarrow{\text{I}} \texttt{S(S(sum(Z, b)))} \xrightarrow{\text{II}} \texttt{S(S(b))}$$

Исходная программа:

```
case class S(x: Any)
case object Z

def sum: (Any, Any) => Any = {
  case (S(x), y) => S(sum(x, y))
  case (Z, y)    => y
}
```

Остаточная программа:

```
case class S(x: Any)
case object Z

def main: (Any, Any) => Any = {
  case(S(S(Z)), y) => sum1(y)
}

def sum1: Any => Any = {
  case y => S(S(y))
}
```

Специализация программ

Пусть P — исходная программа, а Γ — ограничения на условия эксплуатации P, тогда специализатор — программа, принимающая на вход пару (P, Γ) и порождающая на выходе остаточную программу P', удовлетворяющую следующим требованиям.

- P' эквивалентна P, если выполнены условия Γ.
- P' получается путем устранения из P тех частей и действий, которые становятся ненужными в результате наложения условий Γ.

Надежды и ожидания, связанные со специализацией P, состоят в следующем.

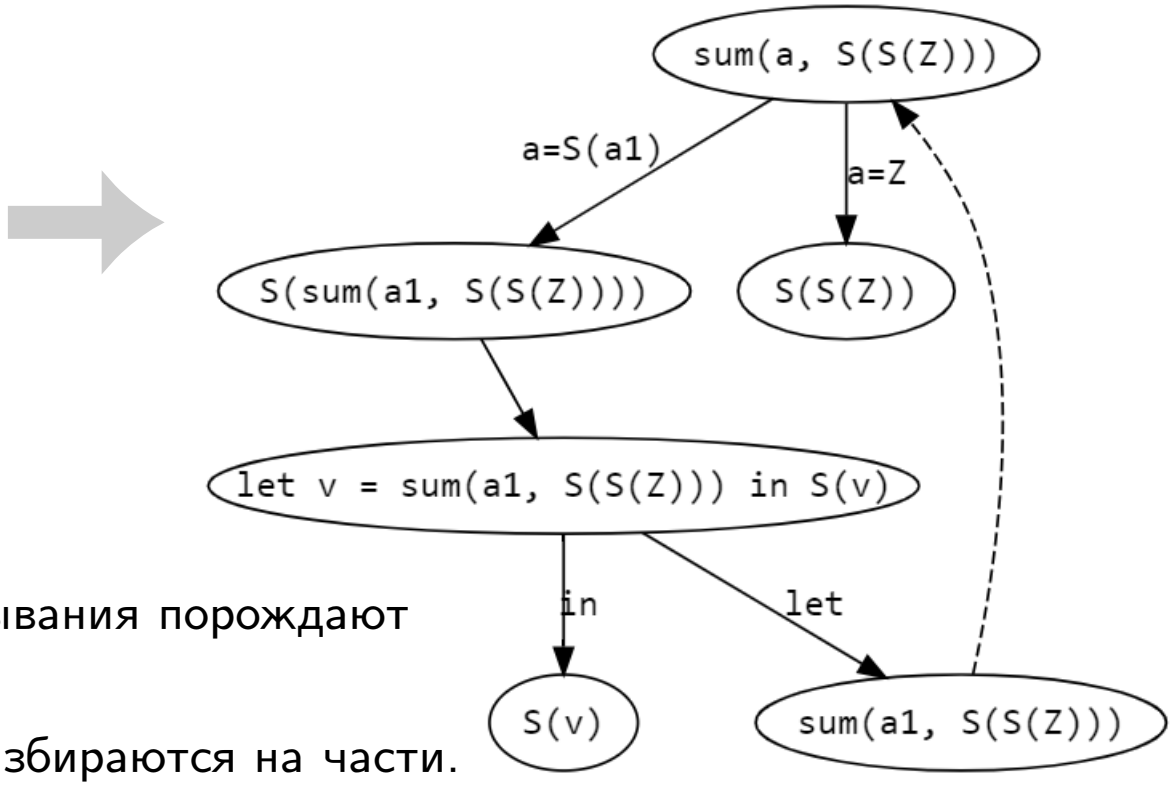
- P' будет эффективнее, чем P.
- P' будет меньше, чем P.
- P' будет проще, чем P

Специализация sum(a, b) по b = S(S(Z)): граф конф-ий

Исходная программа:

```
case class S(x: Any)
case object Z

def sum: (Any, Any) => Any = {
  case (S(x), y) => S(sum(x, y))
  case (Z, y)    => y
}
```



- Образцы правил переписывания порождают ветви прогонки.
- Внешние конструкторы разбираются на части.
- Если дочерняя конфигурация с точностью до имён параметров совпадает с той, что лежит выше по графу конфигураций, проводится обратная стрелка.

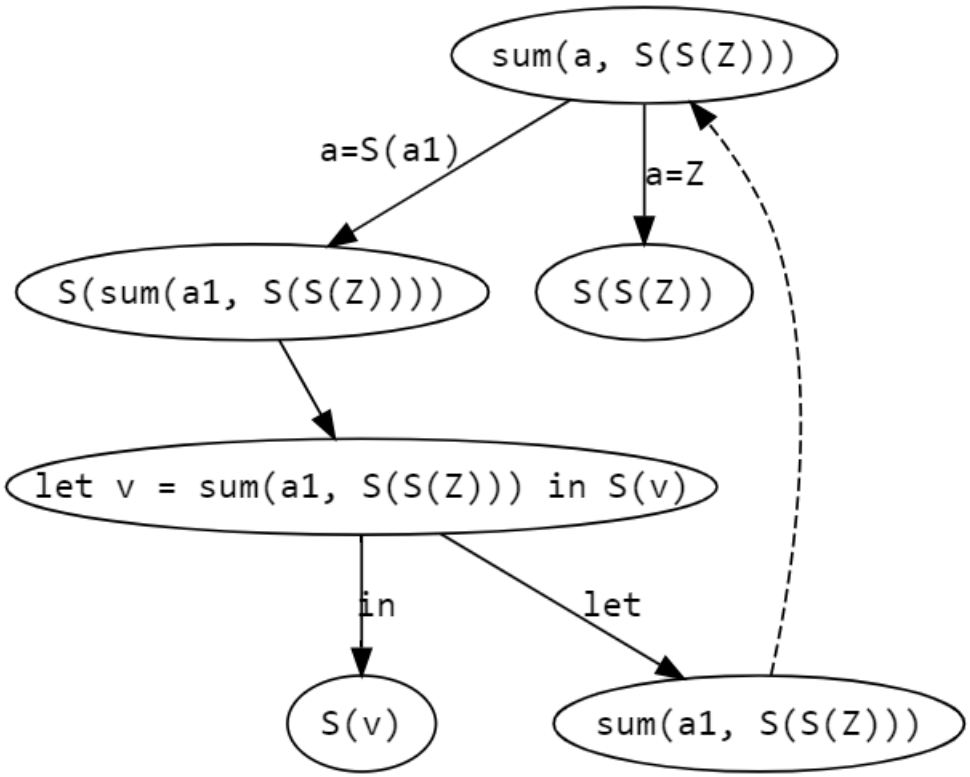
Специализация sum(a, b) по b = S(S(Z)): остат. прогр.

Остаточная программа:

```
case class S(x: Any)
case object Z

def main: (Any, Any) => Any = {
  case (x, S(S(Z))) => sum1(x)
}

def sum1: Any => Any = {
  case S(x) => S(sum1(x))
  case Z => S(S(Z))
}
```



- Базисные конфигурации получают идентификаторы и становятся остаточными функциями.
- Внешние конструкторы разбираем на части.
- Обратные стрелки соответствуют вызовам остаточных функций.

Выходные форматы

- Пусть $\mathbb{D}$ — область применимости программы (конфигурации), т.е. множество входных данных, на которых программа завершается (конфигурация вычисляется) без ошибок и возвращает результат.
- Пусть $\mathbb{E}$ — совокупность возвращаемых программой (конфигурацией) результатов на $\mathbb{D}$.

Выходной формат

Аппроксимация множества $\mathbb{E}$.

Пример: Выходной формат конфигураций `sum(a, S(S(Z)))` и `sum(S(S(Z)), a)`, где a — неизвестный аргумент равен `S(S(x))`.

Выходные форматы в суперкомпиляторе SCP4

Исходная программа:

```
$ENTRY Go { e.1 = <tailRec e.1>; }

tailRec {
    (e.1) = (<tailRec e.1>);
     = ((()));
}
```


Остаточная программа:

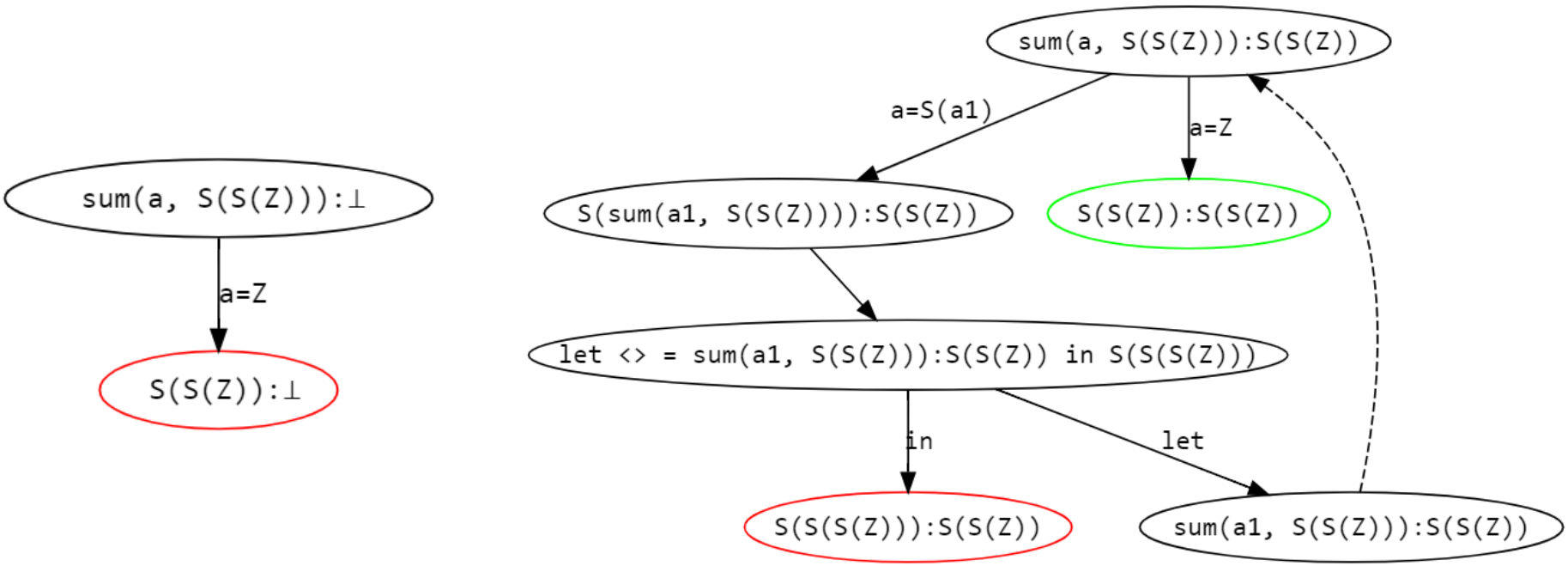
```
$ENTRY Go { e.1 = (<F6 e.1>); }

F6 {
   (e.101) = (<F6 e.101>) ;
    = (()) ;
}
```

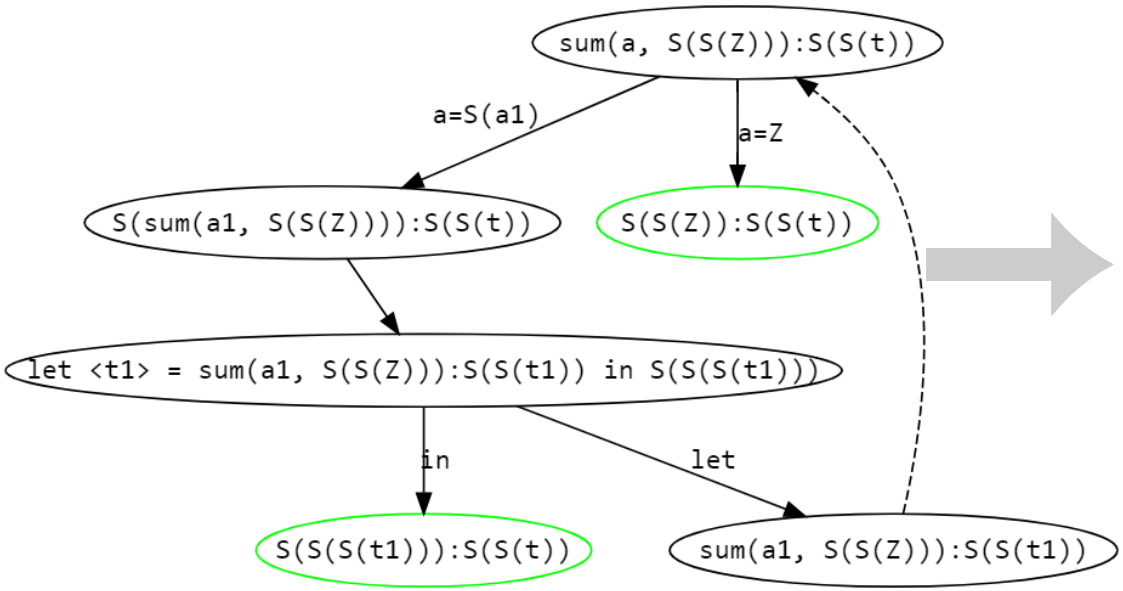
Алгоритм построения выходных форматов

1. Перед построением компоненты вычислительного графа для корневой конфигурации β предполагаем нуль-гипотезу: $\mathbb{E}$ — пустое множество.
2. Вычисляем конфигурацию, т.е. строим вычислительный подграф конфигурации β.
3. Конфигурации подграфа делят с β одну гипотезу.
4. Если в процессе вычисления получили результат в листовой вершине, который не удовлетворяет формат, то:
   - наиболее тесно обобщаем выходной формат;
   - возвращаемся к пункту 2 и перестраиваем всё поддерево с новой форматной гипотезой.
5. Если в процессе вычисления получили конфигурацию γ, из которой выходит обратная стрелка в α, тогда необходимо подставить выходной формат α вместо выражения γ и выполнить проверку из пункта 4.

Построение выходных форматов (1)



Генерация остаточной программы




```
def sum1 : Any => Any = {
  case S(a1) => {
    t1 = sum1(a1)
    S(t1)
  }
  case Z => Z
}
def main : Any => Any = {
  case (a, S(S(Z))) => S(S(sum1(a)))
}
```

- Считаем, что конфигурация теперь не вычисляет свой результат напрямую, а находит подстановку в формат, приводящую к своему результату.
- Необходимая подстановка находится при суперкомпиляции подграфа конфигурации.

Пример преобразования progression.scala

Исходная программа:

```
case class S(x: Any)
case class Cons(x: Any, y: Any)
case object Z
case object Nil_

object Main {
  def main: (Any, Any, Any) => Any = {
    case (st, S(inc), S(S(Z))) => {
      f(Cons(st, Nil_), S(inc), S(S(Z)))
    }
  }
  def f: (Any, Any, Any) => Any = {
    case (xs, inc, Z) => xs
    case (Cons(a, xs), inc, S(x)) => {
      f(Cons(sum(a, inc), Cons(a, xs)), inc, x)
    }
  }
  def sum: (Any, Any) => Any = {
    case (x, S(n1)) => S(sum(x, n1))
    case (x, Z) => x
  }
}
```

Остаточная программа:

```
case class S(x1 : Any)
case object Z
case class Cons(x1 : Any, x2 : Any)
case object Nil_
object Main {
  def sum2 : (Any, Any) => Any = {
    case (v346, S(v392)) => S(sum2(v346, v392))
    case (v346, Z) => v346
  }
  def sum1 : (Any, Any, Any) => Any = {
    case (v346, v348, S(v375)) => S(sum1(v346, v348, v375))
    case (v346, v348, Z) => sum2(v346, v348)
  }
  def sum3 : (Any, Any) => Any = {
    case (x1, S(v427)) => S(sum3(x1, v427))
    case (x1, Z) => x1
  }
  def main : (Any, Any, Any) => Any = {
    case (x1, S(v271), S(S(Z))) => {
       Cons(S(S(sum1(x1, v271, v271))),
            Cons(S(sum3(x1, v271)),
            Cons(x1, Nil_)))
     }
  }
}
```

Пример преобразования fac.scala

Исходная программа:

```
case object A
case object B
case object C
case object Nil_
case class Cons(head: Any, tail: Any)

object Main {
  def fab : Any => Any = {
    case Cons(A, xs) => Cons(B, fab(xs))
    case Cons(x, xs) => Cons(x, fab(xs))
    case Nil_ => Nil_
  }

  def fbc : Any => Any = {
    case Cons(B, xs) => Cons(C, fbc(xs))
    case Cons(x, xs) => Cons(x, fbc(xs))
    case Nil_ => Nil_
  }

  def main : Any => Any = {
    case Cons(A, xs) => fbc(fab(Cons(A,xs)))
  }
}
```

Остаточная программа:

```
case class Cons(x1 : Any, x2 : Any)
case object A
case object C
case object B
case object Nil_
object Main {
  def fbc2 : Any => Any = {
    case Cons(A, v212) => Cons(C, fbc1(v212))
    case Cons(B, v214) => Cons(C, fbc2(v214))
    case Cons(v213, v214) => Cons(v213, fbc2(v214))
    case Nil_ => Nil_
  }
  def fbc1 : Any => Any = {
    case v156 => fbc2(v156)
  }
  def main : Any => Any = {
    case Cons(A, v156) => Cons(C, fbc1(v156))
  }
}
```

Тестирование и результаты

Результаты

- Разработан алгоритм нахождения выходных форматов функций в процессе построения вычислительного графа.
- Разработан алгоритм генерации остаточной программы, учитывающий выходные форматы.
- Спроектирован и реализован экспериментальный суперкомпилятор, оснащённый новым инструментом анализа.

Тестирование

- Проверка корректности преобразований. Осуществлялась проверкой на эквивалентность программ на конечном подмножестве из области определения.
- Тестирование эффективности полученных программ. Проводилось сравнением среднего времени выполнения на тестовых данных.

Спасибо за внимание!

Github: