Суперкомпилятор SC Mini

Приложение к статье «Суперкомпиляция: идеи и методы»

Илья Ключников

Аннотация

В приложении подробно шаг за шагом разбирается устройство суперкомпилятора SC Mini.

The internals of the supercompiler SC Mini are explained step by step in this appendix.

Оглавление

Введение

Код суперкомпилятора SC Mini можно условно разбить на следующие части:

1) Описание выбранных абстракций через типы данных: данные для абстрактного синтаксиса языка SLL и операций с SLL-выражениями, данные для моделирования выполнения SLL-заданий, данные для работы с графом конфигураций.
   - `Data.hs`
2) Вспомогательные функции для работы с данными: парсер, подстановки, сравнения выражений и т.д.
   - `DataUtil.hs`
3) Основная часть – интерпретатор, прогонка, свертка, преобразователи `transform`, `deforest` и `supercompile`, генератор остаточного задания.
   - `Interpreter.hs`
   - `Driving.hs`
   - `TreeInterpreter.hs`
   - `Folding.hs`
   - `Generator.hs`
   - `Prototype.hs`
   - `Deforester.hs`
   - `Supercompiler.hs`
4) Набор демонстрационных примеров.
   - `Demonstration.hs`

Листинги всех частей SC Mini приводятся полностью (пропущены только парсеры/претти-принтеры). Практически каждое вводимое понятие или метод иллюстрируется работающим примером. Все примеры собраны в модуле `Demonstration.hs`. Читатель может самостоятельно запустить пример и убедиться в его работоспособности – везде указывается функция (`demo01`, `demo02`, ...), вызвав которую из интерпретатора ghci, читатель получит тот же результат, что здесь.

Для удобства ниже повторяется определение операционной семантики SLL.

Рис. 1.1 SLL: декомпозиция выражений

con	$::=$	$\langle\rangle \mid g(con, \ldots)$
red	$::=$	$f(e_1, \ldots, e_n) \mid g(C(e_1, \ldots, e_n), \ldots) \mid g(v, \ldots)$

Рис. 1.2 Правила, описывающие шаг редукции

$\mathcal{I}[\![e]\!]$	$\Rightarrow$	e	если e – значение
$\mathcal{I}[\![C(e_1, \ldots, e_n)]\!]$	$\Rightarrow$	$C(\mathcal{I}[\![e_1]\!], \ldots, \mathcal{I}[\![e_n]\!])$	
$\mathcal{I}[\![con\langle f(e_1, \ldots, e_n)\rangle]\!]$	$\Rightarrow$	$\mathcal{I}[\![con\langle e\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]\!]$	при $f(x_1, \ldots, x_n) \stackrel{\text{p}}{=} e$
$\mathcal{I}[\![con\langle g(C(e_1, \ldots, e_m), e_{m+1}, \ldots, e_n)\rangle]\!]$	$\Rightarrow$	$\mathcal{I}[\![con\langle e\{v_1 := e_1, \ldots, v_n := e_n\}\rangle]\!]$	при $g(C(v_1, \ldots, v_m), v_{m+1}, \ldots, v_n) = e$

1.1. Data.hs

Определим вначале абстрактный синтаксис. Для удобства сразу включим в него `let`-выражения.

```
type Name = String
data Expr = Var Name | Ctr Name [Expr] | FCall Name [Expr] | GCall Name [Expr]
          | Let (Name, Expr) Expr deriving (Eq)
data Pat = Pat Name [Name] deriving (Eq)
data GDef = GDef Name Pat [Name] Expr deriving (Eq)
data FDef = FDef Name [Name] Expr deriving (Eq)
data Program = Program [FDef] [GDef] deriving (Eq)
```

Синонимы типов для операций над выражениями: переименовка, подстановка, список “свежих” имен.

```
type Renaming = [(Name, Name)]
type Subst = [(Name, Expr)]
type NameSupply = [Name]
```

Технически мы будем далее практически везде работать с SLL-выражениями. Однако, чтобы различать, какой смысл мы в него вкладываем, – просто выражение, конфигурация, SLL-значение – введем еще несколько синонимов типов:

```
type Conf = Expr
type Value = Expr
type Task = (Conf, Program)
type Env = [(Name, Value)]
```

Самые интересные типы данных:

```
data Contract = Contract Name Pat
data Step a = Transient a | Stop | Decompose [a] | Variants [(Contract, a)] | Fold a Renaming
data Graph a = Node a (Step (Graph a))
type Tree a = Graph a
type Node a = Tree a

type Machine a = NameSupply → a → Step a
```

Далее мы будем рассматривать программу на языке именно как некоторую машину. `Machine a` оперирует некоторыми обобщенными состояниями типа `a`. Мы предполагаем, что состояния описываются с помощью некоторых именованных конструкций (то есть в них встречаются имена – идентификаторы) и машина порождает новые именованные конструкции. Если машине дать некоторое (бесконечное) множество имен и некоторое текущее состояние, то она вычислит шаг перехода в следующее обобщенное состояние. Тип данных `Step a` описывает рассматриваемые нами шаги, – транзитные, остановка, декомпозиция и т.д. Самый интересный шаг – `Variants [(c1, a1), (c2, a2),...]` – описание возможных вариантов. Мы вкладываем в него следующий смысл: если выполняется условие `c1`, то следующим состоянием будет `a1`, при `c2` – `a2` и т.д. Мы будем рассматривать очень простые условия (`Contract`) – что некоторое имя соответствует образцу. Соответственно, `Graph a` – граф переходов между состояниями.

Дальше мы в качестве обобщенных состояний будем рассматривать конфигурации, но я хочу подчеркнуть, что в принципе обобщенным состоянием может быть все, что угодно.

1.2. DataUtil.hs

В модуле `DataUtil` определяются скучные вспомогательные функции. Полный текст приводится исключительно по просьбам трудящихся.

Работа с абстрактным синтаксисом:

```
1  isValue :: Expr → Bool
2  isValue (Ctr _ args) = and $ map isValue args
3  isValue _ = False
4
5  isCall :: Expr → Bool
6  isCall (FCall _ _) = True
7  isCall (GCall _ _) = True
8  isCall _ = False
9
10 isVar :: Expr → Bool
11 isVar (Var _) = True
12 isVar _ = False
13
14 fDef :: Program → Name → FDef
15 fDef (Program fs _) fname = head [f | f@(FDef x _ _) ← fs, x == fname]
16
17 gDefs :: Program → Name → [GDef]
18 gDefs (Program _ gs) gname = [g | g@(GDef x _ _ _) ← gs, x == gname]
19
20 gDef :: Program → Name → Name → GDef
21 gDef p gname cname = head [g | g@(GDef _ (Pat c _) _ _) ← gDefs p gname, c == cname]
```

Применение подстановки:

```
22 (//) :: Expr → Subst → Expr
23 (Var x) // sub = maybe (Var x) id (lookup x sub)
24 (Ctr name args) // sub = Ctr name (map (// sub) args)
25 (FCall name args) // sub = FCall name (map (// sub) args)
26 (GCall name args) // sub = GCall name (map (// sub) args)
27 (Let (x, e1) e2) // sub  = Let (x, (e1 // sub)) (e2 // sub)
```

Работа с именами. `nameSupply` - бесконечное множество имен (для переменных). `unused` – "вычитает" из множества имен имена, использованные в условии. `vnames'` – имена переменных в выражении, `vnames` – то же самое без повторений. `isRepeated` – встречается ли данное имя переменной более одного раза в выражении.

```
28 nameSupply :: NameSupply
29 nameSupply = ["v" ++ (show i) | i ← [1 ..] ]
30
31 unused :: Contract → NameSupply → NameSupply
32 unused (Contract _ (Pat _ vs)) = (\\ vs)
33
34 vnames :: Expr → [Name]
35 vnames = nub · vnames'
36
37 vnames' :: Expr → [Name]
38 vnames' (Var v) = [v]
39 vnames' (Ctr _ args)   = concat $ map vnames' args
40 vnames' (FCall _ args) = concat $ map vnames' args
41 vnames' (GCall _ args) = concat $ map vnames' args
42 vnames' (Let (_, e1) e2) = vnames' e1 ++ vnames' e2
43
44 isRepeated :: Name → Expr → Bool
45 isRepeated vn e = (length $ filter (== vn) (vnames' e)) > 1
```

Размер выражения (используется в свистке):

```
46 size :: Expr → Integer
47 size (Var _)          = 1
48 size (Ctr _ args)     = 1 + sum (map size args)
49 size (FCall _ args)   = 1 + sum (map size args)
50 size (GCall _ args)   = 1 + sum (map size args)
51 size (Let (_, e1) e2) = 1 + (size e1) + (size e2)
```

`renaming e1 e2` – нахождение переименовки (если есть) выражения `e1` в выражение `e2` (написано не самым лучшим образом, должно быть элегантнее):

```
52 renaming :: Expr → Expr → Maybe Renaming
53 renaming e1 e2 = f $ partition isNothing $ renaming' (e1, e2) where
54     f (x:_, _) = Nothing
55     f (_, ps) = g gs1 gs2
56         where
57             gs1 = groupBy (λ(a, b) (c, d) → a == c) $ sortBy h $ nub $ catMaybes ps
58             gs2 = groupBy (λ(a, b) (c, d) → b == d) $ sortBy h $ nub $ catMaybes ps
59             h (a, b) (c, d) = compare a c
60     g xs ys = if all ((== 1) · length) xs && all ((== 1) · length) ys
61         then Just (concat xs) else Nothing
62
63 renaming' :: (Expr, Expr) → [Maybe (Name, Name)]
64 renaming' ((Var x), (Var y)) =
65     [Just (x, y)]
66 renaming' ((Ctr n1 args1), (Ctr n2 args2)) | n1 == n2 =
67     concat $ map renaming' $ zip args1 args2
68 renaming' ((FCall n1 args1), (FCall n2 args2)) | n1 == n2 =
69     concat $ map renaming' $ zip args1 args2
70 renaming' ((GCall n1 args1), (GCall n2 args2)) | n1 == n2 =
71     concat $ map renaming' $ zip args1 args2
72 renaming' (Let (v, e1) e2, Let (v', e1') e2') =
73     renaming' (e1, e1') ++ renaming' (e2, e2' // [(v, Var v')])
74 renaming' _  = [Nothing]
```

1.3. Interpreter.hs

В модуле `Interpreter` определяется два вычислителя SLL-выражений.

Интерпретатор `int` – является интерпретатором с малым шагом и работает следующим образом: делается шаг редукции до тех пор, пока выражение не станет значением. Шаг редукции (`intStep`) не является рекурсивной функцией.

```
1  int :: Program → Expr → Expr
2  int p e = until isValue (intStep p) e
3
4  intStep :: Program → Expr → Expr
5  intStep p (Ctr name args) =
6      Ctr name (values ++ (intStep p x : xs)) where
7          (values, x : xs) = span isValue args
8
9  intStep p (FCall name args) =
10     body // zip vs args where
11         (FDef _ vs body) = fDef p name
12
13 intStep p (GCall gname (Ctr cname cargs : args)) =
14     body // zip (cvs ++ vs) (cargs ++ args) where
15         (GDef _ (Pat _ cvs) vs body) = gDef p gname cname
16
17 intStep p (GCall gname (e:es)) =
18     (GCall gname (intStep p e : es))
19
20 intStep p (Let (x, e1) e2) =
21     e2 // [(x, e1)]
```

Интерпретатор `eval` – интерпретатор с большим шагом. Определяется в виде одной рекурсивной функции.

```
22 eval :: Program → Expr → Expr
23 eval p (Ctr name args) =
24     Ctr name [eval p arg | arg ← args]
25
26 eval p (FCall name args) =
27     eval p (body // zip vs args) where
28         (FDef _ vs body) = fDef p name
29
30 eval p (GCall gname (Ctr cname cargs : args)) =
31     eval p (body // zip (cvs ++ vs) (cargs ++ args)) where
32         (GDef _ (Pat _ cvs) vs body) = gDef p gname cname
33
34 eval p (GCall gname (arg:args)) =
35     eval p (GCall gname (eval p arg:args))
36
37 eval p (Let (x, e1) e2) =
38     eval p (e2 // [(x, e1)])
```

Для вычисления с малым шагом можно относительно просто подсчитать число совершенных шагов редукции, – достаточно ввести счетчик шагов во внешнем цикле. `intC` возвращает пару `(value, n)`, где `value` – значение, а `n` – число совершенных шагов редукции. В этих шагах мы и будем измерять “ускорение оптимизации”. Также определяются вспомогательные функции.

```
39 intC :: Program → Expr → (Expr, Integer)
40 intC p e = until t f (e, 0) where
41     t (e, n) = isValue e
42     f (e, n) = (intStep p e, n + 1)
43
44 sll_run :: Task → Env → Value
45 sll_run (e, program) env = int program (e // env)
46
47 sll_trace :: Task → Subst → (Value, Integer)
48 sll_trace (e, prog) s = intC prog (e // s)
```

С этого места начинаются примеры. Каждый приведенный пример оформлен в виде отдельной функции в модуле `Demonstration`. Нам необходимо различать код на Хаскеле и код на SLL. Код на SLL выделен цветом (реально в `Demonstration.hs` вместо цветного кода соответствующие выражения парсятся, `even(x)` ⇒ `read "even(x)"`)

Вычисляем выражение и узнаем, сколько редукций было сделано:

```
-- demo01
ghci> intC prog1 even(sqr(S(S(Z()))))
(True(), 15)
```

Вычисляем интерпретатором с малым шагом. Вычисляем интерпретатором с большим шагом, – результат тот же:

```
-- demo02
ghci> int  prog1 even(sqr(S(S(Z()))))
True()

-- demo03
ghci> eval prog1 even(sqr(S(S(Z()))))
True()

-- demo04
ghci> int  prog1 sqr(S(S(Z())))
S(S(S(S(Z()))))

-- demo05
ghci> eval prog1 sqr(S(S(Z())))
S(S(S(S(Z()))))
```

Здесь проявляются некоторые различия между интерпретаторами. Если внутри `int` возникает непредвиденная ситуация (вместо замкнутого выражения пытаемся вычислить конфигурацию), то он вообще ничего не вычисляет (не выдает наружу). Интерпретатор же `eval` выбрасывает наружу то, что смог посчитать:

```
-- demo06
ghci> int  prog1 sqr(S(S(x)))
 Exception: Interpreter.hs: Non-exhaustive patterns in function intStep

-- demo07
ghci> eval  prog1 sqr(S(S(x)))
S(S( Exception: Interpreter.hs: Non-exhaustive patterns in function eval
```

Это еще более заметно, если пробовать вычислять бесконечные значения: интерпретатор `int` зависнет, а `eval` начнет лихорадочно выдавать вычисленное. (Не забывайте про `Ctrl-C`.)

```
ghci> prog3 = inf() = S(inf());
inf() = S(inf());

-- demo08
ghci> int prog3 inf()
^CInterrupted.

-- demo09
ghci> eval prog3 inf()
S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S
(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S^CInterrupted.
```

1.4. Driving.hs

Машина, основанная на прогонке, имитирует шаг выполнения `eval`. `driveMachine p` создает такую машину для программы `p`.

Если конфигурация – переменная или значение (строки 4, 5), то выдается остановка. Если конфигурация – конструктор, то `eval` переходит к вычислению аргументов конструктора. Поскольку `driveMachine p` моделирует "шаг" `eval`, то выдается шаг `Decompose args`, что "показывает" что теперь нужно перейти к аргументам. `Let`-выражение мы договорились обрабатывать по частям, поэтому опять переходим к аргументам: `Decompose [e1, e2]`. Прогонка вызова `f`-функции (безразличной) проста: в тело функции подставляются параметры (этот шаг часто обозначается как unfolding). Вызов `g`-функции с конструктором на месте первого параметра рассматривается схожим образом: выдается правая часть соответствующего (конструктору) определения `g`-функции с подставленными параметрами. Самое интересное – дальше. Строчки 13, 14: если первым аргументом вызова `g`-функции является переменная, то рассматриваются все варианты конструкторов (образцы), которые есть в программе. Здесь нам как раз и понадобится множество свежих переменных `ns`, чтобы сгенерировать соответствующий образец. Разберите последний случай (строки 15-18) самостоятельно.

```
1  driveMachine :: Program → Machine Conf
2  driveMachine p = drive where
3      drive :: Machine Conf
4      drive ns (Var _) = Stop
5      drive ns (Ctr _ []) = Stop
6      drive ns (Ctr _ args) = Decompose args
7      drive ns (Let (x, t1) t2) = Decompose [t1, t2]
8      drive ns (FCall name args) = Transient $ e // (zip vs args) where
9          FDef _ vs e = fDef p name
10     drive ns (GCall gn (Ctr cn cargs : args)) = Transient $ e // sub where
11         (GDef _ (Pat _ cvs) vs e) = gDef p gn cn
12         sub = zip (cvs ++ vs) (cargs ++ args)
13     drive ns (GCall gn args@((Var _):_)) = Variants $ variants gn args where
14         variants gn args = map (scrutinize ns args) (gDefs p gn)
15     drive ns (GCall gn (inner:args)) = inject (drive ns inner) where
16         inject (Transient t) = Transient (GCall gn (t:args))
17         inject (Variants cs) = Variants $ map f cs
18         f (c, t) = (c, GCall gn (t:args))
19
20 scrutinize :: NameSupply → [Expr] → GDef → (Contract, Expr)
21 scrutinize ns (Var v : args) (GDef _ (Pat cn cvs) vs body) =
22     (Contract v (Pat cn fresh), body // sub) where
23         fresh = take (length cvs) ns
24         sub = zip (cvs ++ vs) (map Var fresh ++ args)
```

Функция `buildTree` конструирует для данной стартовой конфигурации (бесконечное) дерево конфигураций. Она это делает с помощью вспомогательной функций `bt`, передавая ей еще начальное множество имен `nameSupply`. Неочевидной может быть строка 33: если машина `m` выдала шаг `Variants [(c1, e1), (c2, e2), ...]`, то мы рекурсивно строим поддеревья для `e1`, `e2`, ... Однако, тонкость заключается в том, что мы не можем использовать при построении поддерева для `e1` имена, которые появились в условии `c1`, иначе возникнут нехорошие конфликты имен, поэтому мы используем `(unused c ns)` как следующее множество неиспользованных имен (из `ns` "вычитаются" имена, использованные в `c`).

```
25 buildTree :: Machine Conf → Conf → Tree Conf
26 buildTree m e = bt m nameSupply e
27
28 bt :: Machine Conf → NameSupply → Conf → Tree Conf
29 bt m ns c = case m ns c of
30     Decompose ds → Node c $ Decompose (map (bt m ns) ds)
31     Transient e → Node c $ Transient (bt m ns e)
32     Stop → Node c Stop
33     Variants cs → Node c $ Variants [(c, bt m (unused c ns) e) | (c, e) ← cs]
```

Шаг прогонки с вариантами:

```
-- demo10
ghci> driveMachine prog1 nameSupply odd(add(x, mult(x, S(x))))
x == Z() ⇒ odd(mult(x, S(x)))
x == S(v1) ⇒ odd(S(add(v1, mult(x, S(x)))))
```

Транзитный шаг:

```
-- demo11
ghci> driveMachine prog1 nameSupply odd(S(add(v1, mult(x, S(x)))))
⇒ even(add(v1, mult(x, S(x))))
```

Бесконечное дерево конфигураций:

```
-- demo12
ghci> buildTree (driveMachine prog1) even(sqr(x))
|__even(sqr(x))
 |
 |__even(mult(x, x))
  ?x == Z()
  |__even(Z())
   |
   |__True()
  ?x == S(v1)
  |__even(add(x, mult(v1, x)))
   ?x == Z()
   |__even(mult(v1, x))
    ?v1 == Z()
    |__even(Z())
     |
     |__True()
    ?v1 == S(v2)
    |__even(add(x, mult(v2, x)))
     ?x == Z()
     |__even(mult(v2, x))
      ?v2 == Z()
      |__even(Z())
       |
       |__True()
      ?v2 == S(v3)
      |__even(add(x, mult(v3, x)))
  ...
```

1.5. TreeInterpreter.hs

Древесный интерпретатор пишется тривиальным образом. Тут только 2 интересных места. Строки 8, 9: среди дочерних ветвей ищется и выбирается вариант, соответствующий текущему окружению (`env`). Строки 19-21: вычисление проходит через цикл в графе, – имена переменных в окружении (ключи) переименовываются.

```
1  intTree :: Tree Conf → Env → Value
2  intTree (Node e Stop) env =
3      e // env
4  intTree (Node (Ctr cname _) (Decompose ts)) env =
5      Ctr cname $ map (λt → intTree t env) ts
6  intTree (Node _ (Transient t)) env =
7      intTree t env
8  intTree (Node e (Variants cs)) env =
9      head $ catMaybes $ map (try env) cs
10 intTree (Node (Let (v, e1) e2) (Decompose [t1, t2])) env =
11     intTree t2 ((v, intTree t1 env) : env)
12 intTree (Node _ (Fold t ren)) env =
13     intTree t $ map (λ(k, v) → (renKey k, v)) env where
14         renKey k = maybe k fst (find ((k ==) · snd)  ren)
15
16 try :: Env → (Contract, Tree Conf) → (Maybe Expr)
17 try env (Contract v (Pat pn vs), t) =
18     if cn == pn then (Just $ intTree t extEnv) else Nothing where
19         c@(Ctr cn cargs) = (Var v) // env
20         extEnv = zip vs cargs ++ env
```

Использование “бесконечных” деревьев конфигураций для вычислений:

```
-- demo13
ghci> intTree (buildTree (driveMachine prog1) even(fSqr(x)) ) [("x", S(S(Z())))]
True()

-- demo14
ghci> intTree (buildTree (driveMachine prog1) even(fSqr(x)) ) [("x", S(S(S(Z()))))]
False()
```

1.6. Folding.hs

Дерево сворачивается в граф стандартной техникой "завязывания узлов"[1]. Мы спускаемся по дереву сверху вниз, накапливая пройденные конфигурации. Если текущая конфигурация совпадает с одной из пройденных с точностью до переименования, то создается соответствующий цикл.

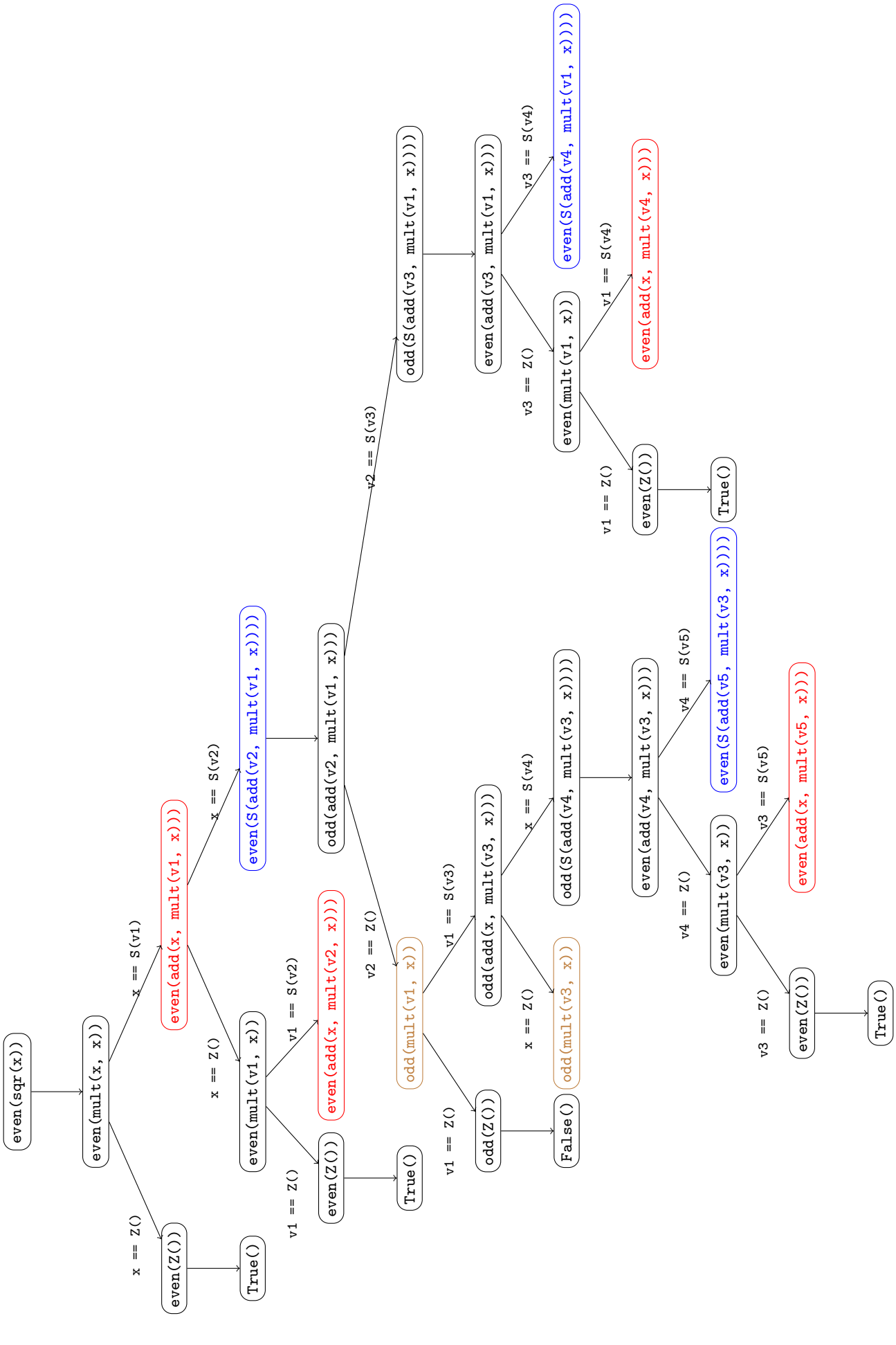
```
foldTree :: Tree Conf → Graph Conf
foldTree t = fixTree (tieKnot []) t

tieKnot :: [Node Conf] → Node Conf → Tree Conf → Graph Conf
tieKnot ns n t@(Node e _) =
    case [(k, r) | k ← n:ns, isCall e, Just r ← [renaming (expr k) e]} of
        [] → fixTree (tieKnot (n:ns)) t
        (k, r):_ → Node e (Fold k r)

fixTree :: (Node t → Tree t → Graph t) → Tree t → Graph t
fixTree f (Node e (Transient c)) = t where
    t = Node e $ Transient $ f t c
fixTree f (Node e (Decompose cs)) = t where
    t = Node e $ Decompose [f t c | c ← cs]
fixTree f (Node e (Variants cs)) = t where
    t = Node e $ Variants [(p, f t c) | (p, c) ← cs]
fixTree f (Node e Stop) = (Node e Stop)
```

Можно полностью посмотреть на граф конфигураций, который обсуждается в статье в разделе 1.3.5 «Свертка».

```
-- demo15
ghci> foldTree $ buildTree (driveMachine prog1) even(sqr(x))
...
```

Это граф также представлен в "читаемой" форме на следующей странице. Конфигурации, совпадающие с точностью до имен переменных, выделены одинаковым цветом.

[1] http://www.haskell.org/haskellwiki/Tying_the_Knot


even(sqr(x))
even(mult(x, x))
x == Z()
x == S(v1)
even(Z())
True()
even(add(x, mult(v1, x)))
x == Z()
x == S(v2)
even(mult(v1, x))
v1 == Z()
v1 == S(v2)
even(Z())
True()
even(add(x, mult(v2, x)))
even(S(add(v2, mult(v1, x))))
odd(add(v2, mult(v1, x)))
v2 == Z()
v2 == S(v3)
odd(mult(v1, x))
v1 == Z()
v1 == S(v3)
odd(Z())
False()
odd(add(x, mult(v3, x)))
x == Z()
x == S(v4)
odd(mult(v3, x))
odd(S(add(v3, mult(v1, x))))
even(add(v3, mult(v1, x)))
v3 == Z()
v3 == S(v4)
even(mult(v1, x))
v1 == Z()
v1 == S(v4)
even(Z())
True()
even(S(add(v4, mult(v1, x))))
even(add(x, mult(v4, x)))
odd(S(add(v3, mult(v1, x))))
even(add(v4, mult(v3, x)))
v4 == Z()
v4 == S(v5)
even(mult(v3, x))
v3 == Z()
v3 == S(v5)
even(Z())
True()
even(S(add(v5, mult(v3, x))))
even(add(x, mult(v5, x)))

1.7. Generator.hs

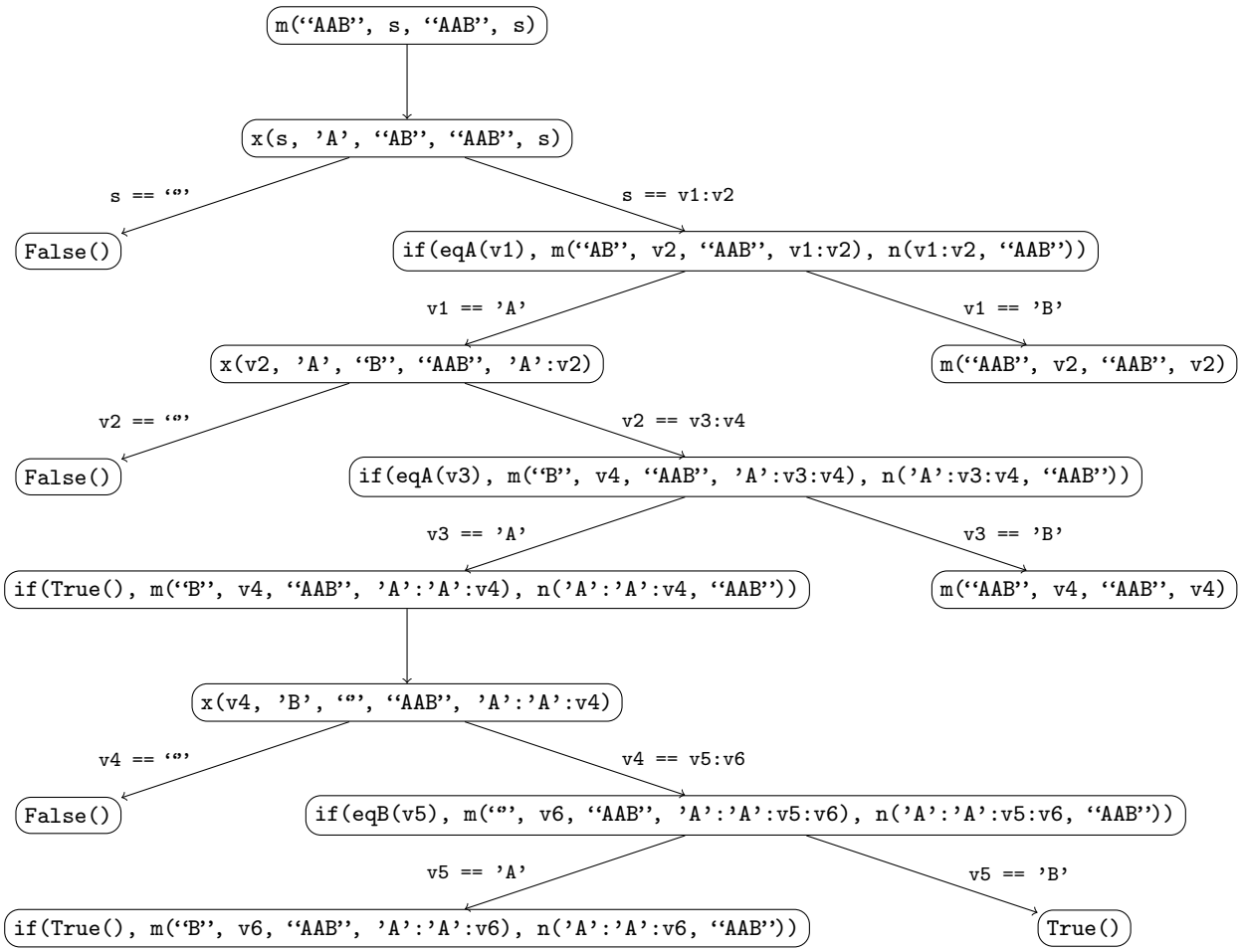
Модуль `Generator` является достаточно большим, – в нем скрыто много различных технических деталей, большинство из которых обусловлено работой с именами.

```
residuate :: Graph Conf → Task
residuate tree = (expr, program) where
    (expr, program, _) = res nameSupply [] tree

res :: NameSupply → [(Conf, Conf)] → Graph Conf → (Conf, Program, NameSupply)
res ns mp (Node e Stop) = (e, Program [] [], ns)

res ns mp (Node (Ctr cname _) (Decompose ts)) = (Ctr cname args, p1, ns1) where
    (args, p1, ns1) = res' ns mp ts

res ns mp (Node (Let (v, _) _) (Decompose ts)) = (e2 // [(v, e1)], p1, ns1) where
    ([e1, e2], p1, ns1) = res' ns mp ts

res (n:ns) mp (Node e (Transient t)) = (fcall, Program ((FDef f1 vs body):fs) gs, ns1) where
    vs = vnames e
    f1 = "f" ++ (tail n)
    fcall = FCall f1 $ map Var vs
    (body, Program fs gs, ns1) = res ns ((e, fcall) : mp) t

res (n:ns) mp (Node e (Variants cs)) = (gcall, Program fs (newGs ++ gs), ns1) where
    vs@(pv:vs') = vnames e
    (vs_, vs'_) = if (isRepeated pv e) && (isUsed pv cs) then (pv:vs, vs) else (vs, vs')
    g1 = "g" ++ (tail n)
    gcall = GCall g1 $ map Var vs_
    (bodies, Program fs gs, ns1) = res' ns ((e, gcall) : mp) $ map snd cs
    pats = [pat | (Contract v pat, _) ← cs]
    newGs = [GDef g1 p vs'_ b | (p, b) ←  (zip pats bodies)]
    isUsed vname cs = any (any (== vname) · vnames · nodeLabel · snd) cs

res ns mp (Node e (Fold (Node base _) ren)) = (call, Program [] [], ns) where
    call = baseCall // [(x, Var y) | (x, y) ← ren]
    Just baseCall = lookup base mp

res' :: NameSupply → [(Conf, Conf)] → [Graph Conf] → ([Conf], Program, NameSupply)
res' ns mp ts = foldl f ([], Program [] [], ns) ts where
    f (cs, Program fs gs, ns1) t = (cs ++ [g], Program (fs ++ fs1) (gs ++ gs1), ns2) where
        (g, Program fs1 gs1, ns2) = res ns1 mp t

isBase e1 (Node _ (Decompose ts)) = or $ map (isBase e1) ts
isBase e1 (Node _ (Variants cs)) = or $ map (isBase e1 · snd) cs
isBase e1 (Node _ (Transient t)) = isBase e1 t
isBase e1 (Node _ (Fold (Node e2 _) _)) = e1 == e2
isBase e1 (Node e2 Stop) = False
```

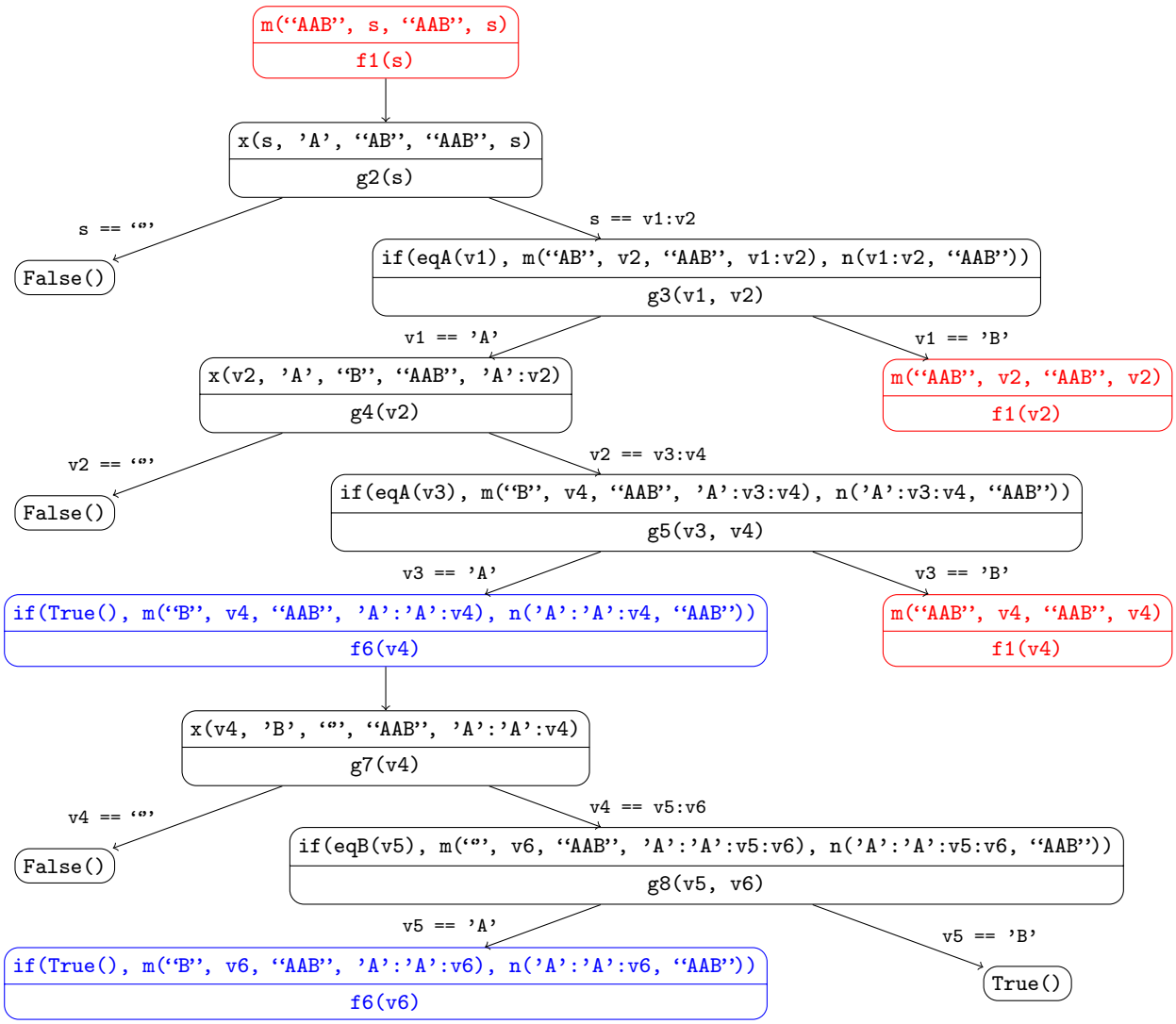
В двух словах, как работает `res`, которую использует главная функция `residuate`. Дерево обходится в глубину слева направо. Результатом обхода любого поддерева является новая конфигурация и программа (список новых определений для f-функций и g-функций). Главная сложность – гарантировать, чтобы у новых функций были уникальные имена.

Подробное объяснение того, как это работает, заняло бы достаточно много места. Мне кажется, что тот, кто действительно интересуется, как это устроено, и так поймет.

Дальше для уяснения, как же оно все-таки работает, приведен граф конфигураций, получающийся при суперкомпиляции нашего КМП-теста из статьи, а затем на него “наложены” новые сгенерированные конфигурации.


m(‘‘AAB’’, s, ‘‘AAB’’, s)
x(s, ’A’, ‘‘AB’’, ‘‘AAB’’, s)
s == ‘‘
s == v1:v2
False()
if(eqA(v1), m(‘‘AB’’, v2, ‘‘AAB’’, v1:v2), n(v1:v2, ‘‘AAB’’))
v1 == ’A’
v1 == ’B’
x(v2, ’A’, ‘‘B’’, ‘‘AAB’’, ’A’:v2)
m(‘‘AAB’’, v2, ‘‘AAB’’, v2)
v2 == ‘‘
v2 == v3:v4
False()
if(eqA(v3), m(‘‘B’’, v4, ‘‘AAB’’, ’A’:v3:v4), n(’A’:v3:v4, ‘‘AAB’’))
v3 == ’A’
v3 == ’B’
if(True(), m(‘‘B’’, v4, ‘‘AAB’’, ’A’:’A’:v4), n(’A’:’A’:v4, ‘‘AAB’’))
m(‘‘AAB’’, v4, ‘‘AAB’’, v4)
x(v4, ’B’, ‘‘’’, ‘‘AAB’’, ’A’:’A’:v4)
v4 == ‘‘
v4 == v5:v6
False()
if(eqB(v5), m(‘‘’’, v6, ‘‘AAB’’, ’A’:’A’:v5:v6), n(’A’:’A’:v5:v6, ‘‘AAB’’))
v5 == ’A’
v5 == ’B’
if(True(), m(‘‘B’’, v6, ‘‘AAB’’, ’A’:’A’:v6), n(’A’:’A’:v6, ‘‘AAB’’))
True()

Просто граф.



Здесь мы, как и прежде, с помощью цвета показываем конфигурации, совпадающие с точностью до переименования. В верхней части узла – старая конфигурация, в нижней – новая (сгенерированная).

```
ghci> let g = ...

-- demo24
ghci> residuate g
f1(s)
f1(s) = g2(s);
g2("") = False();
g2(v1:v2) = g3(v1, v2);
g3('A', v2) = g4(v2);
g3('B', v2) = f1(v2);
g4("") = False();
g4(v3:v4) = g5(v3, v4);
g5('A', v4) = f6(v4);
g5('B', v4) = f1(v4);
f6(v4) = g7(v4);
g7("") = False();
g7(v5:v6) = g8(v5, v6);
g8('A', v6) = f6(v6);
g8('B', v6) = True();
```

1.8. Propotype.hs

Простейший "прототип" суперкомпилятора. Никакого упрощения графа (удаления транзитных переходов), никакого распространения информации.

Простейший свисток:

```
sizeBound = 40
whistle :: Expr → Bool
whistle e@(FCall _ args) = not (all isVar args) && size e > sizeBound
whistle e@(GCall _ args) = not (all isVar args) && size e > sizeBound
whistle _ = False
```

Простейшее обобщение - "вынимается" самое большое подвыражение:

```
generalize :: Name → Expr → Expr
generalize n (FCall f es) =
    Let (n, e) (FCall f es') where (e, es') = extractArg n es
generalize n (GCall g es) =
    Let (n, e) (GCall g es') where (e, es') = extractArg n es

extractArg :: Name → [Expr] → (Expr, [Expr])
extractArg n es = (maxE, vs ++ Var n : ws) where
    maxE = maximumBy ecompare es
    ecompare x y = compare (eType x × size x) (eType y × size y)
    (vs, w : ws) = break (maxE ==) es
    eType e = if isVar e then 0 else 1
```

Построение сворачиваемого дерева. Единственное отличие от `buildTree` в том, что по сигналу свистка делается обобщение:

```
buildFTree :: Machine Conf → Conf → Tree Conf
buildFTree m e = bft m nameSupply e

bft :: Machine Conf → NameSupply → Conf → Tree Conf
bft d (n:ns) e | whistle e = bft d ns $ generalize n e
bft d ns     t | otherwise = case d ns t of
    Decompose ds → Node t $ Decompose $ map (bft d ns) ds
    Transient e → Node t $ Transient $ bft d ns e
    Stop → Node t Stop
    Variants cs → Node t $ Variants [(c, bft d (unused c ns) e) | (c, e) ← cs]
```

Собственно преобразователь:

```
transform :: Task → Task
transform (e, p) =
    residuate $ foldTree $ buildFTree (driveMachine p) e
```

1.9. Deforester.hs

Добавляем `simplify` (удаление транзитных узлов) и получается дефорестация:

```
deforest :: Task → Task
deforest (e, p) =
    residuate $ simplify $ foldTree $ buildFTree (driveMachine p) e

simplify :: Graph Conf → Graph Conf
simplify (Node e (Decompose ts)) =
    Node e (Decompose $ map simplify ts)
simplify (Node e (Variants cs)) =
    Node e (Variants [(c, simplify t) | (c, t) ← cs])
simplify (Node e (Transient t)) | isBase e t =
    Node e $ Transient $ simplify t
simplify (Node e (Transient t)) =
    simplify t
simplify t = t
```

1.10. Supercompiler.hs

Добавляем распространение информации и получаем суперкомпилятор:

```
supercompile :: Task → Task
supercompile (e, p) =
    residuate $ simplify $ foldTree $ buildFTree (addPropagation $ driveMachine p) e

addPropagation :: Machine Conf → Machine Conf
addPropagation m ns e = propagateContract (m ns e)

propagateContract :: Step Conf → Step Conf
propagateContract (Variants vs) =
    Variants [(c, e // [(v, Ctr cn $ map Var vs)]) | (c@(Contract v (Pat cn vs)), e) ← vs]
propagateContract step = step
```

Сравним деревья, порождаемые различными преобразователями (читатель может сам запустить эти задания и посмотреть на деревья)

```
-- demo21
ghci> foldTree $ buildFTree (driveMachine prog2) match("AAB", s)
...

-- demo22
ghci> simplify $ foldTree $ buildFTree (driveMachine prog2) match("AAB", s)
...

-- demo23
ghci> simplify $ foldTree $ buildFTree (addPropagation (driveMachine prog2)) conf2
...
```

...и остаточные задания (приведены в статье):

```
-- demo24
ghci> transform (match("AAB", s), prog2)
...

-- demo25
ghci> deforest (match("AAB", s), prog2)
...

-- demo26
ghci> supercompile (match("AAB", s), prog2)
...
```

1.11. Анти-КМП тест

Теперь обещанные результаты анти-КМП теста.
Для большей «читабельности» результатов, мы установили при запуске этих примеров `sizeBound=10`.
Просто преобразование – в итоге 19 функций.

```
-- demo18
ghci> transform (even(sqr(x)), prog1)
f1(x)
f1(x) = g2(x, x);
f3() = True();
f6() = True();
f7(v2, v1, x) = g8(v2, v1, x);
f10() = False();
f12(v4, v3, x) = g13(v4, v3, x);
f15() = True();
f16(v3, v1, x) = g17(v3, v1, x);
f19() = True();
g2(Z(), x) = f3();
g2(S(v1), x) = g4(x, x, v1);
g4(Z(), x, v1) = g5(v1, x);
g4(S(v2), x, v1) = f7(v2, v1, x);
g5(Z(), x) = f6();
g5(S(v2), x) = g4(x, x, v2);
g8(Z(), v1, x) = g9(v1, x);
g8(S(v3), v1, x) = f16(v3, v1, x);
g9(Z(), x) = f10();
g9(S(v3), x) = g11(x, x, v3);
g11(Z(), x, v3) = g9(v3, x);
g11(S(v4), x, v3) = f12(v4, v3, x);
g13(Z(), v3, x) = g14(v3, x);
g13(S(v5), v3, x) = f7(v5, v3, x);
g14(Z(), x) = f15();
g14(S(v5), x) = g4(x, x, v5);
g17(Z(), v1, x) = g18(v1, x);
g17(S(v4), v1, x) = f7(v4, v1, x);
g18(Z(), x) = f19();
g18(S(v4), x) = g4(x, x, v4);
```

Дефорестация – 11 функций:

```
-- demo19
ghci> deforest (even(sqr(x)), prog1)
g1(x, x)
f4(v2, v1, x) = g5(v2, v1, x);
g1(Z(), x) = True();
g1(S(v1), x) = g2(x, x, v1);
g2(Z(), x, v1) = g3(v1, x);
g2(S(v2), x, v1) = f4(v2, v1, x);
g3(Z(), x) = True();
g3(S(v2), x) = g2(x, x, v2);
g5(Z(), v1, x) = g6(v1, x);
g5(S(v3), v1, x) = g10(v3, v1, x);
g6(Z(), x) = False();
g6(S(v3), x) = g7(x, x, v3);
g7(Z(), x, v3) = g6(v3, x);
g7(S(v4), x, v3) = g8(v4, v3, x);
g8(Z(), v3, x) = g9(v3, x);
g8(S(v5), v3, x) = f4(v5, v3, x);
g9(Z(), x) = True();
g9(S(v5), x) = g2(x, x, v5);
g10(Z(), v1, x) = g11(v1, x);
g10(S(v4), v1, x) = f4(v4, v1, x);
g11(Z(), x) = True();
g11(S(v4), x) = g2(x, x, v4);
```

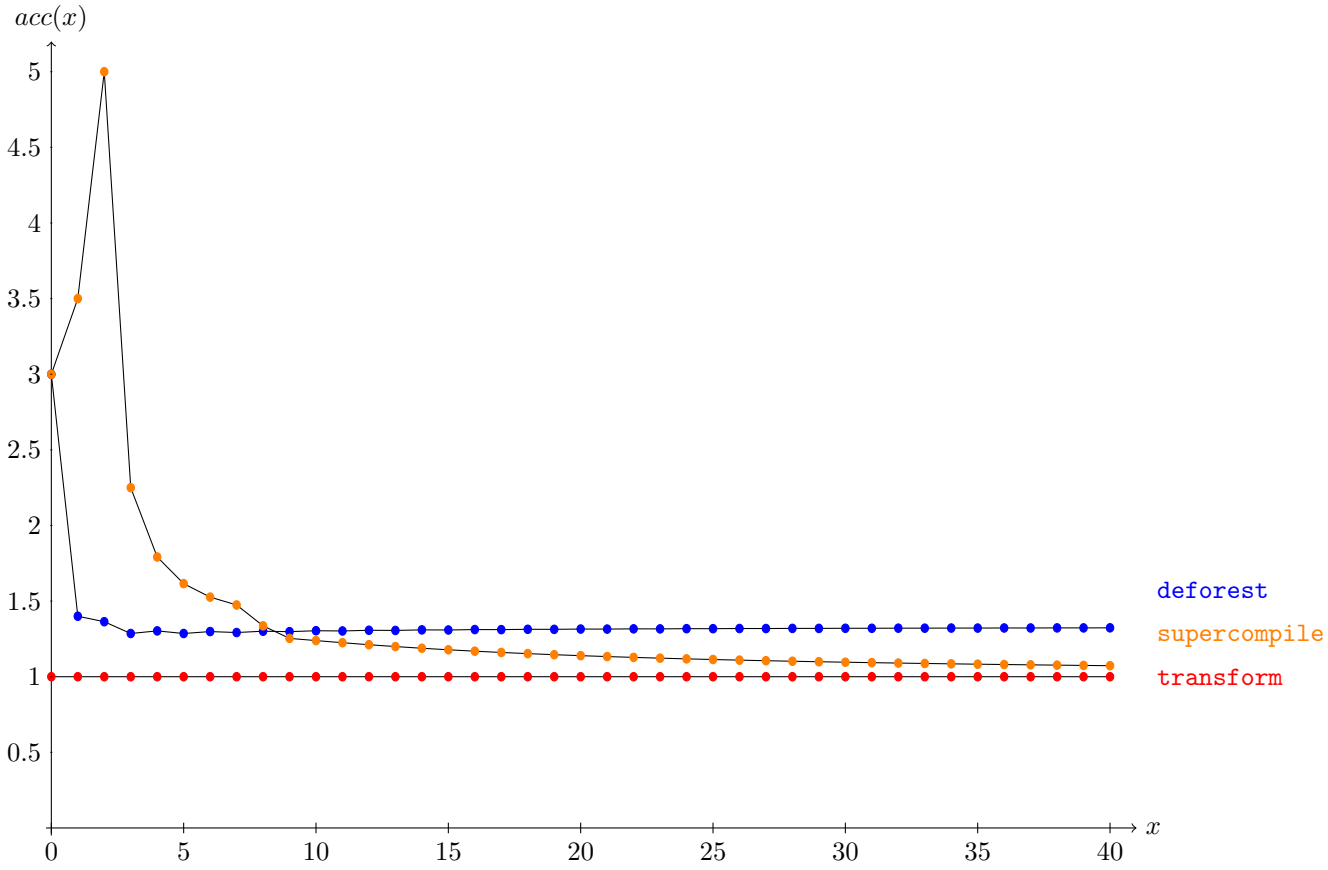
Суперкомпиляция – 34 функции.

```
-- demo20
ghci> supercompile (even(sqr(x)), prog1)
g1(x)
g1(Z()) = True();
g1(S(v1)) = g2(v1);
g2(Z()) = False();
g2(S(v2)) = g3(v2);
g3(Z()) = True();
g3(S(v3)) = g33(S(g4(v3)));
g4(Z()) = S(S(S(S(S(S(Z()))))));
g4(S(v5)) = S(g32(v5, S(S(S(S(g31(v5, S(S(S(S(g30(v5, S(S(S(S(g29(v5, g5(v5))))))))))))))))));
g5(Z()) = Z();
g5(S(v10)) = S(S(S(S(S(g28(v10, g6(v10)))))));
g6(Z()) = Z();
g6(S(v12)) = S(S(S(S(S(g27(v12, g7(v12)))))));
g7(Z()) = Z();
g7(S(v14)) = S(S(S(S(S(S(g26(v14, g8(v14))))))));
g8(Z()) = Z();
g8(S(v16)) = g25(S(S(S(S(S(S(S(v16)))))))), g9(v16, S(S(S(S(S(S(S(v16))))))));
g9(Z(), v18) = Z();
g9(S(v19), v18) = g10(v18, v19);
g10(Z(), v19) = g11(v19);
g10(S(v20), v19) = S(g12(v20, v19));
g11(Z()) = Z();
g11(S(v20)) = g11(v20);
g12(Z(), v19) = g13(v19);
g12(S(v21), v19) = S(g14(v21, v19));
g13(Z()) = Z();
g13(S(v21)) = S(g13(v21));
g14(Z(), v19) = g15(v19);
g14(S(v22), v19) = S(g16(v22, v19));
g15(Z()) = Z();
g15(S(v22)) = S(S(g15(v22)));
g16(Z(), v19) = g17(v19);
g16(S(v23), v19) = S(g18(v23, v19));
g17(Z()) = Z();
g17(S(v23)) = S(S(S(g17(v23))));
g18(Z(), v19) = g19(v19);
g18(S(v24), v19) = S(g20(v24, v19));
g19(Z()) = Z();
g19(S(v24)) = S(S(S(S(g19(v24)))));
g20(Z(), v19) = g21(v19);
g20(S(v25), v19) = S(g24(v25, g22(v19, v25)));
g21(Z()) = Z();
g21(S(v25)) = S(S(S(S(S(g21(v25))))));
g22(Z(), v25) = Z();
g22(S(v27), v25) = S(S(S(S(S(S(g23(v25, g22(v27, v25))))))));
g23(Z(), v28) = v28;
g23(S(v29), v28) = S(g23(v29, v28));
g24(Z(), v26) = v26;
g24(S(v27), v26) = S(g24(v27, v26));
g25(Z(), v17) = v17;
g25(S(v19), v17) = S(g25(v19, v17));
g26(Z(), v15) = v15;
g26(S(v16), v15) = S(g26(v16, v15));
g27(Z(), v13) = v13;
g27(S(v14), v13) = S(g27(v14, v13));
g28(Z(), v11) = v11;
g28(S(v12), v11) = S(g28(v12, v11));
g29(Z(), v9) = v9;
g29(S(v10), v9) = S(g29(v10, v9));
g30(Z(), v8) = v8;
g30(S(v9), v8) = S(g30(v9, v8));
g31(Z(), v7) = v7;
g31(S(v8), v7) = S(g31(v8, v7));
```

```
g32(Z(), v6) = v6;
g32(S(v7), v6) = S(g32(v7, v6));
g33(Z()) = True();
g33(S(v5)) = g34(v5);
g34(Z()) = False();
g34(S(v6)) = g33(v6);
```

Ниже приведены результаты "ускорения" задания `even(sqr(x))`, `prog1` преобразователями `transform`, `deforest` и `supercompile`. Ускорение задания t_2 по отношению к заданию t_1 определялось так, (где `x` – целое, а `x` – соответствующее ему число Пеано):

```
acc t1 t2 x = steps1 / steps2 where
    (_, steps1) = sll_trace t1 [("x"), x]
    (_, steps2) = sll_trace t2 [("x"), x]
```



Из приведенных результатов видно, что при малых x наиболее ощутимо ускорение от суперкомпиляции: при малых x "вычисление" не натыкается на обобщения, которые были сделаны при построении графа (и "выпали" в остаточное задание). При росте , однако, результаты обобщений дают о себе знать и быстрее оказываются "дефорестированные" задания.

Замечание 1 То, что суперкомпиляция может выдавать программы, работающие медленнее, чем дефорестированные, даже при "идеальной" модели стоимости вычисления (у нас скорость вычислений измерялась в шагах интерпретатора), было для меня поначалу большим сюрпризом. Задание объяснить, как такое возможно, достается в качестве награды самому любопытному и дотошному читателю, добравшемуся до конца этого опуса.